Secure Multiparty Shuffle: Linear Online Phase is Almost for Free

Abstract. Shuffle is a frequently used operation in secure multiparty computations, with applications including joint data analysis, anonymous communication systems, secure multiparty sorting, etc. Despite a series of ingenious works, the online (i.e. data-dependent) complexity of malicious secure n-party shuffle protocol remains $\Omega(n^2m)$ for shuffling data array of length m. This potentially slows down the application and MPC primitives built upon MPC shuffle.

In this paper, we study the online complexities of MPC shuffle protocol. We observe that most existing works follow a "permute-in-turn" paradigm, where MPC shuffle protocol consists of n sequential calls to a more basic MPC permutation protocol. We hence raise the following question: given only black-box access to an arbitrary MPC framework and permutation protocol, can we build an MPC shuffle, whose online complexities are independent of the underlying permutation protocol?

We answer this question affirmatively, offering generic transformation from semi-honest/malicious MPC permutation protocols to MPC shuffle protocols with semi-honest/malicious security and only $O(nm)$ online communication and computation. The linear online phase is obtained almost for free via the transformation, in the sense that in terms of overall complexities, the generated protocol equals the protocol generated by naive permute-in-turn paradigm. Notably, instantiating our construction with additive/Shamir secret sharing and corresponding optimal permutation protocol, we obtain the first malicious secure shuffle protocols with linear online complexities for additive/Shamir secret sharing, respectively. These results are to be compared with previous optimal online communication complexities of $O(Bn^2m)$ and $O(n^2m\log m)$ for malicious secure shuffle, for additive and Shamir secret sharing, respectively. We provide formal security proofs for both semi-honest and malicious secure transformations, showing that our malicious secure construction achieves universally composable security. Experimental results indicate that our construction significantly improves online performance while maintaining a moderate increase in offline overhead. Given that shuffle is a frequently used primitive in secure multiparty computation, we anticipate that our constructions will accelerate many real-world MPC applications.

Keywords: multiparty computation, shuffle, random correlation

1 Introduction

Secure multiparty computation (MPC) has various real-world applications. In an MPC scheme, multiple parties jointly compute a function based on their secret inputs, while keeping each party's input confidential from the others.

This is particularly relevant in scenarios such as joint database queries [1][2][3], federated learning [4][5][6], etc.

In this paper, we focus on designing secure and efficient MPC shuffle protocols. In an MPC shuffle protocol, the parties jointly hold an unknown secret data array which is input or generated via MPC functionalities (e.g. secret sharing schemes) while attempting to permute the array by a secret random permutation known to no one. Such an MPC shuffle protocol is a powerful tool in MPC protocol design. For example, the shuffle-then-sort paradigm developed by Hamada et al. [7] represents a class of most efficient MPC comparison-based sorting protocols, which is followed by [8][9]. Another MPC sorting protocol by Hamada et al. [10] that is not comparison-based also utilizes an MPC shuffle protocol as a subroutine, which is one of the most efficient MPC sorting protocols. MPC shuffle protocols have also found direct real-world applications. For example, there is a series of work concentrating on building MPC-based anonymous communication systems [11][12][13], where messages from users are secret shared among all servers. The messages are shuffled and then opened, ensuring that no server knows which message comes from which user.

Compared to other MPC primitives, MPC shuffle often represents the efficiency bottleneck in entire MPC applications, especially when the total number of parties is large [14]. Most existing MPC shuffle protocols follows a "permute-in-turn" paradigm. Specifically, to construct a shuffle protocol, an MPC permutation protocol is developed first, which allows one party to select a secret permutation that is applied to the array. Then, by party permuting in turn, the input array is shuffled by a secret uniform permutation. Following such a paradigm, Keller and Scholl [15] propose a malicious secure MPC shuffle protocol with $O(n^2 m \log m)$ communication, where n is the total number of parties and m is the size of the array. By adopting a variant of permute-in-turn paradigm and enhancing the security with zero-knowledge proofs (ZKPs), Laur et al. [16] propose a malicious secure MPC shuffle protocol with $O(2^n n^{1.5} m \log m)$ communication complexity. As communication complexities here are quadratic in n and super-linear in m, the MPC application built upon these shuffle protocols is potentially slowed down.

Breakthroughs are made by recent works adopting a "two-phase" approach. In this method, parties jointly perform expensive preparation work "offline" before the input arrives. When the input is ready, the parties only need to conduct a significantly reduced amount of work in an "online" phase. Chase et al. [17] propose a very efficient 2-party shuffle protocol with $O(m \log m)$ offline communication and $O(m)$ online communication for semi-honest security. The work of Laud [18] enhance the protocol of Chase et al. [17] to malicious security, and achieve $O(n^2 m \log m)$ offline and $O(n^2 m)$ online communication. Eskandarian and Boneh [13] firstly propose the concept of "shuffle correlation". By generating the correlation with the two-party protocol of [17] in the offline phase, they build a novel shuffle protocol (and an anonymous communication system) that requires only $O(nm)$ online communication, and was believed to be malicious secure. Unfortunately, a more recent study of Song et al. [14] shows that the constructions in

[18][13] are flawed, and achieve only semi-honest security. Although Song et al. [14] propose their own malicious secure construction, it only achieves $O(Bn^2m)$ online complexity where B is introduced by cut-and-choose technique and grows with security parameter. As summarized in Table 1, the best online complexity of existing MPC shuffle protocols remains quadratic in the number of participants. Therefore, it remains an open question how to construct malicious secure MPC shuffle protocols with only linear online overhead.

Table 1. Existing MPC Shuffle Protocols

Protocol	Offline Complexity	Online Complexity	Security	Framework
[16]	$O(1)$	$O(2^n n^{1.5} m \log m)$	malicious	Threshold
[15]	$O(n^2 m \log m)$	$O(n^2 m \log m)$	malicious	Arbitrary
[18]	$O(n^2 m \log m)$	$O(n^2 m)$	semi-honest	SPDZ
[13]	$O(n^2 m \log m + n^3 m)$	$O(nm)$	semi-honest	SPDZ
[14]	$O(Bn^2 m \log m)$	$O(Bn^2 m)$	malicious	SPDZ
ours	$O(n^2 m \log m)$	$O(nm)$[1]	malicious	Arbitrary
ours	$O(Bn^2 m \log m)$	$O(nm)$	malicious	SPDZ

The table lists the upper-bound for both communication and computation complexity. For the online complexities, the two are identical in all above protocols.
n is the number of parties. m is the size of data to be shuffled. B is introduced by cut-and-choose in [14] and grows with security parameter.
[16] works only for threshold secret sharing, e.g. Shamir secret sharing.
We assume the offline complexity of multiplication in SPDZ is $O(n)$ [19].

The online complexity of MPC shuffle protocol is important for several reasons. One is that the online complexity represents the latency experienced at the user end. As the offline task is data-independent, it can be pre-processed before the arrival of user's request. Thus, in real-world MPC application, e.g. MPC-based anonymous communication system [13], it is the online complexities that causes the latency the outside users must endure. Another reason is that MPC shuffle protocol is frequently used as a primitive for building more complicated MPC protocol. During this process, the online complexities are accumulated into the online overheads of higher-level protocols. For example, the MPC oblivious array built by Keller and Scholl [15] requires MPC shuffle as its building block. The shuffle protocol adopted in [15] requires $O(n^2 m \log m)$ online communication, which brings the $n^2 m \log m$ factor to its entire construction. Since MPC oblivious array also serves as a building block for higher-level protocol, this factor could be carried upwards and amplified even further. Such an accumulation could weaken the usability of MPC-based real-world service. Lastly, since offline phase is data-independent, this automatically enables parallel processing

[1] Strictly speaking, the online complexity of our protocol is $\Theta(n(n+m))$. However, as in almost all real-world scenarios, the number of items is (asymptotically) larger than the number of parties (i.e. $n = o(m)$), this is essentially $\Theta(nm)$.

for offline tasks. However, the online computation is data-dependent, which potentially prevents parallelized processing. For example, the MPC radix sort of [10] involves sequential invocations to underlying shuffle protocol, whose online phases thus cannot be batched and parallelized like their offline phases. Hence, it is likely that the communication/computation resource is more (somewhat) expensive in online phase, and an MPC shuffle protocol with better online complexities would bring immediate advances in many other MPC constructions.

In this paper, we handle the online complexities of MPC shuffle protocols by presenting several novel constructions, which transform MPC permutation protocols to MPC shuffle protocols with linear online complexities. At the core of our constructions is a novel technique named shuffle correlation, which helps accelerate the online phase of the shuffle protocol. We define the shuffle correlations for both semi-honest security and malicious security, and show how to utilize our correlations to implement shuffle protocol with linear online phase in both settings. We also show how to generate such shuffle correlations with only generic MPC primitives (i.e. inputting/outputting, addition and multiplication) and a black-box semi-honest/malicious secure MPC permutation protocol. This generality enables automatic generation of MPC shuffle protocols with linear online complexities in several important MPC frameworks. Remarkably, by instantiating our constructions with the SPDZ framework of [19] and the permutation protocol of [14], we obtain a malicious secure MPC shuffle protocol with $O(Bn^2m\log m)/O(nm)$ offline/online communication and computation, which outperforms previous optimal result of $O(Bn^2m\log m)/O(Bn^2m)$ offline/online overheads by [14]. Instantiating with the Shamir secret sharing scheme of [20] and the permutation protocol of [15], we obtain a malicious secure MPC shuffle protocol with $O(n^2m\log m)/O(nm)$ offline/online communication and computation, outperforming previous optimal result of $O(n^2m\log m)/O(n^2m\log m)$ offline/online overheads by [15]. Besides the significant improvements in online complexities, the offline complexities match currently optimal results, respectively. Thus, MPC shuffle protocols with linear online phase are obtained "almost for free".

Fig. 1 below demonstrates how our constructions work: by picking an MPC framework on the left, picking an MPC permutation protocol (compatible with the framework) in the middle, and finally applying our corresponding semi-honest/malicious secure shuffle correlation and protocols, one obtains automatically an MPC shuffle protocol on the right with linear online complexities, with security level and offline complexities varying by choice of framework and permutation protocol. Note that this illustration is not exhaustive, i.e. there may exist other feasible MPC frameworks or permutation protocols not listed in the diagram.

Our contributions are summarized as follows.

1. We refine the concept of shuffle correlation, define it for both the semi-honest security and the malicious security, and show how our definitions can be used to implement MPC shuffle protocol with linear online communication and computation overheads. Our definition is generic, in the sense that it can

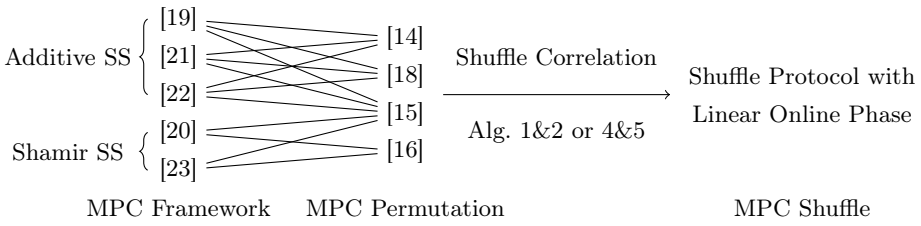

Fig. 1. Illustration of How Our Constructions Work

be (as we will demonstrate) generated with mere black-box access to basic MPC primitives and an MPC permutation protocol. Contrasting previous definition in [13], our definitions of shuffle correlations support malicious security and are compatible with various MPC frameworks and MPC permutation protocols. This compatibility brings potential advances in building MPC shuffle protocol in various MPC frameworks.

2. We study the instantiation of our protocols and present several MPC shuffle protocols with both online and offline complexities currently optimal. Remarkably, our constructions directly imply actively secure shuffle protocols with linear online phase in additive secret sharing scheme and Shamir secret scheme, which are to be compared with currently optimal online communication of $O(Bn^2m)$ and $O(n^2m\log m)$, respectively. This is achieved by combining our construction with the SPDZ framework of [19] and the permutation protocol of [14], and combining Shamir secret sharing scheme of [20] and the permutation protocol of [15], respectively.
3. We formally prove the security of our constructions. In malicious security in particular, we prove that our construction is universally composable secure (UC secure) in $\mathcal{F}_{\text{MPC}}$-hybrid model, where $\mathcal{F}_{\text{MPC}}$ is an ideal functionality, which supports basic MPC arithmetic operations and an MPC permutation. This means our protocol can be instantiated with any MPC framework and permutation protocol, and retains a corresponding security level.
4. Experiments are done to verify theoretical analysis. The results confirm that compared to the shuffle protocol generated by naive permute-in-turn paradigm, our protocol consumes much less online running time and online communication resource, with moderate increasing in offline overheads.

The rest of this paper is organized as follows. In Section 2, we briefly review previous works in literature. In Section 3, we present the primitives required for our constructions, discuss the "permute-in-turn" paradigm and give the definition of security. In Section 4 we discuss the main challenges in constructing shuffle protocols with linear online complexities, and present our core insights in overcoming them. Section 5 shows our definition for semi-honest shuffle correlation and how to generate/use it, and Section 6 for malicious case. For clarity of description and security, the construction given in Section 6 has $O(n^2m)$ online complexity, which will be optimized to $O(nm)$ in Section 7 via a standard batch

checking technique. The fully optimized malicious shuffle protocol is formally presented in Section B, before proving its security. In Section 8, we show the result of our experiments. Due to the page limit, we defer our security proof to appendix. The formal proofs of semi-honest and malicious security are presented in Section A and Section B, respectively. We will case study several candidates that could be used to instantiate our construction to obtain MPC shuffle protocol for different MPC frameworks in Section C. Lastly in appendix, we discuss several important issues w.r.t. the usage of shuffle protocol in practice in Section D.

2 Related Works

This first shuffle protocol dates back to the seminal work of Chaum [24], appearing by the concept of mix-net. From the view of modern cryptography, the work of Chaum [24] implements anonymous communication in a server-aided scheme, with semi-honest server. The work also inspires a serial of works by the concept of "decryption shuffle", which involves the sender encrypting its message with a sequential public keys of servers, and then the server decrypting and permuting the message in turn. The construction requires a linear communication overhead and a moderate computation of asymmetric encryption.

The work of Chaum [24] considers only semi-honest case, in the sense that all the servers must follow the protocol honestly, otherwise the security may completely break down. There is a series of works enhancing the security of the protocol [25][26][27][28]. The most common approach is via zero-knowledge proof, i.e. each server generates a proof which proves that it has permuted the ciphertext honestly, while hiding the permutation applied. This approach, while being effective, is generally expensive and heavy in computation.

Very recently, a new construction for shuffling via multiparty computation (MPC) appears, which offers potentially a different approach to achieve security against malicious adversary. Chase et al. [17] design a shuffle protocol for two-party computation, with obliviously punctured vector (OPV). Due to the invention of oblivious transfer extension, the OPV can be generated considerably fast. The core technique is a protocol that allows one of the parties to permute the secret shared data with a permutation it chooses, and the shuffle protocol consists of each party permuting once. Although the construction is specified for two-party computation, a direct extension to n-party with $O(n^2m)$ online communication is possible, as is shown in [18], in an attempt to construct malicious secure shuffle protocol. [13] also construct a shuffle protocol for anonymous communication system, which is claimed to be malicious secure with only $O(nm)$ online complexity. It seems that malicious secure MPC shuffle with linear online complexity can be easily derived from such a construction.

However, a more recent study by Song et al. [14] points out that the implementations of [18] and [13] are not secure against malicious adversary. By constructing a selective abort attack to these two constructions, Song et al. [14] demonstrate that malicious adversary could bypass the correctness check

of [18] and [13] with non-negligible probability, while gaining information about permutation applied upon success. They hence also designed a shuffle protocol for MPC, which has $O(Bn^2m)$ online communication complexity, where B is a parameter introduced by cut-and-choose technique.

From all the above constructions, one observation is that all these constructions follow a "permute-in-turn" paradigm. That is, letting parties permute the items in turn, and the result is correctly shuffled with unknown permutation. All the constructions of [24][25][26][17][18][13][14] follow this paradigm, even though many of them are not designed for multiparty computation. Although [16] adopts a slightly different approach, it is in essence letting groups of parties permute in turn, which can be seen as a variant of permute-in-turn paradigm. However, to the best of our knowledge, currently all malicious secure shuffle protocols have their online phase executing the (online phase of) permutation protocol directly, e.g. in [16], [15] and [14], parties repeatedly form validly shared permuted secret for n times, which results in $\Omega(n^2m)$ communication and computation. This makes the online phase communication and computation heavy, which is undesirable for many real world applications that require quick response.

Another perspective to view this issue is from the concept of shuffle correlation. A shuffle correlation is a set of random values that are correlated, which helps to implement shuffle operation in the online phase. The works of [17][18][14] utilize permutation correlations that help them perform online permutation. Thus, we may say that they have defined implicitly their shuffle correlation to be the set of n permutation correlations, each for one party. This results in honestly/separately permuting the array n times in the online phase, leading to an $\Omega(n^2m)$ online communication and computation. Though [13] first propose explicitly the concept of shuffle correlation, their shuffle correlation achieves only semi-honest security. Also, as its generation relies heavily on the two-party permutation protocol of [17] and is specialized for additive secret sharing, it is not compatible with other MPC frameworks and permutation protocols.

Although the definition in [13] does not achieve malicious security, it accomplishes linear online complexity with a relatively small constant factor. This raises the question whether it is possible to build a maliciously secure MPC shuffle based on such a definition. However, this turns out to be difficult. The primary obstacle is that all operations (i.e., addition and permutation) are linear throughout the process, yet the parties cannot verify the correctness of intermediate values. During the process, each party locally adds a random mask to the received messages, permutes them, and sends them to the next party. To ensure the secrecy of the underlying data, all values sent and received by a party are independently random, preventing an honest party P_i from confirming the correctness of the values it receives. This inherent characteristic of shuffle correlation makes it vulnerable to selective failure attacks, as constructed by Song et al. [14], where a malicious party introduces additive errors in the intermediate values and later attempts to correct them. Such an attack can succeed with non-negligible probability and, upon success, leaks information about the permutations chosen by honest parties. Therefore, it seems implausible to rely

on the construction of [13] to achieve both malicious security and linear online complexity.

We remark that, such an attack forms a main challenge in constructing malicious secure shuffle correlation with linear online complexities. Facing such an attack, each party must somewhat be able to verify that it is receiving correct messages. However, this check cannot be done by the party alone, as the privacy requires that the received message must appear uniformly random to the party. Since a malicious party can simply send garbage to honest party, it is not possible for a party to locally check the correctness of the message. Worse still, checking this m-long vector by MPC primitives seems also a bad idea, as merely sharing it would require $O(nm)$ communication, which is already $O(n^2m)$ for n parties. We refer the reader to Section 4 for more involved discussions on these challenges and our core insights for overcoming them.

3 Preliminary

3.1 Basic Notations

Throughout this paper, it is assumed that all numbers and operations are in a large prime field $\mathbb{F}$, with $|\mathbb{F}| \geq 2^\kappa$ for any statistical security parameter κ fixed a priori. Suppose there are n parties $P_1, P_2, ..., P_n$, and m field elements to shuffle. Denote by

$$[t] \coloneqq \{1, 2, ..., t\},$$

the set of positive integers ranging from 1 to t for any positive integer t.

Throughout this paper, we use lowercase letters for integers, e.g. x, y, z, and bold font letters for vectors, e.g. $\mathbf{x}, \mathbf{y}, \mathbf{z}$. We denote by $\mathbf{x}(i)$ or x_i the i-th entry of vector $\mathbf{x}$. Notation $\mathbf{x}(i)$ is helpful when we are dealing with vectors with their own indices, e.g. vector $\mathbf{x}_1$ and $\mathbf{x}_2$. As we are mostly dealing with vectors of length m, hence any vector is of length m unless stated otherwise. For an m-permutation $\pi : [m] \to [m]$, define

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

Note that permutation is additively homomorphic, i.e.

$$\pi(\mathbf{x} + \mathbf{y}) = \pi(\mathbf{x}) + \pi(\mathbf{y}),$$

where the addition of two vectors is defined to be entry-wise.

The concatenation of two permutations π_2, π_1 is another permutation, which is denoted as $\pi_2 \circ \pi_1$ and satisfies

$$\forall \mathbf{x} \in \mathbb{F}^m : \pi_2 \circ \pi_1(\mathbf{x}) = \pi_2(\pi_1(\mathbf{x})).$$

For later notation convenience, for a sequence of permutations $\pi_1, \pi_2, ..., \pi_n$, we denote

$$\overline{\pi}_i \coloneqq \pi_i \circ \pi_{i-1} \circ \cdots \circ \pi_1.$$

Executing a sub-protocol is written as:

$$\Pi(P_i : x, y, [\![z]\!]),$$

where "Π" is the name of the protocol. Parameter "$P_i : x$" means that this protocol takes a private input x from party P_i. Parameter "y" means that this protocol takes a public constant y from all parties. Parameter $[\![z]\!]$ means that z is a value stored at $\mathcal{F}_{\text{MPC}}$. We will explain what "stored at $\mathcal{F}_{\text{MPC}}$" means in the next subsection.

