Free Linear Online Phase for Secure Multiparty Shuffle ⋆

Jiacheng Gao[1], Yuan Zhang[1] ✉, Sheng Zhong[1], and Changyu Dong[2]

[1] State Key Laboratory of Novel Software Technology
Nanjing University
jcgao@smail.nju.edu.cn, zhangyuan@nju.edu.cn, zhongsheng@nju.edu.cn
[2] Guangzhou University
changyu.dong@gmail.com

Abstract. Shuffling is a fundamental operation in secure multiparty computation (MPC), yet existing maliciously secure protocols for shuffling m elements among n parties require at least n^2m online communication and computation, compared with $O(nm)$ in the semi-honest additive-sharing setting.

We formalize and extend the *permute-in-turn paradigm*, which constructs MPC shuffles from permutation protocols, capturing the methodology adopted in most prior works. Our generic transformations achieve semi-honest or malicious security with only $O(C_{\mathrm{BC}} + nm)$ online communication and computation, where C_{BC} is the cost of one parallel broadcast (allowing unanimous abort). Our transformations are *for free* in two senses: they yield linear online phases without exceeding the overall asymptotic cost of applying n MPC permutations, while *freeing* the online phase of the shuffle protocol from dependence on the MPC permutation protocol. This decoupling allows future work to optimize permutation protocols independently while benefiting all MPC shuffle protocols that follow the paradigm.

Instantiated with additive and Shamir secret sharing, our transformations yield the first maliciously secure shuffles with $O(C_{\mathrm{BC}} + nm)$ online cost, improving previous bounds of $O(Bn^2m)$ and $O(n^2m \log m)$, respectively, where B is a cut-and-choose parameter. We formally prove semi-honest and universally composable (UC) security for the corresponding transformations. For $m = 2^{12}$ and $n = 15$, our prototype achieves over $60\times$ lower online runtime and $120\times$ lower online communication than prior work, with only a $1.3\times$ increase in total runtime.

Keywords: multiparty computation · shuffle · random correlation

1 Introduction

Secure multiparty computation (MPC) has numerous real-world applications. In an MPC scheme, multiple parties jointly compute a function over their secret

⋆ A full version is available at the Cryptology ePrint Archive: https://eprint.iacr.org/2024/1936.

inputs, while keeping each party's input confidential from the others. This is particularly relevant in practical scenarios such as joint database queries [6, 13, 44], federated learning [29, 39, 43], and other collaborative data analytics.

A fundamental yet costly primitive in MPC is the shuffle operation, which allows the parties to jointly permute a secret-shared data array by a random permutation unknown to all parties. Shuffling serves as a core building block for more complex protocols, including MPC sorting [25,26,36], MPC oblivious RAM (ORAM) [33], and anonymous communication systems [3, 20]. Recent protocols also make shuffling a central component of privacy-preserving graph analysis, including large-scale graph computation and centrality evaluation [4, 8, 34] and heavy-hitter identification [5]. Despite its importance, MPC shuffling remains a performance bottleneck in large-scale deployments [42].

While most existing works focus on optimizing the overall communication and computation of MPC shuffle protocols, when it comes to real-world efficiency, it is usually the online phase (i.e., the data-dependent phase) that forms the bottleneck in MPC applications. However, despite these efforts, the online communication and computation of current maliciously secure MPC shuffle protocols remain over n^2m, for n parties shuffling m items [33,37,42]. For example, in an MPC-based anonymous communication system using the maliciously secure shuffle of [42], $n = 6$ servers shuffling $m = 2^{12}$ client messages would spend over 35 seconds in the online phase alone according to our experiments (cf. Section 7). This directly becomes user-visible latency.

We observe that nearly all existing MPC shuffle protocols share a common structure, which we call the *permute-in-turn paradigm*. In order to shuffle m items among n parties by a secret and uniformly random permutation π, each party P_i locally chooses a private random permutation π_i. After securely applying each permutation π_i to the data in sequence, the resulting permutation applied to the data is $\pi = \pi_n \circ \pi_{n-1} \circ \cdots \circ \pi_1$, which is uniform and hidden from all parties. To securely apply each π_i to the data, the parties typically rely on an MPC permutation protocol Π_{perm} as a primitive. As will be discussed in more detail in Section 2, this simple idea and the primitive Π_{perm} underlie most existing MPC shuffle protocols [10, 12, 20, 27, 33, 35, 37, 40, 42].

Although the paradigm is straightforward and already widely adopted, the n-party MPC shuffle protocols following this paradigm typically incur over n^2m communication and computation in their online (data-dependent) phase [10, 27, 33, 35, 37, 40, 42].[3] This overhead stems from invoking Π_{perm} n times in the online phase, each invocation costing $O(nm)$. The only exceptions are the shuffle protocols by Eskandarian and Boneh [20] and Chen et al. [12], which avoid calling Π_{perm} in their online phases and achieve $O(nm)$ online complexity with a special type of correlated randomness called *shuffle correlation*. However, the constructions by [20] and [12] work only for semi-honest additive secret sharing.

[3] Note that the protocols in [10,27,40] are two-party protocols, where $n = 2$ is constant. Nevertheless, a straightforward generalization of these two-party