3.2 Primitives

We assume that secure arithmetic MPC primitives are available, including inputting a secret input, opening a secret, generating random values, computing additions and multiplications. We assume also an ideal functionality of permutation protocol, which takes as input a shared vector (i.e. shared entries) and a permutation known to one party, and permutes the vector accordingly. By assuming such functionalities in a black-box manner, our constructions can be applied to a generic class of MPC framework to obtain shuffle protocols with linear online complexity with fairly small constant.

To formalize, we follow the approach adopted by Escudero et al. [29], where an ideal arithmetic MPC functionality is viewed as a Turing machine with internal state. The functionality interacts honestly with all parties, taking inputs and (restricted) commands from them and changing its internal state accordingly. We assume that the ideal arithmetic MPC functionality $\mathcal{F}_{\text{MPC}}$ supports following commands:

- $\Pi_{\text{input}}(P_i : x, \text{id})$, which takes as input a field element x from P_i and stores it as (id, x). "id" is a unique identifier that all parties agree on, which can be seen as a memory address of $\mathcal{F}_{\text{MPC}}$.
- $\Pi_{\text{input}}(y, \text{id})$, which takes as input a public constant field element y and stores it as (id, y).
- $\Pi_{\text{rand}}(\text{id})$, which draws a uniform random value $r \in \mathbb{F}$ and stores it as (id, r).
- $\Pi_{\text{add}}(\text{id}, \text{id}_1, C)$, which retrieves (id_1, x) from the memory and stores $(\text{id}, x + C)$, where C is a public constant.
- $\Pi_{\text{add}}(\text{id}, \text{id}_1, \text{id}_2)$, which retrieves (id_1, x_1) and (id_2, x_2) from the memory and stores $(\text{id}, x_1 + x_2)$.
- Π_{mul} works same as Π_{add}, except it's computing multiplications.
- $\Pi_{\text{open}}(\text{id})$, which retrieves (id, x) and outputs x to the adversary. If the adversary replies with "continue", then $\mathcal{F}_{\text{MPC}}$ sends also x to honest parties; otherwise it sends "abort" to the honest parties.
- $\Pi_{\text{open}}(P_i, \text{id})$, which retrieves (id, x) and outputs x to party P_i.
- $\Pi_{\text{send}}(P_i : x, P_j)$ and $\Pi_{\text{broadcast}}(P_i : x)$, which sends the message x from P_i to P_j or broadcasts it to all parties. The communication complexity of broadcasting an item is assumed to be $O(n)$.
- $\Pi_{\text{perm}}(P_i : \pi, (\text{id}_\text{j})_{j=1}^m, (\text{id}'_\text{j})_{j=1}^m)$, which takes a secret m-permutation from P_i, retrieves (id_j, x_j) and stores $(\text{id}'_j, x_{\pi(j)})$.

For semi-honest adversary, we do not need Π_{mul}, and $\mathcal{F}_{\text{MPC}}$ can thus be instantiated with Shamir secret sharing or additive secret sharing without MAC. For malicious adversary, $\mathcal{F}_{\text{MPC}}$ may be instantiated with additive secret sharing (e.g. [22][21][30]), Shamir's secret sharing (e.g. [23][31]), etc., based on specific security requirement and application scenario.

For notation simplicity, we denote by $[\![x]\!]$ a secret value x stored by $\mathcal{F}_{\text{MPC}}$. The reader familiar with secret sharing schemes may also understand this as "sharing x among all parties", since $\mathcal{F}_{\text{MPC}}$ will be implemented by secure multi-party computation in practice. To distinguish between the case of semi-honest adversary and malicious adversary, we write $\langle x \rangle$ for secret stored at malicious secure ideal functionality. We denote also

$$[\![d]\!] \leftarrow [\![a]\!] \cdot [\![b]\!] + [\![c]\!]$$

the process of computing $a \cdot b + c$ and store it in ideal functionality. Note that different symbols refer to different identifiers for $\mathcal{F}_{\text{MPC}}$, which allows us to omit the identifier in later description. We denote also $[\![\mathbf{x}]\!]$ as storing each entry separately in ideal MPC. Hence, we also denote

$$[\![\mathbf{z}]\!] \leftarrow [\![a]\!] \cdot [\![\mathbf{x}]\!] + [\![\mathbf{y}]\!]$$

the process of computing $ax_1 + y_1, ..., ax_m + y_m$ and storing it as $\mathbf{z}$.

Similarly, we write

$$[\![\mathbf{x}']\!] \leftarrow \Pi_{\text{perm}}(P_i : \pi, [\![\mathbf{x}]\!]),$$

where $\mathbf{x} = (x_1, x_2, ..., x_m)$ is a vector of length m and

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

For malicious setting, we assume also a version of batched permutation protocol, where

$$(\langle \pi(\mathbf{x}_1) \rangle, ..., \langle \pi(\mathbf{x}_t) \rangle) \leftarrow \Pi_{\text{perm}}(P_i : \pi, \langle \mathbf{x}_1 \rangle, ..., , \langle \mathbf{x}_t \rangle).$$

This is a natural requirement for permutation protocol, since in real world application, what is to be shuffled is usually a long vector (e.g. rows of the database) instead of a single field element. As explicit examples, the protocols in [16][17][18][13][14] all support such an operation.

To conclude, we assume an ideal functionality $\mathcal{F}_{\text{MPC}}$ that supports the above basic arithmetic commands and Π_{perm}. The construction presented in this paper will be hence proved secure under $\mathcal{F}_{\text{MPC}}$-hybrid model.

3.3 Permute-in-Turn Paradigm

To shuffle a secret shared vector $[\![\mathbf{x}]\!]$ into some $[\![\pi(\mathbf{x})]\!]$ with π unknown to any party, most previous works follow a "permute-in-turn" paradigm. That is, suppose we now have a permutation protocol Π_{perm}, which securely implement the functionality

$$[\![\pi(\mathbf{x})]\!] \leftarrow \Pi_{\text{perm}}(P_i : \pi, [\![\mathbf{x}]\!]).$$

Then the shuffle protocol can be implemented as sequential calls to Π_{perm}. That is, from 1 to n, each party selects a random permutation π_i, and compute sequentially for $i = 1, 2, ..., n$

$$[\![\mathbf{y}_i]\!] \coloneqq [\![\pi_i(\mathbf{y}_{i-1})]\!] \leftarrow \Pi_{\text{perm}}(P_i : \pi_i, [\![\mathbf{y}_{i-1}]\!]),$$

where $\mathbf{y}_0 \coloneqq \mathbf{x}$. Note that

$$[\![\mathbf{y}_n]\!] = [\![\pi(\mathbf{x})]\!] = [\![\pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1(\mathbf{x})]\!],$$

where π is known to no party. In addition, as long as there is at least one honest party P_i that has chosen its permutation π_i uniformly random, the resulted π will be uniformly random.

What is achieved in this paper is to transform above permute-in-turn paradigm into two phases, an offline phase $\text{Shuffle}_{\text{off}}$ and an online phase $\text{Shuffle}_{\text{on}}$. The reason for doing so is to move the most computation and communication overheads to the offline phase, and in specific, the n invocations of basic Π_{perm} protocols. The online complexities of our constructions are hence independent of the implementation of Π_{perm}, while most constructions in previous works are not. For example, the complexity of the online phase of our protocol is linear in both the number of parties and the length of vector, which is not previously achieved under malicious security. (C.f. Table 1 for a review of existing works.)

3.4 Security Model

Throughout this paper, we consider a static adversary, i.e. the corrupted parties are chosen and fixed before protocol starts. We consider the case where a majority of parties could be corrupted, i.e. dishonest majority. Note that, however, our constructions is also applicable to the honest majority setting.

Our first construction guarantees semi-honest security, under $\mathcal{F}_{\text{MPC}}$-hybrid model where $\mathcal{F}_{\text{MPC}}$ is semi-honest secure. In the semi-honest security, the adversary corrupting some parties is assumed to be following the protocol honestly, but nevertheless may do some extra computations to gain the information about other party's input. As an analogue to universally composable security (UC security), we allow the adversary $\mathcal{A}$ to know the entire vector $\mathbf{x}$ that is to be shuffled. This makes sense, since in practice when our protocol is used as subroutine, some entries of $\mathbf{x}$ might come from the input of corrupted parties. Hence, we simply assume that the adversary knows the entire vector $\mathbf{x}$. The security of the protocol states that: even if the adversary combines $\mathbf{x}$ and its view during the execution of the protocol, it cannot guess the input π_i of any honest party P_i better than purely random, i.e. information-theoretic security.

Definition 1 (Semi-honest Security). *Suppose there is a shuffle protocol Π, which takes as input a vector $[\![\mathbf{x}]\!]$ stored at $\mathcal{F}_{\text{MPC}}$, and π_i from each party P_i, where each party P_i chooses π_i independently and uniformly. It interacts with parties and $\mathcal{F}_{\text{MPC}}$, and finally makes $\mathcal{F}_{\text{MPC}}$ store*

$$[\![\pi(\mathbf{x})]\!] = [\![\pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1(\mathbf{x})]\!].$$

Suppose adversary $\mathcal{A}$ corrupts the parties in set $T \subsetneq [n]$, and will follow the protocol honestly. If for any $\mathbf{x}$ and $\{\pi_i\}_{i\in T}$,

$$I(\mathbf{x}, \text{view}_\Pi^T(\mathbf{x}, \pi_1, \pi_2, ..., \pi_n); (\pi_i)_{i\notin T}) = 0,$$

then the protocol Π is said to be semi-honest secure, where I is the mutual information, view_Π^T is the view of corrupted parties in one execution of protocol Π and π_i is the input of party P_i.

Stated otherwise, $(\pi_i)_{i\notin T}$ is independent (in a probability theory sense) of $\mathbf{x}$ and the view of adversary.

In above definition, π_i is the input of party P_i, which is by design a uniformly random permutation and is the only input of party P_i in our construction. In later security proof, we will prove semi-honest security for protocol

$$\Pi = \text{Shuffle}_{\text{on}} \circ \text{Shuffle}_{\text{off}},$$

i.e. the combination of two phases of shuffle protocol is secure.

Our second construction guarantees security against a malicious adversary. We will prove universally composable security (UC security) for this construction. The definition of UC security is as follows.

Definition 2 (Universally Composable Security [32], Sketch). *Suppose there is an environment $\mathcal{E}$ and an adversary $\mathcal{A}$ that controls the corrupted parties $P_i \in T \subsetneq [n]$. Let protocol Π be an implementation of the ideal functionality $\mathcal{F}_{\text{shuffle}}$.*

Consider two games. One happens between $\mathcal{E}$, $\mathcal{A}$, $\mathcal{F}_{\text{MPC}}$ and honest parties $P_i \in \bar{T} = \{P_i \notin T\}$, executing real protocol Π. Another happens between $\mathcal{E}$, simulator $\mathcal{S}$ and ideal functionality $\mathcal{F}$ (which is $\mathcal{F}_{\text{MPC}}$ equipped with additional ideal command Π_{Shuffle}), simulating the view of adversary in ideal execution. In each game, $\mathcal{E}$ chooses inputs of all honest parties, and sends them to either $P_i \in \bar{T}$ or $\mathcal{F}$. When the corrupted parties controlled by $\mathcal{A}$ need to send a message, $\mathcal{E}$ decides it, and when $\mathcal{A}$ receives anything, it reports to $\mathcal{E}$. In this "real" execution, if the protocol does not abort, $\mathcal{E}$ receives outputs of honest parties from honest parties.

*In the "ideal" world, $\mathcal{S}$ will not receive the input of any party. Nevertheless, $\mathcal{S}$ needs to deduce the purported inputs of the corrupted parties, send them to $\mathcal{F}$, and is then informed by $\mathcal{F}$ of the output of the protocol for corrupted parties. If $\mathcal{E}$ does not demand $\mathcal{S}$ to abort, and the protocol ends without abort, $\mathcal{S}$ sends "*continue*" to $\mathcal{F}$, who then sends all outputs of honest parties to $\mathcal{E}$.*

$\mathcal{E}$ keeps interacting with $\mathcal{A}/\mathcal{S}$ during the entire process, while gaining information and doing its own computation. When $\mathcal{E}$ halts, it outputs a bit, representing its guess on which game it is playing.

The protocol Π securely implemented $\mathcal{F}$, if there exists a simulator $\mathcal{S}$, such that $\mathcal{E}$ cannot distinguish what game it is playing, i.e.

$$\left|\Pr\left[1 \leftarrow (\mathcal{E} \leftrightarrows \Pi_{\bar{T},\mathcal{A}})\right] - \Pr\left[1 \leftarrow (\mathcal{E} \leftrightarrows \mathcal{F}_{\bar{T},\mathcal{S}})\right]\right| < \epsilon = O(2^{-\kappa}).$$

We remark that, the above definition considers only a dummy adversary, which acts according to $\mathcal{E}$'s command and sends everything it receives to $\mathcal{E}$. This is equivalent to a more "intelligent adversary", as $\mathcal{E}$ could perform all computations and decide the (malicious) actions. Also, the definition does not limit the computation resource of the environment, and hence achieves statistical security. This is achievable, as we are proving it under $\mathcal{F}_{\text{MPC}}$-hybrid model. Also, the above definition is merely a sketch and misses many important details in constructing simulator. Nevertheless, we will present a more formal definition before presenting a formal security proof.

Note also that in practice, the implementation of $\mathcal{F}_{\text{MPC}}$ or protocol Π_{perm} (that is chosen by the developer) may achieve only simulation/standalone security instead of UC security. Thus, after replacing the functionalities with real implementations, the UC security cannot be achieved at all. Nevertheless, it should be easy to modify our proof to prove that the combination achieves simulation/standalone security.

4 Technical Overview

Before diving into technical details, here we briefly review former attempts in constructing shuffle correlation, as well as the main challenges at hand and our core insights for overcoming them.

4.1 Semi-honest Shuffle Correlation

In the work of [17], a two-party share translation protocol is proposed. For shuffling m-long vector, this protocol outputs $(\pi, \mathbf{\Delta})$ to party P_1 and $(\mathbf{a}, \mathbf{b})$ to party P_2, such that π is chosen (uniformly) by P_1 and $\mathbf{\Delta} = \pi(\mathbf{a}) + \mathbf{b}$.

Such an output can be viewed as a two-party permutation correlation. In the online phase, to permute additively shared $\mathbf{x} = \mathbf{x}_1 + \mathbf{x}_2$ (where P_i holds $\mathbf{x}_i$), P_2 sends $\mathbf{x}_2 + \mathbf{a}$ to P_1, and parties locally compute and hold

$$P_1 : \mathbf{x}'_1 \leftarrow \pi(\mathbf{x}_1 + (\mathbf{x}_2 + \mathbf{a})) - \mathbf{\Delta},$$
$$P_2 : \mathbf{x}'_2 \leftarrow \mathbf{b}.$$

This shares $\mathbf{x}' = \pi(\mathbf{x})$ among parties. Repeating above process for P_2, this results in a two-party shuffle protocol with $O(m)$ online communication.

It is shown in [18] that the above construction can be extended to n-party cases, with $O(n^2)$ invocations to the basic share translation protocol and $O(n^2 m)$ online communication. To overcome the n^2 factor, Eskandarian and Boneh [13] first propose shuffle correlation, which is defined as follows.

Definition 3 (Shuffle Correlation in [13]). *The n-party shuffle correlation is a situation where for $i = 1, ..., n$, P_i holds:*

1. *random vectors $\mathbf{a}_i, \mathbf{b}_i, \mathbf{a}'_i \in \mathbb{F}^m$ and*
2. *a random permutation $\pi_i : [m] \to [m]$.*

P_n holds in addition a vector $\mathbf{\Delta}_n \in \mathbb{F}^m$, such that

$$\mathbf{\Delta}_n = \pi_n(...(\pi_2(\pi_1(\sum_{i=2}^{n} \mathbf{a}_i) + \mathbf{a}'_1) + \mathbf{a}'_2)... + \mathbf{a}'_{n-1}) - \sum_{i=1}^{n-1} \mathbf{b}_i.$$

Suppose $\mathbf{x}$ is additively shared as $\mathbf{x}_1, ..., \mathbf{x}_n$, with P_i holding $\mathbf{x}_i$. With such a correlation, shuffle can be performed by first letting P_i $(i \neq 1)$ send $\mathbf{z}_i \leftarrow \mathbf{x}_i - \mathbf{a}_i$ to P_1, who then computes and sends to P_2

$$\mathbf{y}_1 \leftarrow \pi_1(\sum_{i=2}^{n} \mathbf{z}_i + \mathbf{x}_1) - \mathbf{a}'_1,$$

and locally holds $\mathbf{x}'_1 \leftarrow \mathbf{b}_1$. Then for $2 \leq i \leq k-1$, each P_i computes and sends to P_{i+1}

$$\mathbf{y}_i \leftarrow \pi_i(\mathbf{y}_{i-1}) - \mathbf{a}'_i,$$

and locally holds $\mathbf{x}'_i \leftarrow \mathbf{b}_i$. Lastly, P_n holds $\mathbf{x}'_n \leftarrow \pi_n(\mathbf{y}'_{n-1}) + \mathbf{\Delta}_n$. Thus, all parties additively share $\mathbf{x}'$, with

$$\mathbf{x}' = \pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1(\mathbf{x}).$$

Generating above $\mathbf{\Delta}$ seems to require $\Omega(n)$ sequential invocations of Π_{perm}, which is $\Omega(n)$ rounds. To avoid such suboptimal round complexity, [13] further defines

1. $\mathbf{a}_i \leftarrow \mathbf{a}_{1,i}$, $\mathbf{b}_i \leftarrow \mathbf{b}_{n,i}$.
2. $\mathbf{a}'_i \leftarrow \sum_{j\in[n]\setminus\{i+1\}} \mathbf{a}_{i+1,j} - \sum_{j\in[k]\setminus\{i\}} (\mathbf{b}_{i,j} + \mathbf{\Delta}_{i,j})$.
3. $\mathbf{\Delta}_k \leftarrow \sum_{j\in[k-1]} \mathbf{\Delta}_{k,j}$.

Wherein each $(\mathbf{a}_{i,j}, \mathbf{b}_{i,j}, \mathbf{\Delta}_{i,j})$ is from invoking the share translation protocol of [17] for P_i and P_j. To compute $\mathbf{a}'_i$, [13] suggests each party P_i shares each $\mathbf{a}_{i,j}, \mathbf{b}_{i,j}, \mathbf{\Delta}_{i,j}$ back, which requires in addition $O(n^3 m)$ communication for sharing $O(n^2)$ random values.

From above, it is clear this previous definition of shuffle correlation is specialized for additive secret sharing, whose generation relies on the two-party share translation protocol of [17] and requires $O(n^2 m \log m + n^3 m)$ communication.

Contrasting to previous definition, our definition (c.f. Definition 4) for semi-honest shuffle correlation is compatible with general MPC framework. Our definition follows a similar pattern of above, in that each P_i sends $\mathbf{y}_i = \pi_i(\mathbf{y}_{i-1} + \mathbf{z}_i)$ to P_{i+1}, and finally P_n broadcasts $\mathbf{y}_n$. By subtracting $\mathbf{y}_n$ with

$$[\![\mathbf{\Delta}]\!] = [\![\pi_n(\pi_{n-1}(...\pi_1(\mathbf{z}_1) + \mathbf{z}_2)... + \mathbf{z}_n)]\!],$$

the parties share permuted $\mathbf{x}$ as desired.

Furthermore, in our generation of shuffle correlation, we decompose $\mathbf{z}_i$ as $\mathbf{r}_i - \pi_{i-1}(\mathbf{r}_{i-1})$, and make $\mathbf{\Delta} = \pi_n(\mathbf{r}_n)$. This allows parallel invocations to Π_{perm} and accelerates the generation of these random resources. Also, our definition thus requires only n-party-shared randomness (instead of sharing the randomness from the output of two-party protocol), which avoids the $O(n^3 m)$ terms and achieves $O(n^2 m \log m)$ offline communication. Thus, besides the compatibility with other MPC framework, our definition also achieves better offline complexities (compared to [13]) when applied in semi-honest additive secret sharing.

4.2 Malicious Shuffle Correlation

A major challenge we solve in the paper emerges in the attempt to achieve malicious security. In such an attempt, [13] suggests that parties generate and share in addition a random vector $\mathbf{g}$ and compute and share (via MPC multiplication) $\mathbf{h}$, such that $\mathbf{h}(i) = \mathbf{x}(i) \cdot \mathbf{g}(i)$. By shuffling $\mathbf{g}, \mathbf{h}$ along with $\mathbf{x}$, and checking at last if $\mathbf{h}'(i) = \mathbf{x}'(i) \cdot \mathbf{g}'(i)$ for all entry i, it is suggested that the protocol is secure against malicious adversary.

However, as Song et al. [14] point out, such a construction is vulnerable to a type of selective abort attack. For example, suppose P_1 and P_3 are corrupted. The adversary can let P_1 sends $\mathbf{y}_1 + \boldsymbol{\delta}_i$ instead of $\mathbf{y}_1$, where $\boldsymbol{\delta}_i$ is a one-hot vector with its i-th entry being 1. After receiving $\mathbf{y}_2$ from P_2, P_3 first adjusts it by computing $\mathbf{y}_2 - \boldsymbol{\delta}_j$, then acts honestly. If P_2 happens to choose permutation π_2 with $\pi_2(j) = i$ (with non-negligible probability $1/m$), such action of P_2 corrects precisely the error introduced by P_1. Hence, if parties do not abort in the final check, this leaks information about π_2 to the adversary without detection.

At first glance, it seems quite easy to defend against such an attack: just insert a checking phase after each P_{i+1} receives $\mathbf{y}_i$ from P_i, and everything seems to work out. However, such a check turns out to be difficult. To protect the underlying secret, the message P_{i+1} receives from P_i is masked with uniform random mask, which prevents P_{i+1} from locally checking the correctness of the message. If P_{i+1} wants to check if the message is correct, i.e. if the product relation is maintained, it seems necessary that P_{i+1} shares the entire message back to all parties, and then all parties, after somehow jointly remove the random masks, carry out $O(m)$ MPC multiplications over all $O(m)$ entries, which is already $O(n^2 m)$ online communication and computation.

However, such communication and computation are not inevitable, if we are more selective about the implanted correlation in messages. Roughly, suppose the message sent from P_i to P_{i+1} is $\mathbf{y}_i$. Suppose we require P_i to send in addition some message $\mathbf{y}'_i$ that acts as MAC, which satisfies

$$\beta \mathbf{y}_i = \mathbf{y}'_i + \mathbf{r}'_i,$$

with $\langle \beta \rangle$ and $\langle \mathbf{r}'_i \rangle$ stored at $\mathcal{F}_{\text{MPC}}$. Due to the lack of element β and vector $\mathbf{r}'_i$, P_i will not be able to forge wrong message satisfying such relation.

Further, this check can be batched, in that the P_{i+1} can sample a challenge λ, and turn to check if

$$\langle \beta \rangle \underbrace{\sum_{j=1}^{m} \lambda^{j-1} \mathbf{y}_i(j)}_{=u} = \underbrace{\sum_{j=1}^{m} \lambda^{j-1} \mathbf{y}'_i(j)}_{=v} + \underbrace{\sum_{j=1}^{m} \lambda^{j-1} \langle \mathbf{r}'_i(j) \rangle}_{=w}.$$

Now by P_{i+1} broadcasting λ, u and v and all parties computing and opening the subtraction of above two terms, the check is done with $O(n)$ communication. By further moving the generation of λ (as now $\langle \lambda \rangle$) and w (as $\langle w \rangle$) to the offline phase, the online computation becomes also linear.

We note that, for a clarity of protocol description, the protocols presented in Section 6 achieve only $O(n^2m)$ communication. In Section 7 we formalize the above optimizations and achieve linear online complexities. Due to page limitation, we present the fully developed protocols in Section B, before presenting formal security proof.

5 Semi-honest Secure Shuffle

5.1 Functionality

In this section, we present our semi-honest shuffle protocol. Recall that we assume an ideal functionality $\mathcal{F}_{\mathrm{MPC}}$, which supports

$$[\![\pi(\mathbf{x})]\!] \leftarrow \Pi_{\mathrm{perm}}(P_i : \pi, [\![\mathbf{x}]\!]),$$

where $\mathbf{x}$ is an array of length m, and $\pi : [\![m]\!] \to [\![m]\!]$ is a permutation known only to P_i.

We give a two-phase shuffle protocol, consisting of an offline phase $\mathrm{Shuffle}_{\mathrm{off}}$ and an online phase $\mathrm{Shuffle}_{\mathrm{on}}$. The offline phase is in essence shuffling random numbers in order to generate a shuffle correlation. The online phase takes as input an array $[\![\mathbf{x}]\!]$ of length m, and consumes a fresh shuffle correlation. The online phase consists of mostly plaintext permutation carried out by each party locally, and is hence considerably fast.

5.2 Semi-honest Shuffle Correlation

The shuffle correlation for semi-honest multiparty computation is defined as follows.

Definition 4 (Semi-honest Shuffle Correlation). *The shuffle correlation for semi-honest setting is defined as*

$$\mathrm{cor} \coloneqq \{(\pi_1, ..., \pi_n), [\![\mathbf{r}]\!], [\![\mathbf{s}]\!], (\emptyset, \mathbf{z}_2, \mathbf{z}_3, ..., \mathbf{z}_n)\},$$

where

1. *π_i is an m-permutation (i.e. a permutation on m elements) known only to party P_i. π_i is sampled by P_i uniformly at random from all m-permutations.*
2. *$[\![\mathbf{r}]\!]$ and $[\![\mathbf{s}]\!]$ are two secret shared random vectors uniform over $\mathbb{F}^m$.*
3. *$\mathbf{z}_i$ is a random vector of length m and is known only to party P_i. They are uniformly random under the constraint*

$$\mathbf{s} = \pi_n(\pi_{n-1}(\cdots \pi_2(\pi_1(\mathbf{r}) - \mathbf{z}_2) - \mathbf{z}_3 \cdots) - \mathbf{z}_n).$$

The shuffle correlation is generated in the offline phase of the shuffle protocol and used to perform shuffle in the online phase. It is crucial that one shuffle correlation can be used only in one shuffle protocol session, same as one-time pad.

5.3 Offline Phase

The offline phase protocol $\text{Shuffle}_{\text{off}}$ takes as input the size m of the array and m-permutation π_i from party P_i. It outputs a shuffle correlation for later use in online phase. In later discussion, we show that m does not need to be the exact length of $\mathbf{x}$; by a slight modification to the protocol, an upper bound will be sufficient. For now, let's assume m is precise.

In offline phase, the parties first generate random vectors

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

each of length m. This is in essence generating $n \times m$ random field elements, with MPC primitive Π_{rand}.

Then the parties invoke functionality Π_{perm}, and obtain

$$[\![\pi_1(\mathbf{r}_1)]\!], [\![\pi_2(\mathbf{r}_2)]\!], ..., [\![\pi_n(\mathbf{r}_n)]\!],$$

where π_i is a permutation chosen by P_i. If P_i is honest, π_i is uniformly random and known only to P_i. This can be done by

$$[\![\pi_i(\mathbf{r}_i)]\!] \leftarrow \Pi_{\text{perm}}(P_i : \pi_i, [\![\mathbf{r}_i]\!]).$$

The parties then compute for $i = 2, 3, ..., n$,

$$[\![\mathbf{z}_i]\!] \leftarrow [\![\pi_{i-1}(\mathbf{r}_{i-1})]\!] - [\![\mathbf{r}_i]\!],$$

and open the value of $\mathbf{z}_i$ to party P_i.

This is the offline protocol $\text{Shuffle}_{\text{off}}$, which outputs a random vector of length m to each party $P_2, P_3, ..., P_n$. We denote by

$$\text{cor} \coloneqq \{(\pi_1, ..., \pi_n), [\![\mathbf{r}_1]\!], [\![\pi_n(\mathbf{r}_n)]\!], (\emptyset, \mathbf{z}_2, \mathbf{z}_3, ..., \mathbf{z}_n)\}$$

the semi-honest shuffle correlation. Note that the $[\![\mathbf{r}_1]\!]$ and $[\![\pi_n(\mathbf{r}_n)]\!]$ term are stored at $\mathcal{F}_{\text{MPC}}$, while the rest π_i, $\mathbf{z}_i$ is each held only by party P_i. To see that this is consistent with Definition 4, note that $[\![\mathbf{r}_1]\!]$ and $[\![\pi_n(\mathbf{r}_n)]\!]$ are exactly $[\![\mathbf{r}]\!]$ and $[\![\mathbf{s}]\!]$ in the Definition 4. To see that it also satisfies the conditional independency (i.e. "uniform under the constraint..."), it suffices to note that after fixing π_i, deciding all $\mathbf{r}_i$ will uniquely determine all $\mathbf{z}_i$, $\mathbf{r}$, $\mathbf{s}$. Hence, since all $\mathbf{r}_i$ is uniformly random, the resulting correlation will be uniform under the constraint given by definition.

This protocol is formally presented in Algorithm 1.

5.4 Online Phase

At the online phase of the protocol, the input $[\![\mathbf{x}]\!]$ arrives. The parties then compute $[\![\pi(x)]\!]$, where

$$\pi = \pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1.$$

Algorithm 1 cor $\leftarrow \text{Shuffle}_{\text{off}}(m, P_1 : \pi_1, ..., P_n : \pi_n)$

Require: For honest P_i, π_i is sampled uniformly from all m-permutations.
Ensure: Return a new shuffle correlation for online use.
  for $i = 1$ to n **do parallel**
    Generate random vector $[\![\mathbf{r}_i]\!]$ of length m.
    $[\![\pi_i(\mathbf{r}_i)]\!] \leftarrow \Pi_{\text{perm}}(P_i : \pi_i, [\![\mathbf{r}_i]\!])$.
  end for
  for $i = 2$ to n **do parallel**
    $[\![\mathbf{z}_i]\!] \leftarrow [\![\pi_{i-1}(\mathbf{r}_{i-1})]\!] - [\![\mathbf{r}_i]\!]$
    Open $[\![\mathbf{z}_i]\!]$ to P_i.
  end for
  Return $\{(\pi_1, ..., \pi_n), [\![\mathbf{r}_1]\!], [\![\pi_n(\mathbf{r}_n)]\!], (\emptyset, \mathbf{z}_2, \mathbf{z}_3, ..., \mathbf{z}_n)\}$.

Suppose the parties hold a fresh shuffle correlation

$$\text{cor} = \{(\pi_1, ..., \pi_n), [\![\mathbf{r}_1]\!], [\![\pi_n(\mathbf{r}_n)]\!], (\emptyset, \mathbf{z}_2, \mathbf{z}_3, ..., \mathbf{z}_n)\}$$

The parties first compute $[\![\mathbf{x} - \mathbf{r}_1]\!]$, and open it to party P_1. Denote by

$$\mathbf{z}_1 \coloneqq \mathbf{x} - \mathbf{r}_1,$$

which is known only to P_1.

Party P_1 computes and sends to P_2

$$\mathbf{y}_1 \coloneqq \pi_1(\mathbf{z}_1).$$

Then each party P_i with $i = 2, 3, ..., n-1$ sequentially locally computes and sends to P_{i+1}

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

The last party P_n receives $\mathbf{y}_{n-1}$ from P_{n-1}, computes

$$\mathbf{y}_n \coloneqq \pi_n(\mathbf{z}_n + \mathbf{y}_{n-1}),$$

and broadcasts it to all parties. Then all parties compute

$$[\![\pi(\mathbf{x})]\!] = \mathbf{y}_n + [\![\pi_n(\mathbf{r}_n)]\!],$$

where $[\![\pi_n(\mathbf{r}_n)]\!]$ comes from the correlation.

The above process is formally presented in Algorithm 2.

To see the correctness, note that each party P_i holds and sends

$$\begin{aligned}
&\mathbf{z}_1 \coloneqq \mathbf{x} - \mathbf{r}_1, && \mathbf{y}_1 \coloneqq \pi_1(\mathbf{z}_1) = \pi_1(\mathbf{x}) - \pi_1(\mathbf{r}_1),\\
&\mathbf{z}_2 \coloneqq \pi_1(\mathbf{r}_1) - \mathbf{r}_2, && \mathbf{y}_2 \coloneqq \pi_2(\mathbf{z}_2 + \mathbf{y}_1) = \overline{\pi}_2(\mathbf{x}) - \pi_2(\mathbf{r}_2),\\
&\mathbf{z}_3 \coloneqq \pi_2(\mathbf{r}_2) - \mathbf{r}_3, && \mathbf{y}_3 \coloneqq \pi_3(\mathbf{z}_3 + \mathbf{y}_2) = \overline{\pi}_3(\mathbf{x}) - \pi_3(\mathbf{r}_3),\\
&&&\vdots\\
&\mathbf{z}_n \coloneqq \pi_{n-1}(\mathbf{r}_{n-1}) - \mathbf{r}_n, && \mathbf{y}_n \coloneqq \pi_n(\mathbf{z}_n + \mathbf{y}_{n-1}) = \overline{\pi}_n(\mathbf{x}) - \pi_n(\mathbf{r}_n),
\end{aligned}$$

Algorithm 2 $[\![\pi(\mathbf{x})]\!] \leftarrow \text{Shuffle}_{\text{on}}([\![\mathbf{x}]\!])$

Require: An unused correlation $\{(\pi_1, ..., \pi_n), [\![\mathbf{r}_1]\!], [\![\pi_n(\mathbf{r}_n)]\!], (\emptyset, \mathbf{z}_2, \mathbf{z}_3, ..., \mathbf{z}_n)\}$.
Ensure: Return $[\![\pi(\mathbf{x})]\!]$, with $\pi = \pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1$.
$[\![\mathbf{z}_1]\!] \leftarrow [\![\mathbf{x}]\!] - [\![\mathbf{r}_1]\!]$
Open $\mathbf{z}_1$ to P_1.
P_1 computes locally $\mathbf{y}_1 \leftarrow \pi_1(\mathbf{z}_1)$.
P_1 sends $\mathbf{y}_1$ to P_2.
for $i = 2$ to $n - 1$ **do**
  P_i receives $\mathbf{y}_{i-1}$ from P_{i-1}.
  P_i computes locally $\mathbf{y}_i \leftarrow \pi_i(\mathbf{z}_i + \mathbf{y}_{i-1})$.
  P_i sends $\mathbf{y}_i$ to P_{i+1}.
end for
P_n receives $\mathbf{y}_{n-1}$ from P_{n-1}.
P_n computes locally $\mathbf{y}_n \leftarrow \pi_n(\mathbf{z}_n + \mathbf{y}_{n-1})$.
P_n broadcasts $\mathbf{y}_n$ to all parties.
$[\![\mathbf{x}']\!] \leftarrow \mathbf{y}_n + [\![\pi_n(\mathbf{r}_n)]\!]$
Return $[\![\mathbf{x}']\!]$.

where $\overline{\pi}_i$ is an abbreviation for $\pi_i \circ \pi_{i-1} \circ \cdots \circ \pi_1$. Hence, it is straightforward that the result will be

$$[\![\pi(\mathbf{x})]\!] = \mathbf{y}_n + [\![\pi_n(\mathbf{r}_n)]\!].$$

Due to page limit, the security proof is deferred to Section A in appendix.

6 Malicious Secure Shuffle

6.1 Functionality and Roadmap

In this section, we present our construction of shuffle protocol against malicious adversary and possibly dishonest majority, as long as $\mathcal{F}_{\text{MPC}}$ and Π_{perm} support so. For clarity of security proof, the construction proposed in this section has $O(n^2m)$ online communication and computation complexity. Later in Section 7 we show how to optimize both complexities to $O(nm)$.

Recall that we assume $\mathcal{F}_{\text{MPC}}$ supports command Π_{perm}, such that

$$(\langle\pi(\mathbf{x}_1)\rangle, ..., , \langle\pi(\mathbf{x}_t)\rangle) \leftarrow \Pi_{\text{perm}}(P_i : \pi, \langle\mathbf{x}_1\rangle, ..., , \langle\mathbf{x}_t\rangle).$$

Our construction consists of two phases, an offline phase $\text{Shuffle}_{\text{off}}$ and an online phase $\text{Shuffle}_{\text{on}}$, just like the semi-honest case. The offline phase will be in essence calling Π_{perm} to shuffle random values, and generating a shuffle correlation. The online phase takes vector $\langle x\rangle$ as input, consumes one fresh shuffle correlation, and outputs $\langle\pi(x)\rangle$ with π known to no one.

Recall that in semi-honest construction, each party P_i computes locally

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

and sends it to P_{i+1}. As discussed in Section 4, to prevent a type of selective abort attack, the protocol must guarantee the integrity of each $\mathbf{y}_i$ separately.

Such guarantee is achieved by appending an authentication code (MAC) to $\mathbf{y}_i$, which (implicitly) forms certain correlation with original message. Denote by $\mathbf{y}_i^1$ the original message to be authenticated, and by $\mathbf{y}_i^2$ the corresponding MAC, which is also a vector of length m. The crux is, only if P_i applies honestly the operations specified by the protocol, will it obtain a well-formed message pair $\mathbf{y}_i^1$ and $\mathbf{y}_i^2$ with the correct correlation between them, which will pass the subsequent correlation check. This correlation is hidden, in that anyone ignorant to certain trapdoors would find the message uniformly random. As these trapdoors are stored at $\mathcal{F}_{\text{MPC}}$, the adversary remains unaware of them, which prevents the forging of valid messages that contain the correct correlation. Hence, the honest behavior of P_i is enforced, as the check will fail and the protocol will abort with overwhelming probability if P_i acts maliciously.

6.2 Malicious Shuffle Correlation

The shuffle correlation for malicious multiparty computation is defined as follows.

Definition 5 (Malicious Shuffle Correlation). *The shuffle correlation for malicious setting is defined as*

$$\text{cor} = \left\{ \begin{array}{ccccccccc} \langle\beta\rangle & \langle\mathbf{r}\rangle & \langle\beta\mathbf{r}\rangle & \langle\pi_1^{-1}(\mathbf{r}'_1)\rangle & \langle\mathbf{s}\rangle & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\ \langle\mathbf{r}'_1\rangle & \langle\mathbf{r}'_2\rangle & \langle\mathbf{r}'_3\rangle & \langle\mathbf{r}'_4\rangle & \langle\mathbf{r}'_5\rangle & \cdots & \langle\mathbf{r}'_n\rangle \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n \end{array} \right\},$$

where

1. *π_i is an m-permutation known only to P_i. It is sampled uniformly from all possible m-permutations by P_i (if P_i is honest).*
2. *$\langle\beta\rangle$ is a secret shared random variable. It plays the role of the authentication key, i.e. MAC key.*
3. *$\langle\mathbf{r}\rangle, \langle\mathbf{s}\rangle, \langle\mathbf{r}'_1\rangle, ..., \langle\mathbf{r}'_n\rangle$ are secret shared vectors of length m, with each entry independently uniformly random. $\langle\beta\mathbf{r}\rangle$ is secret shared $\beta \cdot \mathbf{r}$.*
4. *$\mathbf{z}_i = (\mathbf{z}_i^1, \mathbf{z}_i^2)$, where each $\mathbf{z}_i^j$ is a vector of length m. The entries of $\mathbf{z}_i^1$ are uniformly random, under the constraint*

$$\begin{cases} \mathbf{z}_i^2 = \beta\mathbf{z}_i^1 + \mathbf{r}'_{i-1} - \pi_i^{-1}(\mathbf{r}'_i) & \forall i = 2, 3, ..., n \\ \mathbf{s} = \pi_n(\pi_{n-1}(\cdots\pi_2(\pi_1(\mathbf{r}) - \mathbf{z}_2^1) - \mathbf{z}_3^1\cdots) - \mathbf{z}_n^1) \end{cases}$$

This shuffle correlation will be generated in the offline phase of the shuffle protocol and used in the online phase, as we demonstrate in following subsections. Looking ahead, β plays a role that is similar to the MAC key in SPDZ framework [22] (which is usually denoted as "α" in context), and helps check if the result after each permutation is correct. Nevertheless, it should be noted that our definition of shuffle correlation is not hence limited to SPDZ framework.

6.3 Correlation Check of Public Values

The functionality of protocol Verify$(a, b, \langle\beta\rangle, \langle r\rangle)$ is to check publicly whether $\beta a = b + r$, where both a and b are publicly known. This is useful because, looking ahead, we will implant a hidden correlation between the message $\mathbf{y}_i^1$ and $\mathbf{y}_i^2$ sent by P_i. This correlation will assist the parties in checking whether P_i has acted honestly. Specifically, the correlation is exactly $\mathbf{y}_i^2 = \beta \mathbf{y}_i^1 + \mathbf{r}_i'$, with $\langle\beta\rangle$ and $\langle\mathbf{r}_i'\rangle$ hidden behind $\mathcal{F}_{\text{MPC}}$. Therefore, this protocol can be used to verify whether all parties have followed the protocol honestly up to this point.

Such a check can be done easily. The parties simply compute and open

$$a\langle\beta\rangle - b - \langle r\rangle,$$

and check if it equals zero. As many MPC frameworks are based on linear secret sharing scheme, it is likely that this computation can be done locally, as it involves only addition and multiplication with public constant. Hence, the overhead of this protocol equals an opening operation in MPC.

This protocol is formally presented in Algorithm 3.

Algorithm 3 Verify$(a, b, \langle\beta\rangle, \langle r\rangle)$

Require: By protocol design, $\beta a = b + r$.
Ensure: Abort if $\beta a \neq b + r$.
  Compute $d \leftarrow a\langle\beta\rangle - b - \langle r\rangle$.
  Open d for all parties.
  Abort if $d \neq 0$.

To see the security of this protocol, note that every operation carried out in the protocol is via $\mathcal{F}_{\text{MPC}}$, i.e. some trivial arithmetic operations plus a public opening operation. Hence, the protocol is secure by definition.

6.4 Offline Phase

The offline phase protocol $\text{Shuffle}_{\text{off}}$ takes input the length m of the vector $\langle\mathbf{x}\rangle$ and π_i from P_i. As mentioned in Section 5, let's assume for now m is the exact length of later input vector $\langle\mathbf{x}\rangle$.

In offline protocol $\text{Shuffle}_{\text{off}}$, the parties first generate a random field element $\langle\beta\rangle$, which is referred to as the shuffle authentication key. Then the parties generate $2n$ random vectors, denoted as

$$\langle\mathbf{r}_1\rangle, \langle\mathbf{r}_2\rangle, ..., \langle\mathbf{r}_n\rangle, \langle\mathbf{r}_1'\rangle, \langle\mathbf{r}_2'\rangle, ..., \langle\mathbf{r}_n'\rangle.$$

The parties then call protocol Π_{mul} and acquire

$$\langle\beta\mathbf{r}_1\rangle, \langle\beta\mathbf{r}_2\rangle, ..., \langle\beta\mathbf{r}_n\rangle,$$

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

The parties then invoke Π_{perm} n times, and acquire for $i \in [n]$

$$\left(\langle \pi_i(\mathbf{r}_i)\rangle, \langle \pi_i(\beta\mathbf{r}_i)\rangle, \langle \pi_i(\mathbf{r}'_i)\rangle\right) \leftarrow \Pi_{\text{perm}}(P_i : \pi_i, \langle \mathbf{r}_i\rangle, \langle \beta\mathbf{r}_i\rangle, \langle \mathbf{r}'_i\rangle).$$

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

$$\begin{aligned}
\langle \mathbf{z}_i^1\rangle &\leftarrow \langle \pi_{i-1}(\mathbf{r}_{i-1})\rangle - \langle \mathbf{r}_i\rangle,\\
\langle \mathbf{z}_i^2\rangle &\leftarrow \langle \pi_{i-1}(\beta\mathbf{r}_{i-1})\rangle + \langle \pi_{i-1}(\mathbf{r}'_{i-1})\rangle - \langle \beta\mathbf{r}_i\rangle - \langle \mathbf{r}'_i\rangle,
\end{aligned}$$

where the superscript is simply an index, not "exponentiation". For notation convenience, denote by

$$\langle \mathbf{z}_i\rangle = (\langle \mathbf{z}_i^1\rangle, \langle \mathbf{z}_i^2\rangle).$$

Note that $\mathbf{z}_i$ is a vector of length $2m$.

The parties then open each $\langle \mathbf{z}_i\rangle$ to P_i, for $i \geq 2$. The shuffle correlation returned by this protocol is

$$\text{cor} = \left\{ \begin{array}{ccccccc}
\langle \beta\rangle & \langle \mathbf{r}_1\rangle & \langle \beta\mathbf{r}_1\rangle & \langle \mathbf{r}'_1\rangle & \langle \pi_n(\mathbf{r}_n)\rangle & & \\
\pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\
\langle \pi_1(\mathbf{r}'_1)\rangle & \langle \pi_2(\mathbf{r}'_2)\rangle & \langle \pi_3(\mathbf{r}'_3)\rangle & \langle \pi_4(\mathbf{r}'_4)\rangle & \langle \pi_5(\mathbf{r}'_5)\rangle & \cdots & \langle \pi_n(\mathbf{r}'_n)\rangle \\
 & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n
\end{array} \right\}.$$

Note that each $\mathbf{z}_i$ is held as plaintext by party P_i for $i \geq 2$, and variables with bracket is stored at $\mathcal{F}_{\text{MPC}}$, hidden from parties. To see that this is consistent with Definition 5, note that $\mathbf{r}_1, \beta\mathbf{r}_1, \mathbf{r}'_1, \pi_n(\mathbf{r}_n)$ and $\pi_i(\mathbf{r}'_i)$ are exactly $\mathbf{r}, \beta\mathbf{r}, \pi_1^{-1}(\mathbf{r}'_1), \mathbf{s}$ and $\mathbf{r}'_i$ in former definition. Also, it is straightforward to verify that these variables are uniformly independently random under the constraint given in Definition 5, as determining all $\beta, \pi_i, \mathbf{r}_i$ and $\mathbf{r}'_i$ will uniquely determine the entire shuffle correlation, i.e. there is a bijection between the correlations in definition and the ones generated by the protocol.

This protocol is formally described in Algorithm 4.

6.5 Online Shuffle

The online shuffle protocol $\text{Shuffle}_{\text{on}}$ takes as input a length m secret shared vector $\langle \mathbf{x}\rangle$. It consumes a fresh shuffle correlation cor, and outputs

$$\langle \overline{\pi}_n(\mathbf{x})\rangle = \langle \pi_n \circ \cdots \circ \pi_1(\mathbf{x})\rangle,$$

where π_i is known to P_i and is stored in the shuffle correlation. To pass the later linear test, it is crucial that the performed permutation π_i is exactly the one used to generate correlation.

The parties first compute

$$\langle \beta\mathbf{x}\rangle \leftarrow \Pi_{\text{mul}}(\langle \beta\rangle, \langle \mathbf{x}\rangle),$$

i.e. multiplying every entry of $\mathbf{x}$ by β. Then they compute

$$\begin{aligned}
\langle \mathbf{z}_1^1\rangle &\leftarrow \langle \mathbf{x}\rangle - \langle \mathbf{r}_1\rangle,\\
\langle \mathbf{z}_1^2\rangle &\leftarrow \langle \beta\mathbf{x}\rangle - \langle \beta\mathbf{r}_1\rangle - \langle \mathbf{r}'_1\rangle.
\end{aligned}$$

Algorithm 4 cor $\leftarrow \text{Shuffle}_{\text{off}}(m, P_1 : \pi_1, ..., P_n : \pi_n)$

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

Generate random value $\langle\beta\rangle$.
for $i = 1$ to n **do parallel**
  Generate random vectors $\langle\mathbf{r}_i\rangle$ and $\langle\mathbf{r}'_i\rangle$.
  $\langle\beta\mathbf{r}_i\rangle \leftarrow \Pi_{\text{mul}}(\langle\mathbf{r}_i\rangle, \langle\beta\rangle)$
  $\left(\langle\pi_i(\mathbf{r}_i)\rangle, \langle\pi_i(\beta\mathbf{r}_i)\rangle, \langle\pi_i(\mathbf{r}'_i)\rangle\right) \leftarrow \Pi_{\text{perm}}\left(P_i : \pi_i, \langle\mathbf{r}_i\rangle, \langle\beta\mathbf{r}_i\rangle, \langle\mathbf{r}'_i\rangle\right)$
end for
for $i = 2$ to n **do parallel**
  $\langle\mathbf{z}_i^1\rangle \leftarrow \langle\pi_{i-1}(\mathbf{r}_{i-1})\rangle - \langle\mathbf{r}_i\rangle$
  $\langle\mathbf{z}_i^2\rangle \leftarrow \langle\pi_{i-1}(\beta\mathbf{r}_{i-1})\rangle - \langle\beta\mathbf{r}_i\rangle + \langle\pi_{i-1}(\mathbf{r}'_{i-1})\rangle - \langle\mathbf{r}'_i\rangle$
  $\langle\mathbf{z}_i\rangle \coloneqq (\langle\mathbf{z}_i^1\rangle, \langle\mathbf{z}_i^2\rangle)$
  Open $\mathbf{z}_i$ to P_i.
end for

$$\text{Return cor} = \left\{ \begin{array}{ccccccc} \langle\beta\rangle & \langle\mathbf{r}_1\rangle & \langle\beta\mathbf{r}_1\rangle & \langle\mathbf{r}'_1\rangle & \langle\pi_n(\mathbf{r}_n)\rangle & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\ \langle\pi_1(\mathbf{r}'_1)\rangle & \langle\pi_2(\mathbf{r}'_2)\rangle & \langle\pi_3(\mathbf{r}'_3)\rangle & \langle\pi_4(\mathbf{r}'_4)\rangle & \langle\pi_5(\mathbf{r}'_5)\rangle & \cdots & \langle\pi_n(\mathbf{r}'_n)\rangle \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n \end{array} \right\}$$

Note that except $\langle\mathbf{x}\rangle$ and $\langle\beta\mathbf{x}\rangle$, all terms on the right-hand side above come from correlation. The parties then open these two value to P_1 as

$$\mathbf{z}_1 = (\mathbf{z}_1^1, \mathbf{z}_1^2).$$

Then, P_1 computes and broadcasts

$$\mathbf{y}_1 \leftarrow \pi_1(\mathbf{z}_1).$$

Note that $\mathbf{z}_1$ is a vector of length $2m$, and here we slightly abuse the notation, and define

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

We define thus also for all vectors of length $2m$. This gives a simple representation of operation which is consistent with semi-honest case, albeit what's happening under the surface is essentially different.

Then, after receiving $\mathbf{y}_{i-1}$ from P_{i-1}, party P_i computes and broadcasts

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

The parties eventually receive

$$\mathbf{y}_n = (\mathbf{y}_n^1, \mathbf{y}_n^2).$$

The parties then invoke Π_{input} to acquire $\langle\mathbf{y}_n^1\rangle$, and compute

$$\langle\mathbf{x}'\rangle \leftarrow \langle\mathbf{y}_n^1\rangle + \langle\pi_n(\mathbf{r}_n)\rangle,$$

which would equal $\overline{\pi}_n(\mathbf{x})$ in plaintext, if all parties have acted honestly.

To enforce the honest behavior of each party, however, additional check regarding $\mathbf{y}_i$ must be done. Note that by design,

$$\beta \mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}'_i)$$

should hold for every $i \in [n]$, where $\langle r'_i \rangle$ is never opened. Hence, the parties call protocol

$$\text{Verify}(\mathbf{y}_i^1, \mathbf{y}_i^2, \langle \beta \rangle, \langle \pi_i(\mathbf{r}'_i) \rangle)$$

for all $i \in [n]$, and abort if any of them fail. Note that we here slightly abuse the notation by passing vectors as parameter into the protocol, as an abbreviation for calling in parallel

$$\text{Verify}(\mathbf{y}_i^1(j), \mathbf{y}_i^2(j), \langle \beta \rangle, \langle \mathbf{r}'_i(\pi(j)) \rangle),$$

for each $j \in [m]$.

The protocol is formally presented in Algorithm 5.

Algorithm 5 $\langle \overline{\pi}_n(\mathbf{x}) \rangle \leftarrow \text{Shuffle}_{\text{on}}(\langle \mathbf{x} \rangle)$

Require: Fresh cor $= \left\{ \begin{array}{ccccccc} \langle \beta \rangle & \langle \mathbf{r}_1 \rangle & \langle \beta \mathbf{r}_1 \rangle & \langle \mathbf{r}'_1 \rangle & \langle \pi_n(\mathbf{r}_n) \rangle & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\ \langle \pi_1(\mathbf{r}'_1) \rangle & \langle \pi_2(\mathbf{r}'_2) \rangle & \langle \pi_3(\mathbf{r}'_3) \rangle & \langle \pi_4(\mathbf{r}'_4) \rangle & \langle \pi_5(\mathbf{r}'_5) \rangle & \cdots & \langle \pi_n(\mathbf{r}'_n) \rangle \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n \end{array} \right\}$.

Ensure: Return $\langle \overline{\pi}_n(\mathbf{x}) \rangle$ if the protocol does not abort.

$\langle \beta \mathbf{x} \rangle \leftarrow \Pi_{\text{mul}}(\langle \beta \rangle, \langle \mathbf{x} \rangle)$
$\langle \mathbf{z}_1^1 \rangle \leftarrow \langle \mathbf{x} \rangle - \langle \mathbf{r}_1 \rangle$
$\langle \mathbf{z}_1^2 \rangle \leftarrow \langle \beta \mathbf{x} \rangle - \langle \beta \mathbf{r}_1 \rangle - \langle \mathbf{r}'_1 \rangle$
Open $\mathbf{z}_1 \coloneqq (\mathbf{z}_1^1, \mathbf{z}_1^2)$ to P_1.
P_1 computes and broadcasts $\mathbf{y}_1 \leftarrow \pi_1(\mathbf{z}_1) = (\pi_1(\mathbf{z}_1^1), \pi_1(\mathbf{z}_1^2))$.
for $i = 2$ to n **do**
    P_i computes and broadcasts $\mathbf{y}_i \leftarrow \pi_i(\mathbf{z}_i + \mathbf{y}_{i-1})$.
end for
for $i = 1$ to n **do parallel**
    $\text{Verify}(\mathbf{y}_i^1, \mathbf{y}_i^2, \langle \beta \rangle, \langle \pi_i(\mathbf{r}'_i) \rangle)$
end for
$\langle \mathbf{y}_n^1 \rangle \leftarrow \Pi_{\text{input}}(\mathbf{y}_n^1)$.
$\langle \mathbf{x}' \rangle \leftarrow \langle \mathbf{y}_n^1 \rangle + \langle \pi_n(\mathbf{r}_n) \rangle$
Return $\langle \mathbf{x}' \rangle$.

To see the correctness of above process when all parties are honest, note that each P_i holds $\mathbf{z}_i$, where

$$\begin{aligned}
\mathbf{z}_1 &= \quad \mathbf{x} - \mathbf{r}_1, && \beta\mathbf{x} - \beta\mathbf{r}_1 - \mathbf{r}'_1, \\
\mathbf{z}_2 &= \quad \pi_1(\mathbf{r}_1) - \mathbf{r}_2, && \pi_1(\beta\mathbf{r}_1 + \mathbf{r}'_1) - \beta\mathbf{r}_2 - \mathbf{r}'_2, \\
\mathbf{z}_3 &= \quad \pi_2(\mathbf{r}_2) - \mathbf{r}_3, && \pi_2(\beta\mathbf{r}_2 + \mathbf{r}'_2) - \beta\mathbf{r}_3 - \mathbf{r}'_3, \\
&\ \ \vdots \\
\mathbf{z}_n &= \pi_{n-1}(\mathbf{r}_{n-1}) - \mathbf{r}_n, && \pi_{n-1}(\beta\mathbf{r}_{n-1} + \mathbf{r}'_{n-1}) - \beta\mathbf{r}_n - \mathbf{r}'_n.
\end{aligned}$$

And the $\mathbf{y}_i$ broadcast by each party P_i is

$$\begin{aligned}
\mathbf{y}_1 &= \pi_1(\mathbf{x}) - \pi_1(\mathbf{r}_1),\ \pi_1(\beta\mathbf{x}) - \pi_1(\beta\mathbf{r}_1 + \mathbf{r}'_1),\\
\mathbf{y}_2 &= \overline{\pi}_2(\mathbf{x}) - \pi_2(\mathbf{r}_2),\ \overline{\pi}_2(\beta\mathbf{x}) - \pi_2(\beta\mathbf{r}_2 + \mathbf{r}'_2),\\
\mathbf{y}_3 &= \overline{\pi}_3(\mathbf{x}) - \pi_3(\mathbf{r}_3),\ \overline{\pi}_3(\beta\mathbf{x}) - \pi_3(\beta\mathbf{r}_3 + \mathbf{r}'_3),\\
&\vdots\\
\mathbf{y}_n &= \overline{\pi}_n(\mathbf{x}) - \pi_n(\mathbf{r}_n), \overline{\pi}_n(\beta\mathbf{x}) - \pi_n(\beta\mathbf{r}_n + \mathbf{r}'_n).
\end{aligned}$$

Hence, if the parties input the first m entries of $\mathbf{y}_n$ as $\langle \mathbf{y}_n^1 \rangle$ and add it by $\langle \pi_n(\mathbf{r}_n) \rangle$, the result is indeed $\langle \overline{\pi}_n(\mathbf{x}) \rangle$. Also, if all parties act honestly, the calls to Verify$(\mathbf{y}_i^1, \mathbf{y}_i^2, \langle \beta \rangle, \langle \pi_i(\mathbf{r}'_i) \rangle)$ should all pass.

Due to the page limit, the security proof is deferred to Section B in appendix.

7 Achieve Linear Online Phase

7.1 Linear Online Communication

The online communication complexity for malicious secure shuffle has so far been $O(n^2 m)$, where n is the number of parties and m is the dimension of vector. This is due to the broadcast of $\mathbf{y}_i$, which is a vector of length $2m$.

However, we make the following simple observations:

1. If $\mathbf{y}_{i-1}$ is correct, then the permutation chosen by P_i is protected.
2. If $\mathbf{y}_n$ is correct, then the result is correct.
3. The verifications of the correlation can be batched.

Hence, P_i needs not care if the preceding $\mathbf{y}_j$ (except $\mathbf{y}_{i-1}$) is correct or not. For example, if $P_1, ..., P_{i-1}$ are corrupted, $\mathbf{y}_1, ..., \mathbf{y}_{i-2}$ can be arbitrary values. But as long as $\mathbf{y}_{i-1}$ is correct, this is somewhat the same as honest behavior, as other parties do not care about preceding values at all. This gives the following idea, that instead of broadcast, each party P_i sends $\mathbf{y}_i$ only to P_{i+1}, who batch-checks if $\mathbf{y}_i$ is correct, i.e. if it satisfies $\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}'_i)$. This is formally present in Algorithm 6.

Algorithm 6 PartialVerify$(P_{i+1}, \mathbf{y}_i^1, \mathbf{y}_i^2, \langle \beta \rangle, \langle \pi_i(\mathbf{r}'_i) \rangle)$

Require: By design $\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}'_i)$, with $\mathbf{y}_i^1, \mathbf{y}_i^2$ sent from P_i to P_{i+1}.
Ensure: If P_{i+1} does not know correct $\mathbf{y}_i^b$, abort w.h.p.
P_{i+1} locally samples $\lambda \leftarrow \mathbb{F}$.
P_{i+1} computes $w_b \coloneqq \sum_{j=1}^{m} \lambda^{j-1} \mathbf{y}_i^b(j)$, $b \in \{1, 2\}$.
P_{i+1} broadcasts λ, w_1 and w_2.
$\langle c \rangle \leftarrow w_1 \cdot \langle \beta \rangle - w_2 - \sum_{j=1}^{m} \lambda^{j-1} \cdot \langle \pi_i(\mathbf{r}'_i)(j) \rangle$
Open c to all parties.
Abort if $c \neq 0$.

We remark that the protocol is in essence a test of knowledge, in that it passes if and only if P_{i+1} (malicious or not) knows the correct $\mathbf{y}_i$. In particular,

if P_i and P_{i+1} are both corrupted, P_i can send garbage to P_{i+1} as $\mathbf{y}_i$, while P_{i+1} runs the protocol as if it's receiving another version of messages. However, the test will pass only if P_{i+1} broadcasts λ', $\mathbf{w}'_1$ and $\mathbf{w}'_2$ that are identical to the ones generated with correct $\lambda = \lambda'$ and $\mathbf{y}_i$. As long as malicious P_{i+1} proves that it knows what the messages ought to be, the protocol can proceed normally.

By plugging in such check for each P_i ($i \geq 2$), and finally P_n broadcasting $\mathbf{y}_n$ and all parties checking the correctness of $\mathbf{y}_n$ together, the shuffle protocol is implemented with $O(nm)$ communication and $O(n^2m)$ computation.

7.2 Linear Computation

After the optimization of the previous section, the protocol now has $O(nm)$ online communication. However, it has still $O(n^2m)$ online computation, due to computing the term $\sum_{j=1}^{m} \lambda^{j-1} \cdot \pi_i(\mathbf{r}'_i(j))$, which involves all parties scanning through the entire vector $\mathbf{r}'_i$, resulting in $O(nm)$ computation per check. Since this term is independent of $\mathbf{x}$, it can be easily removed to the offline phase.

In the offline phase, the parties can call Π_{rand} and generate n random values

$$\langle c_1 \rangle, \langle c_2 \rangle, ..., \langle c_n \rangle.$$

Then for each $\langle c_i \rangle$, the parties can raise it to j-th power, i.e. $\langle c_i^j \rangle$, for all $j \in [m]$. This can be done in $\log m$ rounds, with nm multiplication in total. We note that there are also techniques to optimize this to $O(1)$ rounds with help of other MPC primitives [33], albeit in practice this tends to be more inefficient due to large constant. The parties are now able to compute the term

$$\langle d_i \rangle \coloneqq \sum_{j=1}^{m} \langle c_i^{j-1} \rangle \cdot \langle \mathbf{r}'_i(j) \rangle,$$

in the offline phase, with again another nm multiplications.

Then in the online phase, when party P_i receives $\mathbf{y}_{i-1}$, all parties open $\langle c_i \rangle$ to P_i. P_i then computes locally and broadcasts

$$w_1 \coloneqq \sum_{j=1}^{m} c_i^{j-1} \mathbf{y}_i^1(j),$$

$$w_2 \coloneqq \sum_{j=1}^{m} c_i^{j-1} \mathbf{y}_i^2(j),$$

and all parties move to check if

$$\beta w_1 \stackrel{?}{=} w_2 + \langle d_i \rangle,$$

which is trivial.

Due to page limitation, the full protocols undergone above optimizations are presented in appendix, in Algorithm 7, 8 and 9.

We remark that, this approach achieves linear online computation with n more calls to Π_{rand} and $2nm$ more multiplication in offline phase. Although this does not increase asymptotic offline complexity, since the $O(n^2m)$ online computation (before this optimization) consists mostly of local field operations, it is possible that the version with $O(n^2m)$ online computation will be a better trade off between offline and online overheads.

7.3 Complexity Analysis

Assume we enhance the protocol with above two optimizations. Below we briefly analyze the complexity of our malicious secure shuffle protocol.

The offline communication and computation complexity of our protocol is $O(nP+nmR+nmM)$, where P is the overall complexity for Π_{perm}, R is the overall complexity for Π_{rand} and M is the overall complexity for Π_{mul}. If $R = M = O(n)$ (e.g. [19][20]), since in current constructions $P = \Omega(nm \log m)$ (e.g. [14][15]), the offline complexities are $O(nP)$. This indicates that the offline complexities are asymptotically the same as invoking Π_{perm} for n times. Since these invocations to Π_{perm} are inevitable in permute-in-turn paradigm, our construction obtains linear online phase almost for free.

The online communication complexity of our protocol is $O(n(n+m))$. By passing around $\mathbf{y}_i$ and finally broadcasting $\mathbf{y}_n$, the communication of all parties is $O(nm)$, since each vector is of length $2m$. The verification requires that parties open c_i to P_i, and that P_i computes and broadcasts two elements w_1 and w_2, which is $O(n)$ communication per party. Also, each check requires opening one shared element, which is in total $O(n^2)$ communication. Summing up, this is $O(n(n+m))$. Since in practice, the number of items to be shuffled is usually much larger than the number of participants, this is $O(nm)$ in most cases.

The online computation complexity of our protocol is $O(nm)$, counted by field operation. After receiving $\mathbf{y}_{i-1}$, party P_i needs to compute w_1, w_2 and $\mathbf{y}_i$, each of which requires $O(m)$ computation. This is hence $O(nm)$ for all parties.

8 Experiments

8.1 Experiments Setting

We implement our shuffle protocol with the shuffle protocol of Song et al. [14] and MP-SPDZ [34] MPC framework. MP-SPDZ framework supports Mascot protocol [30], an MPC arithmetic protocol based on additive secret sharing. The Mascot supports the multiplication operation our protocol require, and is secure against malicious adversary and dishonest majority. We first implement the permutation protocol and the shuffle protocol of Song et al. [14] (for convenient, we denote it as "basic shuffle protocol" in the following), then implement our shuffle protocol based on it, compare the performance of the two protocols. The target of our experiment is to see if our construction could indeed significantly improve the online communication/computation complexity of the basic shuffle protocol as suggested by theoretical analysis.

We note that the choice of Mascot as the specific MPC framework is rather casual, as our constructions are also compatible to other frameworks. The important thing is that we implement the shuffle protocol of Song et al. [14] also for Mascot, so we are comparing the shuffle protocols under same MPC framework.

Our experiment is run on a host equipped with 32 processors, each being Intel(R) Xeon(R) Gold 5122 CPU @ 3.60GHz. The operating system is Ubuntu 18.04.6 LTS. Each party is simulated by a process on the host, and the communication in between is via loopback. We use the Linux tc command to simulate WAN network, with a bandwidth of 80 MB/s and RTT 60 ms.

We set the security parameter as $\kappa = 40$ and choose a prime field with size around 2^{64} for Mascot protocol. The basic shuffle protocol of [14] contains a parameter to trade-off between communication overhead and computation overhead, due to the application of sub-permutation decomposition technique of [17]. This parameter also affects offline/online running time. Hence, we report the results of two versions of basic protocols, one with the least online running time and one with the least overall time. As the online phase of our protocol is unaffected by basic permutation protocol, we report the result of our protocol with the least overall running time.

Our code is available at https://github.com/GJCPP/MP-SPDZ-Shuffle.

8.2 Experiments with Number of Parties

We first test the protocols running by $n = 3, 6, 9, 12, 15$ parties. This test aims to verify that after the enhancement suggested in Section 7, our protocol has linear online communication, while the basic shuffle protocol has $O(Bn^2m)$ online communication, where m is the number of items. The results of the experiment is listed in Table 2.

From these tables, it can be seen that the growth of both online communication and running time of our protocol appear to be much slower than that of basic protocol. One of the significant phenomenons is that the online communication overhead of our protocol is almost invariant for each party, which coincides well with the theoretic result that the online communication should be $O(n(n+m))$. Also, it can be seen that the trade-off of basic protocol has its limit, in the sense that even if we choose a parameter that completely ignores the offline cost, its online running time is still slower than our protocol. This is because the basic shuffle protocol has its complexity inherently $\Theta(Bn^2m)$, which cannot be overcome by increasing offline computation overheads.

It should also be noted that, when optimizing the basic protocol by its overall running time, its offline communication and running time are both much smaller than ours, by approximately a fraction of 0.33 and 0.65, respectively. However, the online communication and running time of our protocol are better than that of the basic protocol by approximately two orders of magnitude. This coincides with the theoretical analysis that the online complexity of basic protocol should grow quadratically with the number of parties, while our protocol only linearly.

Table 2. Offline/Online Communication and Running Time

		Communication per Party (MB)					Running Time (s)				
	n	3	6	9	12	15	3	6	9	12	15
Offline	$[14]^1$	184	493	865	1260	1679	64.3	282	663	1214	2008
	$[14]^2$	135	355	629	899	1278	102	401	936	1621	2735
	ours	412	1294	2327	3552	4896	103	444	1085	1987	2875
Online	$[14]^1$	14.0	35.0	56.1	76.4	92.6	8.65	38.6	76.8	128	187
	$[14]^2$	7.73	18.3	28.3	36.0	45.9	6.67	25.3	47.8	66.0	95.4
	ours	0.262	0.329	0.353	0.366	0.376	0.625	1.27	1.94	3.41	4.32

n is the number of parties. The number of items is $m = 2^{12}$.
$[14]^1$ is optimized to minimize total running time.
$[14]^2$ is optimized to minimize online running time.

Table 3. Offline/Online Communication and Running Time

		Communication per Party (MB)					Running Time (s)				
	$\log_2 m$	10	12	14	16	18	10	12	14	16	18
Offline	$[14]^1$	22.4	89.0	449	1770	8473	7.84	30.0	135	553	2586
	$[14]^2$	23.5	64.8	275	1069	5012	140	91.2	314	2029	3481
	ours	56.5	230	908	2759	11950	15.7	51.7	213	771	3495
Online	$[14]^1$	1.88	6.94	35.6	135	666	1.15	1.88	8.66	38.3	159
	$[14]^2$	0.999	3.40	14.1	49.2	285	0.972	0.972	1.75	10.8	35.2
	ours	0.049	0.196	0.786	3.14	12.5	0.336	0.350	0.398	0.502	1.23

m is the number of items. The number of parties is $n = 2$.
$[14]^1$ is optimized to minimize total running time.
$[14]^2$ is optimized to minimize online running time.

8.3 Experiments with Number of Items

We also test the protocols for shuffling $m = 2^{10}, 2^{12}, ..., 2^{18}$ items between $n = 2$ parties. These tests aim at testing the scalability of the protocol with respect to the growing of data. The results are listed in Table 3.

From the table we can see that, both online communication and running time are far less than those of basic protocols. Also, it can be observed that the communication and running time of all the three protocols scale approximately by a factor of four, which is consistent with the theoretical result that all the protocols are linear with respect to the number of items. Compared to the basic protocol optimized to minimize total running time, our protocol is approximately 1.5× slower in the offline phase, yet is around 100× faster in the online phase when the number of items is large. This advantage is due to the constant B introduced by cut-and-choose technique in the basic protocol. The gap becomes more significant with the growing of number of items, possibly due to that when the number of items is small, the computation (in contrast to communication) is the only bottleneck.

9 Conclusion

In this paper, we study how to design an MPC shuffle protocol with least online overheads. We define shuffle correlation for both semi-honest and malicious MPC. We show how to generate them and use them to obtain a shuffle protocol with linear online communication and computation, with black-box access to MPC permutation protocol and basic MPC arithmetic operations. Our definitions are thus generic and can be instantiated with various MPC frameworks. Remarkably, by instantiating our construction with the MPC permutation protocol by Song et al. [14], we obtain the first malicious secure MPC shuffle protocol with linear online phase for additive secret sharing scheme. Instantiating with Shamir secret sharing and the MPC permutation protocol by Keller and Scholl [15], we obtain the first MPC shuffle protocol with linear online phase for Shamir secret sharing scheme, for both semi-honest and malicious security. The security proofs for both constructions are presented, which show that our malicious secure construction could achieve UC security in $\mathcal{F}_{\text{MPC}}$-hybrid model. The experiments show that, compared to the basic shuffle protocol that is used to instantiate our construction, our protocol performs notably better in both online communication and running time.

Bibliography

[1] Sherman S.M. Chow, Jie Han Lee, and Lakshminarayanan Subramanian. Two-party computation model for privacy-preserving queries over distributed databases. 2009. Publisher Copyright: © 2009 Proceedings of the Symposium on Network and Distributed System Security, NDSS 2009.

[2] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. Smcql: Secure querying for federated databases, 2017.

[3] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. Conclave: secure multi-party computation on big data. EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362818. https://doi.org/10.1145/3302424.3303982.

[4] Renuga Kanagavelu, Zengxiang Li, Juniarto Samsudin, Yechao Yang, Feng Yang, Rick Siow Mong Goh, Mervyn Cheah, Praewpiraya Wiwatphonthana, Khajonpong Akkarajitsakul, and Shangguang Wang. Two-phase multi-party computation enabled privacy-preserving federated learning. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 410–419, 2020. https://doi.org/10.1109/CCGrid49817.2020.00-52.

[5] Vaikkunth Mugunthan, Antigoni Polychroniadou, David Byrd, and Tucker Hybinette Balch. Smpai: Secure multi-party computation for federated learning. In *Proceedings of the NeurIPS 2019 Workshop on Robust AI in Financial Services*, volume 21. MIT Press Cambridge, MA, USA, 2019.

[6] Ekanut Sotthiwat, Liangli Zhen, Zengxiang Li, and Chi Zhang. Partially encrypted multi-party computation for federated learning. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 828–835. IEEE, 2021. https://doi.org/10.1109/CCGrid51090.2021.00101.

[7] 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. https://doi.org/10.1007/978-3-642-37682-5_15.

[8] 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. https://doi.org/10.1007/978-3-319-11599-3_4.

[9] Peeter Laud and Martin Pettai. Secure multiparty sorting protocols with covert privacy. In *Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016. Proceedings 21*, pages 216–231. Springer, 2016. https://doi.org/10.1007/978-3-319-47560-8_14.

[10] 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.

[11] 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.

[12] 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*, pages 887–903, 2019. https://doi.org/10.1145/3319535.3354238.

[13] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. *Cryptology ePrint Archive*, 2021. https://doi.org/10.14722/ndss.2022.24141.

[14] Xiangfu Song, Dong Yin, Jianli Bai, Changyu Dong, and Ee-Chien Chang. Secret-shared shuffle with malicious security. *Cryptology ePrint Archive*, 2023. https://doi.org/10.14722/ndss.2024.24021.

[15] 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. ISBN 978-3-662-45608-8.

[16] 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. https://doi.org/10.1007/978-3-642-24861-0_18.

[17] 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. https://doi.org/10.1007/978-3-030-64840-4_12.

[18] Peeter Laud. Linear-time oblivious permutations for spdz. In *Cryptology and Network Security: 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings 20*, pages 245–252. Springer, 2021. https://doi.org/10.1007/978-3-030-92548-2_13.

[19] S. Dov Gordon, Phi Hung Le, and Daniel McVicker. Linear communication in malicious majority mpc. CCS '23, page 2173–2187, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400700507. https://doi.org/10.1145/3576915.3623162. URL https://doi.org/10.1145/3576915.3623162.

[20] Vipul Goyal, Yifan Song, and Chenzhi Zhu. Guaranteed Output Delivery Comes Free in Honest Majority MPC. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020*, pages 618–646, Cham, 2020. Springer International Publishing. ISBN 978-3-030-56880-1. https://doi.org/10.1007/978-3-030-56880-1_22.

[21] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, 2018.

[22] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012. https://doi.org/10.1007/978-3-642-32009-5_38.

[23] Ivan Damgård and Jesper Buus Nielsen. Scalable and unconditionally secure multiparty computation. In *Annual International Cryptology Conference*, pages 572–590. Springer, 2007. https://doi.org/10.1007/978-3-540-74143-5_32.

[24] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981. https://doi.org/10.1145/358549.358563.

[25] 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. https://doi.org/10.1007/978-3-540-70936-7_30.

[26] Jens Groth. A verifiable secret shuffle of homomorphic encryptions. *Journal of Cryptology*, 23:546–579, 2010. https://doi.org/10.1007/3-540-36288-6_11.

[27] Susan Hohenberger, Guy N Rothblum, Abhi Shelat, and Vinod Vaikuntanathan. Securely obfuscating re-encryption. In *Theory of Cryptography Conference*, pages 233–252. Springer, 2007. https://doi.org/10.1007/978-3-540-70936-7_13.

[28] 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.

[29] Daniel Escudero, Satrajit Ghosh, Marcel Keller, Rahul Rachuri, and Peter Scholl. Improved primitives for mpc over mixed arithmetic-binary circuits. In *Advances in Cryptology–CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II 40*, pages 823–852. Springer, 2020. https://doi.org/10.1007/978-3-030-56880-1_29.

[30] Marcel Keller, Emmanuela Orsini, and Peter Scholl. Mascot: faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 830–842, 2016. https://doi.org/10.1145/2976749.2978357.

[31] Zuzana Beerliová-Trubíniová and Martin Hirt. Perfectly-secure mpc with linear communication complexity. In *Theory of Cryptography Conference*, pages 213–230. Springer, 2008. https://doi.org/10.1007/978-3-540-78524-8_13.

[32] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001. https://doi.org/10.1109/SFCS.2001.959888.

[33] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In *Theory of Cryptography Conference*, pages 285–304. Springer, 2006. https://doi.org/10.1007/11681878_15.

[34] Marcel Keller. MP-SPDZ: A versatile framework for multi-party computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020. https://doi.org/10.1145/3372297.3417872.

A Security Proof for Semi-honest Secure Shuffle

Definition 6 (Security for Semi-honest Shuffle). *Suppose there is an adversary $\mathcal{A}$, which can corrupt up to $n-1$ parties. The parties are to execute protocol* $\text{Shuffle}_{\text{off}}$ *followed by a protocol* $\text{Shuffle}_{\text{on}}$ *with the correlation just generated. After protocol* $\text{Shuffle}_{\text{off}}$ *finishes, $\mathcal{A}$ is to choose a vector $\mathbf{x}$, and input it with Π_{input}. At the end of the protocol* $\text{Shuffle}_{\text{on}}$*, $\mathcal{A}$ obtains an overall view, denoted by* $\text{view}_{\mathcal{A}}$*, which contains all values visible to corrupted parties.*

Let the set of corrupted parties by $T \subsetneq [n]$, and $\bar{T}$ the set of honest parties. The shuffle protocol is said to be semi-honest secure, if for any possible permutations $(\pi'_i)_{i\in\bar{T}}$,

$$\Pr[(\pi_i)_{i\in\bar{T}} = (\pi'_i)_{i\in\bar{T}} \mid \text{view}_{\mathcal{A}}] = \frac{1}{(m!)^{|\bar{T}|}}.$$

I.e. the tuple of all permutations performed by honest party $P_i \in \bar{T}$, which is $(\pi_i)_{i\in\bar{T}}$, is uniform over all possible permutations, even if conditioned on the view of adversary. This can be expressed equivalently via mutual information, by stating

$$I((\pi_i)_{i\in\bar{T}}; \text{view}_{\mathcal{A}}) = 0,$$

i.e. the permutations of honest parties are independent of the view of adversary.

Theorem 1. *The semi-honest two-phase shuffle protocol* $\text{Shuffle}_{\text{off}}$ *and* $\text{Shuffle}_{\text{on}}$ *is secure, in the sense that it satisfies the above definition for semi-honest shuffle protocol. That is, the view of any adversary corrupting up to $n-1$ parties is independent of the permutations of honest parties, i.e.*

$$I((\pi_i)_{i\in\bar{T}}; \text{view}_{\mathcal{A}}) = 0.$$

Proof. The core idea of proof is that, given the view of the adversary and any choices of possible permutations of honest parties, there is exactly one assignment to $(\mathbf{r}_i)_{i\in[n]}$ and $(\mathbf{z}_i)_{i\in\bar{T}}$, such that the view of adversary is not altered, yet the result becomes applying new permutations to $\mathbf{x}$.

For a clear demonstration, let's suppose the adversary corrupts exactly $n-1$ parties, leaving P_k the only honest party. The view of adversary contains

$$\text{view}_{\mathcal{A}} = \begin{Bmatrix} \mathbf{x} & & & & & & & \\ \pi_1 & \pi_2 & \cdots & \pi_{k-1} & & \pi_{k+1} & \cdots & \pi_n \\ \mathbf{z}_1 & \mathbf{z}_2 & \cdots & \mathbf{z}_{k-1} & & \mathbf{z}_{k+1} & \cdots & \mathbf{z}_n \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_{k-1} & \mathbf{y}_k & \mathbf{y}_{k+1} & \cdots & \mathbf{y}_n \end{Bmatrix}.$$

Note that the view contains all $\mathbf{y}_i$, including $\mathbf{y}_k$, since it's the message from P_k to P_{k+1}.

Let's suppose honest party P_k has originally chosen π_k, but now we want the result to be as if it has chosen π'_k. We claim that, there is exactly one assignment to $\{\mathbf{z}_i, \mathbf{r}_i, \pi_i\}_{i\in[n]}$, denoted by $(\mathbf{z}'_i)_{i\in[n]}$, $(\mathbf{r}'_i)_{i\in[n]}$ and $(\pi'_i)_{i\in[n]}$, such that

1. For each $i \neq k$, $\pi'_i = \pi_i$.

2. For each $i \neq k$, $\mathbf{z}_i = \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i = \pi'_{i-1}(\mathbf{r}'_{i-1}) - \mathbf{r}'_i = \mathbf{z}'_i$.
3. For P_k, $\mathbf{z}'_k = \pi'_{k-1}(\mathbf{r}'_{k-1}) - \mathbf{r}'_k$, which may not equal $\mathbf{z}_k$.
4. The output will be

$$[\![\overline{\pi}'_n]\!] \coloneqq [\![\pi'_n \circ \pi'_{n-1} \circ \cdots \circ \pi'_1(\mathbf{x})]\!].$$

Note that the above statements imply that the view of corrupted parties is not altered by replacing π_k with arbitrary π'_k, and even remains consistent with the new meaning we assign to it (i.e. shuffling with π'_i). Hence, since π'_k is an arbitrary permutation, this indicates that any permutation π'_k is equally possible in the view of adversary.

This exact assignment can be solved from the constraints put by the view. Suppose first $k = 1$, note that

$$\mathbf{z}_1 = \mathbf{x} - \mathbf{r}_1,\ \mathbf{y}_1 = \pi_1(\mathbf{z}_1).$$

Since $\mathbf{y}_1$ appears in the view, it demands $\mathbf{z}'_1 = \pi'^{-1}_1(\mathbf{y}_1)$. Luckily, $\mathbf{z}_1$ and $\mathbf{z}'_1$ does not appear in the view, hence they can be different. And this gives the only $\mathbf{r}'_1 = \mathbf{x} - \pi'^{-1}_1(\mathbf{y}_1)$. By examination,

$$\mathbf{y}_1 = \pi_1(\mathbf{x}) - \pi_1(\mathbf{r}_1) = \pi'_1(\mathbf{x}) - \pi'_1(\mathbf{r}'_1) = \mathbf{y}'_1,$$

which coincides with the requirement of $\mathbf{y}_1 = \mathbf{y}'_1$.

Now that it is demanded that $\mathbf{z}_2 = \mathbf{z}'_2 = \pi'_1(\mathbf{r}'_1) - \mathbf{r}'_2$, this gives exactly one solution to $\mathbf{r}'_2 = \pi'_1(\mathbf{r}'_1) - \mathbf{z}_2$. Since $\pi_2 = \pi'_2$, $\mathbf{y}'_1 = \mathbf{y}_1$ and $\mathbf{z}'_2 = \mathbf{z}_2$, we have $\mathbf{y}'_2 = \mathbf{y}_2$ naturally. What's more, it can be checked that

$$\mathbf{y}'_2 = \mathbf{y}_2 = \pi_2(\mathbf{y}_1 + \mathbf{z}_2) = \overline{\pi}'_2(\mathbf{x}) - \pi'_2(\mathbf{r}'_2),$$

which follows trivially from the fact that

$$\mathbf{y}_1 = \mathbf{y}'_1 = \pi'_1(\mathbf{x}) - \pi'_1(\mathbf{r}'_1),$$

and

$$\mathbf{z}_2 = \mathbf{z}'_2 = \pi'_1(\mathbf{r}'_1) - \mathbf{r}'_2,$$

and $\pi_2 = \pi'_2$.

Now following this path, we can assign new value to each $\mathbf{r}_i$ as $\mathbf{r}'_i$ for $i \geq 2$, such that $\mathbf{z}'_i = \mathbf{z}_i$, yet the "explanation" of $\mathbf{y}_i$ is replaced by

$$\mathbf{y}_i = \mathbf{y}'_i = \overline{\pi}'_i(\mathbf{x}) - \pi'_i(\mathbf{r}'_i).$$

And at the end, party P_n obtains

$$\mathbf{y}_n = \mathbf{y}'_n = \overline{\pi}'_n(\mathbf{x}) - \pi'_n(\mathbf{r}'_n).$$

After broadcasting and adding it with $[\![\pi'_n(\mathbf{r}'_n)]\!]$, the result is indeed $\overline{\pi}'_n(\mathbf{x})$.

Hence, if P_1 is honest, by seeing only the values in the view, π_1 can be any permutation π'_1. And the probabilities of π_1 being any π'_1 are equal, since there

is exactly one set of assignment to $\mathbf{r}_i$ that supports the claim $\pi_1 = \pi'_1$. As the adversary does not know any $\mathbf{r}_i$ or $\pi_i(\mathbf{r}_i)$, it cannot prefer any value of π_1 over another. Thus, the choice of π_1 is independent of the view of $\mathcal{A}$.

The case for $k \geq 2$ can be deduced trivially from the case of $k = 1$. In the case of $k \geq 2$, all $\mathbf{r}_i$ for $i < k$ does not need to be modified, since they are already uniquely decided given $\pi_1, ..., \pi_{k-1}$ and $\mathbf{z}_1, ..., \mathbf{z}_{k-1}$ and $\mathbf{x}$. Further, they cannot be modified, since $\mathcal{A}$ can deduce the exact value of $\mathbf{r}_1, ..., \mathbf{r}_{k-1}$. However, since the adversary lacks the view of $\mathbf{z}_k$, we are able to modify $\mathbf{r}_k$ as $\mathbf{r}'_k$, and the same argument will continue to be valid.

Argument for the case where $\mathcal{A}$ corrupts arbitrary $T \subsetneq [n]$ can be easily generalized from above argument, and hence we simply claim it true here. □

B Security Proof for Malicious Secure Shuffle

B.1 Fully Developed Shuffle Protocol

Below in Algorithm 7, 8 and 9 are our final version of shuffle protocol for malicious security, whose online communication, computation and round complexity is $O(nm)$, $O(nm)$ and $O(n)$, respectively. In the following, we are to prove that this construction satisfies UC security under $\mathcal{F}_{\text{MPC}}$-hybrid model.

First, we present an extended malicious shuffle correlation. Compared to Definition 5, this definition includes additional auxiliary information to reduce online computation.

Definition 7 ((Extended) Malicious Shuffle Correlation). *The shuffle correlation for malicious setting is defined as*

$$\text{cor} = \left\{ \begin{array}{ccccccc} \langle\beta\rangle & \langle\mathbf{r}\rangle & \langle\beta\mathbf{r}\rangle & \langle\pi_1^{-1}(\mathbf{r}'_1)\rangle & \langle\mathbf{s}\rangle & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\ \langle\mathbf{r}'_1\rangle & \langle\mathbf{r}'_2\rangle & \langle\mathbf{r}'_3\rangle & \langle\mathbf{r}'_4\rangle & \langle\mathbf{r}'_5\rangle & \cdots & \langle\mathbf{r}'_n\rangle \\ & \langle c_2\rangle & \langle c_3\rangle & \langle c_4\rangle & \langle c_5\rangle & ... & \langle c_n\rangle \\ & \langle d_2\rangle & \langle d_3\rangle & \langle d_4\rangle & \langle d_5\rangle & ... & \langle d_n\rangle \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n \end{array} \right\},$$

where

1. *π_i is an m-permutation known only to P_i. It is sampled uniformly from all possible m-permutations by P_i (if P_i is honest).*
2. *$\langle\beta\rangle$ is a secret shared random variable. It plays the role of the authentication key, i.e. MAC key.*
3. *$\langle\mathbf{r}\rangle, \langle\mathbf{s}\rangle, \langle\mathbf{r}'_1\rangle, ..., \langle\mathbf{r}'_n\rangle$ are secret shared vectors of length m, with each entry independently uniformly random. $\langle\beta\mathbf{r}\rangle$ is secret shared $\beta \cdot \mathbf{r}$.*
4. *c_i is uniformly random, with*

$$d_i = \sum_{j=1}^{m} c_i^{j-1} \mathbf{r}'_i.$$

5. $\mathbf{z}_i = (\mathbf{z}_i^1, \mathbf{z}_i^2)$, *where each* $\mathbf{z}_i^j$ *is a vector of length* m. *The entries of* $\mathbf{z}_i^1$ *are uniformly random, under the constraint*

$$\begin{cases} \mathbf{z}_i^2 = \beta \mathbf{z}_i^1 + \mathbf{r}'_{i-1} - \pi_i^{-1}(\mathbf{r}'_i) & \forall i = 2, 3, ..., n \\ \mathbf{s} = \pi_n(\pi_{n-1}(\cdots \pi_2(\pi_1(\mathbf{r}) - \mathbf{z}_2^1) - \mathbf{z}_3^1 \cdots) - \mathbf{z}_n^1) \end{cases}$$

Algorithm 7 cor $\leftarrow \text{Shuffle}_{\text{off}}(m, P_1 : \pi_1, ..., P_n : \pi_n)$

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

Generate random value $\langle \beta \rangle$.
for $i = 1$ to n **do parallel**
  Generate random vectors $\langle \mathbf{r}_i \rangle$ and $\langle \mathbf{r}'_i \rangle$.
  $\langle \beta \mathbf{r}_i \rangle \leftarrow \Pi_{\text{mul}}(\langle \mathbf{r}_i \rangle, \langle \beta \rangle)$
  $\left(\langle \pi_i(\mathbf{r}_i) \rangle, \langle \pi_i(\beta \mathbf{r}_i) \rangle, \langle \pi_i(\mathbf{r}'_i) \rangle\right) \leftarrow \Pi_{\text{perm}}\left(P_i : \pi_i, \langle \mathbf{r}_i \rangle, \langle \beta \mathbf{r}_i \rangle, \langle \mathbf{r}'_i \rangle\right)$
  if $i \geq 2$ **then**
    $\langle \mathbf{z}_i^1 \rangle \leftarrow \langle \pi_{i-1}(\mathbf{r}_{i-1}) \rangle - \langle \mathbf{r}_i \rangle$
    $\langle \mathbf{z}_i^2 \rangle \leftarrow \langle \pi_{i-1}(\beta \mathbf{r}_{i-1}) \rangle - \langle \beta \mathbf{r}_i \rangle + \langle \pi_{i-1}(\mathbf{r}'_{i-1}) \rangle - \langle \mathbf{r}'_i \rangle$
    $\langle \mathbf{z}_i \rangle \coloneqq (\langle \mathbf{z}_i^1 \rangle, \langle \mathbf{z}_i^2 \rangle)$
    $\langle c_i \rangle \leftarrow \Pi_{\text{rand}}()$
    Compute $\langle c_i \rangle, \langle c_i^2 \rangle, ..., \langle c_i^{m-1} \rangle$. ▷ $O(m)$ calls to Π_{mul}.
    $\langle d_i \rangle \leftarrow \sum_{j=1}^{m} \langle c_i^{j-1} \rangle \cdot \langle \pi_{i-1}(\mathbf{r}'_{i-1}(j)) \rangle$
  end if
end for
for $i = 2$ to n **do parallel**
  Open $\mathbf{z}_i$ to P_i.
end for

$$\text{Return cor} = \left\{ \begin{array}{ccccccc} \langle \beta \rangle & \langle \mathbf{r}_1 \rangle & \langle \beta \mathbf{r}_1 \rangle & \langle \mathbf{r}'_1 \rangle & \langle \pi_n(\mathbf{r}_n) \rangle & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\ \langle \pi_1(\mathbf{r}'_1) \rangle & \langle \pi_2(\mathbf{r}'_2) \rangle & \langle \pi_3(\mathbf{r}'_3) \rangle & \langle \pi_4(\mathbf{r}'_4) \rangle & \langle \pi_5(\mathbf{r}'_4) \rangle & \cdots & \langle \pi_n(\mathbf{r}'_n) \rangle \\ & \langle c_2 \rangle & \langle c_3 \rangle & \langle c_4 \rangle & \langle c_5 \rangle & ... & \langle c_n \rangle \\ & \langle d_2 \rangle & \langle d_3 \rangle & \langle d_4 \rangle & \langle d_5 \rangle & ... & \langle d_n \rangle \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n \end{array} \right\}$$

B.2 Roadmap and Ideal Functionality

Before we dive into a formal security proof, let's give some intuitions on why the above offline and online protocol is secure.

First recall that the offline phase of the protocol $\text{Shuffle}_{\text{off}}$ has in essence done nothing besides generating random value, doing some multiplications, calling the functionality Π_{perm} and opening some values. Since all these operations are supported by $\mathcal{F}_{\text{MPC}}$, the security follows trivially.

In the online phase $\text{Shuffle}_{\text{on}}$, what could potentially harm the security is $\mathbf{y}_i$. Since $\mathbf{z}_i$ and π_i is by design only known to party P_i, honest parties could

Algorithm 8 PartialVerify$(P_{i+1}, \mathbf{y}_i^1, \mathbf{y}_i^2, \langle\beta\rangle, \langle c_{i+1}\rangle, \langle d_{i+1}\rangle)$

Require: $\langle c_{i+1}\rangle$ and $\langle d_{i+1}\rangle$ come from malicious shuffle correlation.
Ensure: If honest P_{i+1} does not receive $\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}'_i)$, abort w.h.p.
Parties open $\langle c_{i+1}\rangle$ to P_{i+1}.
P_{i+1} computes $w_b \coloneqq \sum_{j=1}^{m} c_{i+1}^{j-1}\mathbf{y}_i^b(j)$, $b \in \{1,2\}$.
P_{i+1} broadcasts w_1 and w_2.
$\langle e\rangle \leftarrow w_1 \cdot \langle\beta\rangle - w_2 - \langle d_{i+1}\rangle$.
Open e for all parties.
Abort if $e \neq 0$.

Algorithm 9 $\langle\overline{\pi}_n(\mathbf{x})\rangle \leftarrow \text{Shuffle}_{\text{on}}(\langle\mathbf{x}\rangle)$

Require: Fresh cor $= \left\{\begin{array}{ccccccc} \langle\beta\rangle & \langle\mathbf{r}_1\rangle & \langle\beta\mathbf{r}_1\rangle & \langle\mathbf{r}'_1\rangle & \langle\pi_n(\mathbf{r}_n)\rangle & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\ \langle\pi_1(\mathbf{r}'_1)\rangle & \langle\pi_2(\mathbf{r}'_2)\rangle & \langle\pi_3(\mathbf{r}'_3)\rangle & \langle\pi_4(\mathbf{r}'_4)\rangle & \langle\pi_5(\mathbf{r}'_4)\rangle & \cdots & \langle\pi_n(\mathbf{r}'_n)\rangle \\ & \langle c_2\rangle & \langle c_3\rangle & \langle c_4\rangle & \langle c_5\rangle & ... & \langle c_n\rangle \\ & \langle d_2\rangle & \langle d_3\rangle & \langle d_4\rangle & \langle d_5\rangle & ... & \langle d_n\rangle \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n \end{array}\right\}$.
Ensure: Return $\langle\overline{\pi}_n(\mathbf{x})\rangle$ if the protocol does not abort.
$\langle\beta\mathbf{x}\rangle \leftarrow \Pi_{\text{mul}}(\langle\beta\rangle, \langle\mathbf{x}\rangle)$
$\langle\mathbf{z}_1^1\rangle \leftarrow \langle\mathbf{x}\rangle - \langle\mathbf{r}_1\rangle$
$\langle\mathbf{z}_1^2\rangle \leftarrow \langle\beta\mathbf{x}\rangle - \langle\beta\mathbf{r}_1\rangle - \langle\mathbf{r}'_1\rangle$
Open $\mathbf{z}_1 \coloneqq (\mathbf{z}_1^1, \mathbf{z}_1^2)$ to P_1.
P_1 computes and broadcasts $\mathbf{y}_1 = \pi_1(\mathbf{z}_1)$.
for $i = 2$ to $n-1$ **do**
  P_i computes and sends to P_{i+1} $\mathbf{y}_i = \pi_i(\mathbf{z}_i + \mathbf{y}_{i-1})$.
end for
P_n computes and broadcasts $\mathbf{y}_n = \pi_n(\mathbf{z}_n + \mathbf{y}_{n-1})$.
for $i = 1$ to $n-1$ **do**
  PartialVerify$(P_{i+1}, \mathbf{y}_i^1, \mathbf{y}_i^2, \langle\beta\rangle, \langle c_{i+1}\rangle, \langle d_{i+1}\rangle)$
end for
Verify$(\mathbf{y}_n^1, \mathbf{y}_n^2, \langle\beta\rangle, \langle\pi_n(\mathbf{r}'_n)\rangle)$
$\langle\mathbf{y}_n^1\rangle \leftarrow \Pi_{\text{input}}(\mathbf{y}_n^1)$.
$\langle\mathbf{x}'\rangle \leftarrow \langle\mathbf{y}_n^1\rangle + \langle\pi_n(\mathbf{r}_n)\rangle$
Return $\langle\mathbf{x}'\rangle$.

not tell if what P_i sends is indeed $\mathbf{y}_i = \pi_i(\mathbf{z}_i + \mathbf{y}_{i-1})$. However, note that in our construction, the parties will be able to check whether the correlation implanted in $\mathbf{y}_i$ is sound by PartialVerify. Intuitively, as β and $\mathbf{r}'_i$ are both never revealed, P_i cannot forge a fake $\mathbf{y}'_i$ and still pass the test. This guarantees all $\mathbf{y}_i$ to be correct, and hence the result.

Hence, in this section, we prove first that adversary cannot learn the shuffle authentication key β before entering any PartialCheck, even if it deviates from the protocol. This is, of course, of vital importance. Then, we will prove that due to having no knowledge to β, the adversary cannot forge $\mathbf{y}_i$ and pass the test with non-negligible probability, nor can it forge w_1 and w_2 broadcast in PartialVerify and pass the test with non-negligible probability. This means that if the protocol does not abort, w.h.p. all messages sent to honest parties are correctly computed according to the protocol, which roughly makes the adversary semi-honest. Finally, we construct a simulator for the entire process, which either aborts with same probability at the same step in a real execution, or proceeds and outputs a view indistinguishable from the view in real execution, which finally concludes the UC security.

Below, we assume the adversary $\mathcal{A}$ has corrupted all parties in $T \subsetneq [n]$, leaving $\bar{T}$ the set of honest parties.

The formal description of the functionality of the protocol is as follows. The security is defined by the indistinguishability between the execution of protocol in real world with an execution of the protocol in ideal world, with the help of ideal functionality $\mathcal{F}_{\text{shuffle}}$.

Definition 8. *(Ideal Functionality of* $\text{Shuffle}_{\text{off}}$*) Denote by* $\mathcal{F}_{\text{off}}$ *an ideal functionality, which is* $\mathcal{F}_{\text{MPC}}$ *equipped with an additional (ideal) command* Π_{off}*. This command takes as input a public length* m *and each* m*-permutation* π_i *from* P_i*, and generates a shuffle correlation. That is,*

$$\text{cor} = \left\{ \begin{array}{ccccccccc} \langle\beta\rangle & \langle\mathbf{r}\rangle & \langle\beta\mathbf{r}\rangle & \langle\pi_1^{-1}(\mathbf{r}'_1)\rangle & \langle\mathbf{s}\rangle & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n \\ \langle\mathbf{r}'_1\rangle & \langle\mathbf{r}'_2\rangle & \langle\mathbf{r}'_3\rangle & \langle\mathbf{r}'_4\rangle & \langle\mathbf{r}'_5\rangle & \cdots & \langle\mathbf{r}'_n\rangle \\ & \langle c_2\rangle & \langle c_3\rangle & \langle c_4\rangle & \langle c_5\rangle & \ldots & \langle c_n\rangle \\ & \langle d_2\rangle & \langle d_3\rangle & \langle d_4\rangle & \langle d_5\rangle & \ldots & \langle d_n\rangle \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n \end{array} \right\},$$

which satisfies Definition 7.

Definition 9. *(Ideal Functionality of Shuffle) Denote by* $\mathcal{F}_{\text{shuffle}}$ *an ideal functionality, which is the ideal functionality* $\mathcal{F}_{\text{MPC}}$ *equipped with an additional command* Π_{shuffle}*. This command takes as input* $\langle\mathbf{x}\rangle$*, and stores* $\langle\pi(\mathbf{x})\rangle$ *in the ideal functionality, where* π *is uniformly randomly drawn by* $\mathcal{F}_{\text{shuffle}}$*.*

We begin with three lemmas. The first states that the offline phase is secure, in the sense that the adversary cannot learn information more than $\mathbf{z}_i$ that is revealed to it. The second and third describe the security of protocol Verify and PartialVerify, respectively.

Lemma 1 (Security of Offline Phase). *The protocol* $\text{Shuffle}_{\text{off}}$ *is UC secure in the* $\mathcal{F}_{\text{MPC}}$*-hybrid model. Further, the view of adversary (besides* m*) in offline phase is equivalent to*

$$\text{view}_{\mathcal{A}} = \{\pi_i, \mathbf{z}_i\}_{i \in T}.$$

Note that since $\mathbf{z}_1$ *is not included in the offline phase, it should be understood as a null value here.*

Proof. The protocol $\text{Shuffle}_{\text{off}}$ is almost secure by definition. Nevertheless, as a warming up, we prove the security by constructing the simulator.

First, we note that the protocol $\text{Shuffle}_{\text{off}}$ correctly implements ideal functionality $\mathcal{F}_{\text{off}}$ by outputting a correlation that satisfies Definition 7. This can be verified directly by first noting that the constraints are satisfied. Moreover, note that since in the protocol,

$$\langle \mathbf{z}_i^1 \rangle \leftarrow \langle \pi_{i-1}(\mathbf{r}_{i-1}) \rangle - \langle \mathbf{r}_i \rangle,$$
$$\langle \mathbf{z}_i^2 \rangle \leftarrow \langle \pi_{i-1}(\beta \mathbf{r}_{i-1}) \rangle - \langle \beta \mathbf{r}_i \rangle + \langle \pi_{i-1}(\mathbf{r}'_{i-1}) \rangle - \langle \mathbf{r}'_i \rangle.$$

This means that after fixing $\pi_i, \beta, \mathbf{r}'_i, \mathbf{r}_1$ and $\mathbf{r}_n$, there is a bijection between set

$$\{(\mathbf{r}_i)_{2 \le i \le n-1} : \mathbf{r}_i \in \mathbb{F}^m\} \text{ and } \{(\mathbf{z}_i)_{2 \le i \le n} : \text{ constraint is satisfied}\}.$$

Since the protocol generates $\mathbf{r}_i$ for $i \in \{2, 3, ..., n-1\}$ uniformly, this means $\mathbf{z}_i$ will be uniform under the constraint, as required by definition.

Consider a simulator $\mathcal{S}$ that acts as follows. At the start of the game, $\mathcal{E}$ first sends to $\mathcal{F}_{\text{off}}$ the input of honest parties, which is π_i for P_i. $\mathcal{S}$ simply follows the protocol, since $\mathcal{A}$ will obtain no output by calling to ideal $\mathcal{F}_{\text{MPC}}$ in the real world. In the step of calling $\Pi_{\text{perm}}(P_i : \pi_i, ...)$ for corrupted party P_i, since in the real world execution $\mathcal{A}$ will need to send π_i to Π_{perm}, $\mathcal{S}$ will receive (purported) π_i from $\mathcal{E}$. Hence, $\mathcal{S}$ can send π_i to $\mathcal{F}_{\text{off}}$, as part of the input required by command Π_{off}.

When all inputs are gathered, $\mathcal{F}_{\text{off}}$ will send $\mathbf{z}_i$ in the shuffle correlation to $\mathcal{S}$, for each corrupted P_i with $i \ge 2$. $\mathcal{S}$ is then able to simulate the receiving of $\mathbf{z}_i$ for corrupted party P_i, as if P_i has received $\mathbf{z}_i$ in the real world.

Note also that during the process, whenever $\mathcal{E}$ demands $\mathcal{S}$ to abort, $\mathcal{S}$ simply aborts as $\mathcal{A}$ does. If the protocol does not eventually abort, $\mathcal{S}$ sends "continue" to $\mathcal{F}_{\text{off}}$, who then reveals its final state (i.e. the entire shuffle correlation cor) to $\mathcal{E}$. This shuffle correlation is of course consistent with the view of adversary $\mathcal{A}$, as the $\mathbf{z}_i$ and π_i are the same. Hence, the above process is perfect, i.e. is identical with the execution in the real world.

Additionally, it is also clear from above simulation that the adversary's view is $\mathbf{z}_i$ opened to it during the protocol, plus the π_i chosen by itself. □

Lemma 2. *The protocol* $\text{Verify}(a, b, \langle \beta \rangle, \langle r \rangle)$ *is secure, in the sense that if* $\beta a \neq b + r$*, the test fails immediately and all parties abort.*

Proof. Suppose $\beta a \neq b + r$. In protocol, parties compute and open

$$d \leftarrow \alpha \langle \beta \rangle - b - \langle r \rangle.$$

By the merit of the functionality $\mathcal{F}_{\mathrm{MPC}}$, it is clear that the protocol will abort if $d \neq 0$, since all computations included in the protocol is via $\mathcal{F}_{\mathrm{MPC}}$. □

Lemma 3. *The protocol* PartialVerify$(P_{i+1}, \mathbf{y}_i^1, \mathbf{y}_i^2, \langle\beta\rangle, \langle c_{i+1}\rangle, \langle d_{i+1}\rangle)$ *is secure, in the sense that if honest* P_{i+1} *does not receive valid* $\mathbf{y}_i$ *such that* $\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}_i')$*, it aborts with probability*

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

Remark. We remark that receiving "valid" $\mathbf{y}_i$ such that $\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}_i')$ is slightly different from receiving "correct" $\mathbf{y}_i$ specified by the protocol, since the adversary might (somehow) forge $\mathbf{y}_i' \neq \mathbf{y}_i$ that satisfies this correlation.

This lemma claims only that upon receiving $\mathbf{y}_i$ that does not satisfy the correlation, the parties abort w.h.p. In later proof, however, it turns out that the adversary cannot forge $\mathbf{y}_i' \neq \mathbf{y}_i$ that satisfies this correlation, which makes the statement "receiving valid $\mathbf{y}_i$" equivalent to "receiving correct $\mathbf{y}_i$" (except with negligible probability).

Proof. Suppose P_{i+1} is honest, and $\beta\mathbf{y}_i^1 \neq \mathbf{y}_i^2 + \pi_i(\mathbf{r}_i')$. By Lemma 1, the offline phase of the shuffle is secure. Thus, c_{i+1} is uniformly random, and

$$d_{i+1} = \sum_{j=1}^{m} c_{i+1}^{j-1} \pi_i(\mathbf{r}_i')(j).$$

Consider the polynomial

$$f(X) = \sum_{j=1}^{m} \left(\beta \cdot \mathbf{y}_i^1(j) - \mathbf{y}_i^2(j) - \pi_i(\mathbf{r}_i')(j)\right) \cdot X^{j-1}.$$

As $\beta\mathbf{y}_i^1 \neq \mathbf{y}_i^2 + \pi_i(\mathbf{r}_i')$, f is a non-zero polynomial of degree at most $m-1$. Note that the protocol PartialVerify opens exactly $f(c_{i+1})$ to all parties. As c_{i+1} appears only in $\langle c_{i+1}\rangle$ and $\langle d_{i+1}\rangle$, which are never opened before the protocol PartialVerify, c_{i+1} appears uniformly random to the adversary. Thus, as f has at most $m-1$ roots in $\mathbb{F}$, c_{i+1} happens to be one of them with probability

$$p_{\mathrm{pass}} = \Pr(f(c_{i+1}) = 0 \mid c_{i+1} \leftarrow \mathbb{F}) \leq \frac{m-1}{|\mathbb{F}|}.$$

Thus,

$$p_{\mathrm{abort}} = 1 - p_{\mathrm{pass}} \geq 1 - \frac{m-1}{|\mathbb{F}|} \geq 1 - \frac{m-1}{2^\kappa}.$$

□

B.3 Ignorance of Shuffle Authentication Key

Before proving the security of entire protocol, we first propose a somewhat bizarre lemma, which claims that the "view" of all parties up to checking phase

(i.e. before executing any PartialVerify) cannot break the shuffle authentication key β. This seems bizarre, because all parties united together should reveal everything in any multiparty computation, since there is nothing then to protect. The point is that this "view" does not contain the value of β and $\mathbf{r}'_i$, which is hidden behind $\mathcal{F}_{\text{MPC}}$.

Lemma 4. *Consider the following view:*

$$\text{view} = \begin{Bmatrix} \mathbf{x} & & & \\ \mathbf{z}_1 & \mathbf{z}_2 & \cdots & \mathbf{z}_n \\ \pi_1 & \pi_2 & \cdots & \pi_n \end{Bmatrix}.$$

This view consists of $\mathbf{x}$*, all secret permutations held by parties and all opened value during both the offline and online phase of the protocol, before entering any* PartialVerify*. Note that* $\mathbf{y}_i$ *can all be deduced from the view.*

This lemma claims that, this view is independent of β*, i.e.*

$$I(\beta; \text{view}) = 0.$$

Or equivalently,

$$\Pr[\beta = \beta' \mid \text{view}] = \frac{1}{|\mathbb{F}|},$$

for any $\beta' \in \mathbb{F}$*.*

Proof. This view can be expanded as

$$\text{view} = \begin{Bmatrix} \mathbf{x} & & \\ \mathbf{x} - \mathbf{r}_1 & \pi_1(\mathbf{r}_1) - \mathbf{r_2} & \cdots \\ \beta\mathbf{x} - \beta\mathbf{r}_1 - \mathbf{r}'_1 & \pi_1(\beta\mathbf{r}_1 + \mathbf{r}'_1) - \beta\mathbf{r}_2 - \mathbf{r}'_2 & \cdots \\ \pi_1 & \pi_2 & \cdots \end{Bmatrix},$$

where the second row is $\mathbf{z}_i^1$ and third row $\mathbf{z}_i^2$. Now note that, we may further simplify this view, and replace it by

$$\text{view} = \begin{Bmatrix} \mathbf{x} & & \\ \mathbf{r}_1 & \mathbf{r}_2 & \cdots \\ \beta\mathbf{x} - \beta\mathbf{r}_1 - \mathbf{r}'_1 & \pi_1(\beta\mathbf{r}_1 + \mathbf{r}'_1) - \beta\mathbf{r}_2 - \mathbf{r}'_2 & \cdots \\ \pi_1 & \pi_2 & \cdots \end{Bmatrix},$$

since this view can deduce all the second row in former view, and vice versa.

Now it should be clear that this view is independent of β, since the third row is the only row containing term β, yet each term is masked by an independent random vector $\mathbf{r}'_i$. Stated otherwise, for each possible assignment of β, there is exactly one set of assignment to all $\mathbf{r}'_i$ such that the above view does not change.

Hence, the view is independent of β. □

What can this lemma do? No surprisingly, it turns out that what it really means is that, however the corrupted parties act, they cannot deduce any information about β, before entering the checking phase. This is formally stated in the following theorem.

Theorem 2. *Before execution of any* PartialVerify *protocol, the adversary cannot deduce any information about β from the combined protocol* Shuffle*, however it acts. Hence, the probability of the adversary guessing β correct is $1/|\mathbb{F}|$, the same as randomly drawing a field element.*

Proof. Let's consider the view of adversary. Firstly in the offline phase $\text{Shuffle}_{\text{off}}$, the view is exactly $\{\mathbf{z}_i\}_{i\in T\setminus\{1\}}$ and $\{\pi_i\}_{i\in T}$. This is due to Lemma 1. As the offline phase consists only of calls to $\mathcal{F}_{\text{MPC}}$ subroutines, however the adversary acts, it cannot learn more than this view.

In the online phase, the adversary $\mathcal{A}$ learns in extra the message of honest party P_k, which is $\mathbf{y}_k$. However, recall that

$$\mathbf{y}_k = \pi_k(\mathbf{y}_{k-1} + \mathbf{z}_k),$$

by the design of the protocol. Whatever $\mathbf{y}_{k-1}$ the adversary may choose, the information available in $\mathbf{y}_k$ is no more than π_k and $\mathbf{z}_k$. Note that in online phase, before entering the checking phase, the only chance for the adversary to learn any information is to choose possibly arbitrary $\mathbf{y}_k$ and observe the output of honest party P_k.

Recall in Lemma 4, where the adversary learns precisely $\mathbf{x}$ and all the π_i and $\mathbf{z}_i$ for $i \in [n]$. The lemma has stated that, this is still insufficient to deduce any information about β due to the ignorance of $\mathbf{r}'_i$. □

Remark. In above theorem we state that even if the adversary learns both π_i and $\mathbf{z}_i$ of honest P_i, it cannot learn anything about β before entering the checking phase. The fact that the adversary (in fact) cannot learn anything about π_i of honest P_i is later shown by constructing UC simulator.

B.4 Enforcement of Honest Behaviors in Online Phase

The enforcement of honest behaviors of the adversary means that the adversary cannot send any wrong messages (different from what is specified by protocol) to any honest party, otherwise the protocol aborts with overwhelming probability in the first check related to that message. To prove this, we first need another theorem regarding the security of the correlation test protocol.

Theorem 3. *In the sequential invocations of* PartialVerify*, if the invocation does not yet abort for P_{i+1}, then with probability at least $1-\frac{1}{2^\kappa}$ P_{i+1} has broadcast correct w_1 and w_2, and the MAC key β is safe.*

Proof. Let's prove by induction. The basic case is the invocation of verification protocol for P_2.

Without loss of generality, consider a malicious P_2. By Theorem 2, before entering the checking phase, the adversary does not have any knowledge about β. By examining the protocol, it is clear that the adversary remains ignorant about β upon broadcasting w_1 and w_2, since c_2 opened to it is independent of β. Also note that, if P_2 is malicious, it must know correct w_1 and w_2, regardless of

whether P_1 is honest or not. However, for one who knows (w_1, w_2) and is ignorant to β, forging valid $(\hat{w}_1, \hat{w}_2)$ different from (w_1, w_2) is equivalent to guessing β, since

$$\beta = \frac{\hat{w}_2 - w_2}{\hat{w}_1 - w_1}.$$

Therefore, if the protocol exits normally, then with probability at least $1 - \frac{1}{2^\kappa}$ that correct w_1 and w_2 is broadcast.

By a similar argument and induction, as long as the protocol does not abort for P_i, the adversary remains ignorant of β. And due to its ignorance, the w_1 and w_2 broadcast so far must all be correct, except with negligible probability. □

The next theorem enforces the correctness of all $\mathbf{y}_i$, hence an honest online phase.

Theorem 4. *If the adversary $\mathcal{A}$ sends any wrong $\mathbf{y}_i$ that is different from what is specified by the protocol to honest P_{i+1}, then the protocol will abort with probability at least $1 - \frac{m}{2^\kappa}$.*

Further, if $\mathbf{y}_n$ broadcast by P_n is different from what is specified by the protocol, the protocol aborts with probability at least $1 - \frac{1}{2^\kappa}$.

Hence, if the tests pass, with overwhelming probability that all $\mathbf{y}_i$ sent to honest P_{i+1} are accordant with the protocol, as well as the $\mathbf{y}_n$ broadcast by P_n.

Proof. In the online phase $\text{Shuffle}_{\text{on}}$, it is required that the test

$$\text{PartialVerify}(P_{i+1}, \mathbf{y}_i^1, \mathbf{y}_i^2, \langle\beta\rangle, \langle c_{i+1}\rangle, \langle d_{i+1}\rangle)$$

passes, for all honest P_{i+1}. This can be divided into two cases:

1. $\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}'_i)$, i.e. the message satisfies correct correlation, and the protocol exits normally.
2. $\beta\mathbf{y}_i^1 \neq \mathbf{y}_i^2 + \pi_i(\mathbf{r}'_i)$, but the protocol does not abort.

By Lemma 3, the second case happens with probability at most $(m-1)/2^\kappa$. Also, it is required that

$$\text{Verify}(\mathbf{y}_n^1, \mathbf{y}_n^2, \langle\beta\rangle, \langle\pi_n(\mathbf{r}'_n)\rangle)$$

passes, which by Lemma 2 means precisely $\beta\mathbf{y}_n^1 = \mathbf{y}_n^2 + \pi_n(\mathbf{r}'_n)$. Hence, it remains to prove that the adversary cannot forge wrong message with correct correlation, except with probability at most $1/2^\kappa$.

Assume for contradiction that the adversary somehow forges different $\hat{\mathbf{y}}_i \neq \mathbf{y}_i$ for some $i \in S \subset [n]$, such that P_{i+1} is honest (if P_{i+1} exists) and

$$\beta\hat{\mathbf{y}}_i^1 = \hat{\mathbf{y}}_i^2 + \pi_i(\mathbf{r}'_i).$$

Let's fix i as the least element in S, so P_{i+1} is the first "victim". Since there are no wrong messages sent to honest party before i, (malicious) P_i must know correct $\mathbf{y}_i$ to send, which satisfies

$$\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \pi_i(\mathbf{r}'_i).$$

By subtracting two equations, the adversary obtains

$$\beta(\mathbf{y}_i^1 - \hat{\mathbf{y}}_i^1) = \mathbf{y}_i^2 - \hat{\mathbf{y}}_i^2.$$

Since $\mathbf{y}_i \neq \hat{\mathbf{y}}_i$, it has to be the case that

$$\mathbf{y}_i^1 \neq \hat{\mathbf{y}}_i^1.$$

Hence, the adversary will be able to find some index $j \in [m]$, such that

$$\beta(\mathbf{y}_i^1(j) - \hat{\mathbf{y}}_i^1(j)) = \mathbf{y}_i^2(j) - \hat{\mathbf{y}}_i^2(j) \text{ s.t. } \mathbf{y}_i^1(j) - \hat{\mathbf{y}}_i^1(j) \neq 0,$$

where $\mathbf{y}_i^1(j)$ denotes the j-th entry of the vector. This allows the adversary to solve β explicitly as

$$\beta = \frac{\mathbf{y}_i^2(j) - \hat{\mathbf{y}}_i^2(j)}{\mathbf{y}_i^1(j) - \hat{\mathbf{y}}_i^1(j)}.$$

Note that the action of sending $\hat{\mathbf{y}}_i$ to honest party happens before the checking phase. By Theorem 2, upon sending $\hat{\mathbf{y}}_i$, the adversary could not learn β from its view, whereas here it has already solved it explicitly. Thus, the assumption that "the adversary could somehow forge some different $\hat{\mathbf{y}}_i$ that satisfies the correlation" will not happen with probability better than the same probability of guessing β correctly, which is at most $1/2^\kappa$.

Thus, if the protocol does not abort after all the checks, with overwhelming probability $p \geq 1 - \frac{m}{2^\kappa}$ all $\mathbf{y}_i$ sent to honest parties are correct, in the sense that it is precisely the one produced by honestly following the protocol. □

Hence, we conclude in the following corollary that the honest behavior of the adversary is enforced in both offline and online phase.

Corollary 1. *The shuffle protocol enforces honest behaviors in both offline phase and online phase, in the sense that any misbehavior deviating from the protocol will cause abort with overwhelming probability.*

In particular, in the online phase, this means that sending any incorrect messages (e.g. $\mathbf{y}_i$, w_1, w_2*) to honest parties will lead to immediate abort in the first subsequent check related to that message, with overwhelming probability.*

Proof. This is obtained by combining Theorem 3 and 4. □

B.5 UC Simulator

We are now able to prove the UC security of the shuffle protocol. Firstly we formally define the two games in real and ideal world, and the security of the protocol depends upon.

Definition 10 (Real and Ideal Worlds). *In the real execution of the shuffle protocol, at the start of the game, the environment* $\mathcal{E}$ *sends* $\mathbf{x}$ *to* $\mathcal{F}_{\text{MPC}}$*, who stores it locally as* $\langle \mathbf{x} \rangle$*.* $\mathcal{E}$ *also sends* π_i *to honest* P_i*, as* P_i*'s input for protocol* Shuffle*.*

Then the adversary $\mathcal{A}$ corrupting $T \subsetneq [n]$ starts executing protocol Shuffle *with honest parties, while receiving commands from and sending information to $\mathcal{E}$. At the end of the protocol, if the parties do not abort, $\mathcal{F}_{\text{MPC}}$ sends its output of* Shuffle *to $\mathcal{E}$, which is (by design) $\pi(\mathbf{x})$. Denote this process by $\mathcal{E} \leftrightarrows \Pi_{\bar{T},\mathcal{A}}$.*

In the ideal world, $\mathcal{A}$ is replaced by simulator $\mathcal{S}$, which plays the role of all corrupted parties. $\mathcal{F}_{\text{MPC}}$ is equipped with a new command Π_{shuffle}, which requires parties to agree on $\langle \mathbf{x} \rangle$ to be shuffled, and takes as input π_i from P_i. The modified $\mathcal{F}_{\text{MPC}}$ is hence denoted as $\mathcal{F}_{\text{shuffle}}$. At the start of the game, $\mathcal{E}$ sends $\mathbf{x}$ to the ideal functionality $\mathcal{F}_{\text{shuffle}}$. $\mathcal{E}$ also sends π_i for honest party P_i to $\mathcal{F}_{\text{shuffle}}$, as part of the inputs for Π_{shuffle}. Then $\mathcal{E}$ interacts with $\mathcal{S}$ as if with $\mathcal{A}$, controlling what are sent from corrupted parties and receiving what are received by corrupted parties. If the game does not abort, $\mathcal{E}$ receives from $\mathcal{F}_{\text{shuffle}}$ the output vector. Denote this process by $\mathcal{E} \leftrightarrows \mathcal{F}_{\bar{T},\mathcal{S}}$, where $\mathcal{F}$ means $\mathcal{F}_{\text{shuffle}}$.

At the end of each game, $\mathcal{E}$ outputs a single bit, representing its judgement on whether this is the real world game.

The protocol Shuffle *UC-securely implements Π_{shuffle} in $\mathcal{F}_{\text{MPC}}$-hybrid model, if for every $\mathcal{E}$ and $\mathcal{A}$, there is a probabilistic polynomial time simulator $\mathcal{S}$, such that*

$$\left|\Pr\left[1 \leftarrow (\mathcal{E} \leftrightarrows \Pi_{\bar{T},\mathcal{A}})\right] - \Pr\left[1 \leftarrow (\mathcal{E} \leftrightarrows \mathcal{F}_{\bar{T},\mathcal{S}})\right]\right| < \epsilon = O(\frac{m}{2^\kappa}),$$

for some statistical secure parameter κ fixed a priori and length m of $\mathbf{x}$.

Remark. For the protocol to be UC secure, it is necessary to let $\mathcal{E}$ choose the $\mathbf{x}$ to be shuffled, and then obtain the shuffled result. Since in practice, it could be the case that $\mathbf{x}$ comes exactly from a Π_{input} command precedes the shuffle, and will be opened directly after. The security relies on the fact that the simulator $\mathcal{S}$, while being completely ignorant of $\mathbf{x}$ and the π_i of the honest party, can still create the view of the adversary $\mathcal{A}$ in real world that is indistinguishable to any $\mathcal{E}$. When $\mathcal{F}_{\text{shuffle}}$ is replaced by real implementation of multiparty computation, the universal composition theorem will guarantee the overall security.

Theorem 5. *Assume the protocols are working in a field of size no smaller than 2^κ, where κ is a statistical security parameter chosen arbitrarily.*

Then the protocol Shuffle *is universally composable secure in $\mathcal{F}_{\text{MPC}}$-hybrid model, in the sense that for any environment $\mathcal{E}$ and adversary $\mathcal{A}$, there exists probabilistic polynomial time simulator $\mathcal{S}$ such that*

$$\left|\Pr\left[1 \leftarrow (\mathcal{E} \leftrightarrows \Pi_{\bar{T},\mathcal{A}})\right] - \Pr\left[1 \leftarrow (\mathcal{E} \leftrightarrows \mathcal{F}_{\bar{T},\mathcal{S}})\right]\right| < \epsilon = O(\frac{m}{2^\kappa}).$$

Proof. We construct a simulator for dummy adversary $\mathcal{A}$, who sends $\mathcal{E}$ what it receives and lets $\mathcal{E}$ carry out computation and decide what to send.

At the beginning of the protocol, $\mathcal{E}$ decides the input $\mathbf{x}$ and the input π_i' for honest party P_i, and sends all of them to the ideal functionality $\mathcal{F}_{\text{shuffle}}$.

The simulator $\mathcal{S}$ works as follows. In the offline phase of the shuffle, $\mathcal{S}$ first draws β uniformly from $\mathbb{F}$. It interacts with $\mathcal{E}$ and obtains the purported input π_i' for corrupted party P_i. The simulator can obtain π_i', because it is the input of corrupted party P_i for Π_{perm}, which means that $\mathcal{E}$ needs to send π_i' explicitly

to $\mathcal{S}$ upon calling Π_{perm}. Then $\mathcal{S}$ simulates the output of offline phase with β and π_i, acting as if each honest P_i has chosen its permutation to be identical permutation. It draws uniformly random vectors $\mathbf{r}_i$, $\mathbf{r}'_i$ and c_i, and computes $\mathbf{z}_i$ and d_i as is specified by the protocol. To be specific, $\mathcal{S}$ hence generates and stores locally the entire shuffle correlation (in plaintext)

$$\text{cor} = \left\{ \begin{array}{cccccccc} \langle\beta\rangle & \langle\mathbf{r}_1\rangle & \langle\beta\mathbf{r}_1\rangle & \langle\mathbf{r}'_1\rangle & \langle\pi_n(\mathbf{r}_n)\rangle & & & \\ \pi_1 & \pi_2 & \pi_3 & \pi_4 & \pi_5 & \cdots & \pi_n & \\ \langle\pi_1(\mathbf{r}'_1)\rangle & \langle\pi_2(\mathbf{r}'_2)\rangle & \langle\pi_3(\mathbf{r}'_3)\rangle & \langle\pi_4(\mathbf{r}'_4)\rangle & \langle\pi_5(\mathbf{r}'_4)\rangle & \cdots & \langle\pi_n(\mathbf{r}'_n)\rangle & \\ & \langle c_2\rangle & \langle c_3\rangle & \langle c_4\rangle & \langle c_5\rangle & ... & \langle c_n\rangle & \\ & \langle d_2\rangle & \langle d_3\rangle & \langle d_4\rangle & \langle d_5\rangle & ... & \langle d_n\rangle & \\ & \mathbf{z}_2 & \mathbf{z}_3 & \mathbf{z}_4 & \mathbf{z}_5 & \cdots & \mathbf{z}_n & \end{array} \right\}$$

where

$$\pi_i = \begin{cases} \text{identical permutation} & \text{if } P_i \text{ is honest,} \\ \pi'_i & \text{otherwise.} \end{cases}$$

When corrupted party P_i needs to receive $\mathbf{z}_i$, $\mathcal{S}$ simply sends the $\mathbf{z}_i$ it computes internally to $\mathcal{E}$. $\mathcal{S}$ also sends all π'_i for corrupted parties to $\mathcal{F}_{\text{shuffle}}$. The simulation up to here is perfect, since $\mathbf{z}_i$ is uniformly random in both real and ideal world games.

Then in the online phase, $\mathcal{S}$ simulates the message of each P_i. If P_1 is honest, $\mathcal{S}$ sends on behalf of P_1

$$\mathbf{y}_1 \coloneqq \pi_1(\mathbf{z}_1) = \mathbf{z}_1 = (\mathbf{r}_1, -\beta\mathbf{r}_1 - \mathbf{r}'_1),$$

as if $\mathbf{x} = \mathbf{0}$. If P_1 is dishonest, $\mathcal{S}$ will inform $\mathcal{E}$ of P_1 receiving the above $\mathbf{z}_1$, leaving $\mathcal{E}$ to decide $\mathbf{y}_1$. Then for each P_i, if P_i is honest, since $\mathcal{S}$ knows the entire shuffle correlation, $\mathcal{S}$ acts honestly and sends on P_i's behalf

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

to P_{i+1}. For dishonest P_i, $\mathcal{S}$ leaves $\mathcal{E}$ to decide the message to send.

After P_n broadcasts $\mathbf{y}_n$, for each $i \geq 2$, $\mathcal{S}$ simulates PartialVerify. $\mathcal{S}$ first opens c_i to P_i. If P_i is honest, $\mathcal{S}$ simulates it honestly by broadcasting

$$w_b = \sum_{j=1}^{m} c_{i+1}^{j-1} \mathbf{y}_i^b(j).$$

If P_i is corrupted, $\mathcal{E}$ decides w_b to be broadcast. $\mathcal{S}$ then opens the resulted d, and inform $\mathcal{E}$ of abortion if $d \neq 0$.

If above process does not yet abort, $\mathcal{S}$ continues to simulate Verify. If the protocol ends without abort, and $\mathcal{E}$ does not demand $\mathcal{S}$ to abort, $\mathcal{S}$ requires $\mathcal{F}_{\text{shuffle}}$ to continue. $\mathcal{F}_{\text{shuffle}}$ then proceeds to send

$$\pi(\mathbf{x}) \coloneqq \pi'_n \circ \pi'_{n-1} \circ \cdots \circ \pi'_1(\mathbf{x})$$

to $\mathcal{E}$, where recall that π'_i of honest P_i is sent earlier from $\mathcal{E}$ to $\mathcal{F}_{\text{shuffle}}$, and π'_i of corrupted P_i is sent earlier from $\mathcal{S}$.

To argue that this simulator indeed generates a view statistically indistinguishable from the view of the adversary in the real world game, note that by Corollary 1, if $\mathcal{E}$ misbehaves (by sending wrong messages to any honest party), the protocol aborts with overwhelming probability $p \geq 1 - \frac{m}{2^\kappa}$ in the next (related) check in both real and ideal world. So if $\mathcal{E}$ misbehaves, the view will be statistically indistinguishable, since if the protocol aborts, all values appear so far are uniform random field elements, and in both worlds, the protocol aborts with overwhelming probability.

If $\mathcal{E}$ acts honestly and the protocol finishes without abort, $\mathcal{E}$ will receive permuted $\pi(\mathbf{x})$. Tracing back, due to the ignorance of $\mathbf{r}_i$ and $\mathbf{r}'_i$, all values in its view are "reasonable". To be specific, for each possible assignment to β and $\{\pi_i\}_{i\in\bar{I}}$, $\mathcal{E}$ can find exactly one set of assignment to all $\mathbf{r}_i$ and $\mathbf{r}'_i$, so that all $\mathbf{z}_i$ and $\mathbf{y}_i$ are consistent with its view and the result is indeed $\pi(\mathbf{x})$. This is the same as the semi-honest case, in proof of Theorem 1.

□

C Case Studies

In this section, we present several concrete protocols generated by instantiating our constructions with existing MPC permutation protocol of [14], [15] and [16], respectively. The first construction works only for additive secret sharing, and is considerably efficient, due to utilizing possibly currently most efficient MPC permutation protocol. The second construction works for arbitrary MPC framework supporting MPC multiplication, with fairly simple implementation. The third construction may be adapted to threshold secret sharing scheme, but it requires ZK proofs in its offline phase to achieve malicious security.

However, despite all these differences, all three constructions are actively secure and have linear online complexity, due to our malicious-secure shuffle correlation.

C.1 Shuffle Protocol for Additive Secret Sharing

The construction of Song et al. [14] is based on SPDZ framework [22], where a variable stored at MPC is additively secret shared among all parties, along with an additional MAC. That is, the statement "a variable a is secret shared as $\langle a\rangle$" means that each party P_i holds locally a tuple $(a_i, \gamma_i(a))$ such that

$$\sum_{i=1}^{n} a_i = a, \ \sum_{i=1}^{n} \gamma_i(a) = \alpha a,$$

where α is a secret global key that is used to guarantee the integrity of opening. Song et al. [14] proposes a two-party (sender and receiver) permutation protocol, which permutes a secret shared array with permutation specified by one of the party (i.e. the receiver).

Basically, this two-party permutation protocol is built from repeating basic CGP protocol [17] for B times with additional correctness checks, while B is a

parameter required for applying cut-and-choose technique. If the receiver has chosen permutation $\pi_1, ..., \pi_B$ respectively in these sessions, the overall effect is that the array is permuted by a single permutation π:

$$\pi = \pi_B \circ \pi_{B-1} \circ \cdots \circ \pi_1.$$

This two-party permutation protocol is then extended to the case of n-party by letting a fixed party P_i act as the receiver in all $(n-1) \times B$ sessions, which can be considered as permuting the shares of different party with same B permutations. Of course, extra works are done in Song et al. [14] to prevent P_i from acting inconsistently in session with different party, e.g. choosing different B permutations when the sender is different. The protocol then checks if each session of the CGP protocol finishes correctly, to guarantee that no misbehavior happens so far.

To conclude, the work of Song et al. [14] provides us with exactly the functionality

$$(\langle \pi(\mathbf{x}_1) \rangle, ..., , \langle \pi(\mathbf{x}_t) \rangle) \leftarrow \Pi_{\text{perm}}(P_i : \pi, \langle \mathbf{x}_1 \rangle, ..., , \langle \mathbf{x}_t \rangle),$$

with an overhead of $O(Bnmt \log m)$ communication and computation complexity. By instantiating with the SPDZ framework of [19] and fitting above permutation protocol into Algorithm 7, we obtain a shuffle protocol with $O(Bn^2 m \log m)$ offline communication and computation complexity, whose online complexity is linear.

One intricate detail regarding implementation is that at the end of the shuffle protocol, it must be guaranteed that the array is correctly shuffled. This is due to the usage of MPC shuffle protocol, which is often followed by partial or complete information disclosure, e.g. the Clarion anonymous communication system designed by [13] opens all values after the shuffle. However, SPDZ itself does not guarantee safe opening, i.e. before the MAC check passes, all opened value could be wrong. This means that Π_{open} does not guarantee a safe open (which we have assumed throughout), which means that all values during the shuffle protocol, including those used in verification, could be wrong, and the correctness is not guaranteed even if the protocol doesn't abort. Luckily, SPDZ framework supports immediate MAC check, which checks if all previously opened values are correct, with only $O(n)$ communication and computation complexity. Therefore, in practice, the shuffle protocol must be followed by such an immediate MAC check, which in turn makes all previous Π_{open} safe. We remark that this is also the approach followed by Song et al. [14], as their shuffle protocol ends with precisely the MAC check of SPDZ.

We note that, this results in the first malicious secure shuffle protocol for additive secret sharing with linear online complexities.

C.2 Shuffle Protocol for Shamir Secret Sharing

The construction of permutation protocol in [15] works for arbitrary MPC framework supporting MPC multiplication. Assuming MPC multiplication can be done within $O(n)$ communication and computation, its implementation of Π_{perm} requires $O(nm \log m)$ communication and computation in total.

The permutation protocol developed in [15] utilizes a specific permutation network, which is a switching network of size $O(m \log m)$ and of depth $O(\log m)$. Such a network consists of $O(\log m)$ layers of switches, each switch randomly swapping two entries of the vector depending on an extra control bit. As a permutation network is capable of representing any permutation by $O(m \log m)$ switches, this allows parties to represent a permutation by $O(m \log m)$ bits.

Now suppose P_i wants to permute $\langle \mathbf{x} \rangle$ by permutation π. P_i first represents π by permutation network, which results in $O(m \log m)$ control bits, each corresponding to the control bit of one switch. P_i then shares these bits to parties, who then check together whether the shared values are bits. This check can be realized by MPC multiplication, by noting that $\langle b \rangle$ is a bit if and only if $b \cdot (1-b)$ is zero. By following the permutation network and using MPC multiplication to simulate each switch, the parties are now able to permute any m-long vector by π.

Such an implementation matches exactly our requirements for Π_{perm}. By instantiating Π_{perm} in Algorithm 7 with such implementation, we immediately obtain a shuffle protocol for arbitrary MPC framework supporting MPC multiplication. The offline phase consists of $O(nm \log m)$ many multiplications, which results in $O(n^2 m \log m)$ communication, computation and $O(\log m)$ rounds, assuming the communication and computation overhead of each multiplication are both linear. By instantiating with the Shamir secret sharing scheme in [20], we obtain a concrete construction that meets above complexities. The above implementation can also be instantiated with SPDZ framework of [19], with $O(n^2 m \log m)$ offline complexities that is asymptotically better than the above. Nevertheless, since the primitives used in [14] might be cheaper than MPC multiplication, which construction is more concretely efficient may also depend on concrete implementation.

We note that, this results in the first shuffle protocols for Shamir secret sharing with linear online complexities, for both semi-honest and malicious security.

C.3 Shuffle Protocol Instantiated by Protocol of [18]

The construction of Laur et al. [16] works for threshold secret sharing, with possibly malicious adversary. By its design, each party belongs to several groups, and each group of parties will agree on a common permutation and permute the array once. The size of each group is large enough, so that it is possible to reconstruct all secret with the shares of the group member, which allows the group to permute the array trivially by each member permuting the shares locally and re-distributing their shares. For $(n/2)$-threshold secret sharing, there will be asymptotically $O(2^n/\sqrt{n})$ many groups, and by design at least one of the groups consists of only honest parties. Hence, since this all-honest group permutes the array with a uniform random permutation known only to its group members, the array is shuffled with a permutation known to no one. The advantage of this construction is that, when the number of parties are small and all parties are semi-honest, it does not utilize any public key primitives and is extremely fast. Also, most of the other MPC shuffle protocols works only with additive secret

sharing (e.g. [17][18][13][14]), while the construction of [16] works with threshold secret sharing, e.g. Shamir secret sharing.

It is possible to extend our construction, so that the shuffle protocol may be instantiated by the protocol of Laur et al. [16]. For clarity, suppose we are working in semi-honest case. Suppose we have t such groups

$$\mathcal{G} = \{G_1, G_2, ..., G_t\},$$

by the design of [16], with $G_i \subsetneq \{P_i\}_{i\in[n]}$. All that needs to be done is to replace the Π_{perm} functionality by

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

where $G \in \mathcal{G}$, i.e. π is known only to members of G. The rest follows naturally by viewing each G_i as an individual, i.e. generating and permuting random vectors with each permutation π_i known only to members of G_i, opening $\mathbf{z}_i$ to only members of G_i, etc. The online phase hence consists of $O(2^n/\sqrt{n})$ rounds and has $O(2^n m/\sqrt{n})$ communication complexity. Note that instead of repeating this same process on all members of G_i, each group G_i can elect a party in the group as its agent, who will permute the masked vector $\mathbf{y}_{i-1}$, add the vector $\mathbf{z}_i$ and send $\mathbf{y}_i$ to agent of the next group G_{i+1}.

Further, to reduce the online round complexity to $O(n)$ and online communication to $O(nm)$, we can sort the groups in a particular order, such that each P_i is the agent of only a consecutive sequence of groups. For example, if

$$\mathcal{G} = (\{P_1, P_2\}, \{P_2, P_3\}, \{P_1, P_3\}),$$

we can sort it as

$$\mathcal{G} = (\{P_1, P_2\}, \{P_1, P_3\}, \{P_2, P_3\}).$$

Now, by letting P_1 be the agent of G_1 and G_2, P_1 will be responsible for sending both $\mathbf{y}_1$ and $\mathbf{y}_2$ in the online phase, and sending $\mathbf{y}_2$ to the agent of G_3. It is clear that P_1 can carry out the computation locally, and sends only $\mathbf{y}_2$ to the agent of G_3, which consumes only 1 round instead of 2. By letting a consecutive sequence to have a common agent, this results in an $O(n)$-round online phase with $O(nm)$ communication.

The online computation can also be made $O(nm)$. Since agent P_i knows all the permutations and masks it needs in online phase, it can carry out the data independent part of the computation in offline phase. To be more specific, suppose it needs to compute

$$\begin{aligned}
\mathbf{y}_1 &= \pi_1(\mathbf{x} + \mathbf{z}_1),\\
\mathbf{y}_2 &= \pi_2(\mathbf{y}_1 + \mathbf{z}_2),\\
\mathbf{y}_3 &= \pi_3(\mathbf{y}_2 + \mathbf{z}_3),\\
&\vdots\\
\mathbf{y}_k &= \pi_k(\mathbf{y}_{k-1} + \mathbf{z}_k).
\end{aligned}$$

This is simplified to compute

$$\begin{aligned}\mathbf{y}_k = \pi_k \circ \pi_{k-1} \circ \cdots \circ \pi_1(\mathbf{y}_1) \\ + \pi_k \circ \pi_{k-1} \circ \cdots \circ \pi_1(\mathbf{z}_1) \\ + \pi_k \circ \pi_{k-1} \circ \cdots \circ \pi_2(\mathbf{z}_2) \\ + \pi_k \circ \pi_{k-1} \circ \cdots \circ \pi_3(\mathbf{z}_3) \\ \vdots \\ + \pi_k(\mathbf{z}_k).\end{aligned}$$

It is clear that the permutations of $\mathbf{z}_i$ can be computed and sumed up in offline phase. Also, the party can first compute the permutation $\pi_k \circ \cdots \circ \pi_1$, and hence in the online phase it needs to only perform one permutation for $\mathbf{y}_1$ and add it with the pre-processed sum.

This can be also extended to malicious security, with the same technique used earlier in Section 6 and Section 7. Note that we only need to check if the final result of agent P_i is correct, since all other computations are done internal P_i. Hence, the overall online computation complexity remains $O(nm)$. Note that as the construction of [16] utilizes ZK proof for malicious security, it is likely that the construction instantiated with [14] or [15] would be more efficient in practice. Nevertheless, if one wishes to build their MPC application with Shamir secret sharing or replicated secret sharing (instead of additive secret sharing), the construction of [14] will not be available, and instantiating shuffle protocol with the construction of [16] might be a good choice.

D Discussion

D.1 Security of Π_{open}

One thing concerning the practical use is that, in most advanced MPC frameworks, the integrity check of Π_{open} may not be immediate. For example, by original design of SPDZ [22], all opened values should be independent random values before the final output phase, which is preceded by a big batched check Π_{check}. This seems preferable, and it is tempting to batch all integrity checks, both Π_{open} and Verify, into one big linearity check.

However, in real world application of shuffle, it is usually the case that the shuffled values will be opened immediately without masks, e.g. the anonymous communication service designed in [13]. Or more generally, the information of the underlying element may be revealed, e.g. the shuffle-then-sort paradigm in [7], which opens the result of comparison between elements directly after the shuffle operation. After all, one major reason for turning to a shuffle protocol is to reveal some information that is previously related to the "memory address". Hence, it is clear that the shuffle protocol must guarantee that if the protocol finishes without abortion, then the array is indeed a correctly shuffled, with the permutation uniform in the adversary's view.

Hence, each shuffle protocol must be followed by immediate integrity check regarding previously opened values, and the check of $\mathbf{y}_i^1$ and $\mathbf{y}_i^2$ must be carried out before any information regarding the elements is to be opened. So it's probably the safest to do the batched check immediately. This is also the strategy adapted by Song et al. [14] to implement their malicious secure shuffle protocol.

D.2 Approximated Length of Vector

In the above protocol, the offline phase $\text{Shuffle}_{\text{off}}$ is assumed to have m as the exact length of later input vector. In reality, it is very likely that m will not be exact, since the input has yet come. Below, we give a simple extension so that the protocol requires only m as an upper bound of length.

First note that, by a trivial extension, the shuffle protocol designed in this paper can be used to shuffle vectors instead of field elements, as long as the underlying Π_{perm} supports so. This trivial extension simply replaces all "vectors" by "matrices", and everything works fine.

Hence, the parties could first pad the vector with dummy elements, so that the vector is of length m. The parties then extend each element to a vector of length 2, with second entry a public constant 0/1 indicating if this entry is dummy. Then the parties could run the shuffle protocol for vectors of length 2, and later open all the second entries and discard the dummy ones.

Note that this padding trick could also be applied to shuffling entries of a vector, where only upper bound of length is known a priori.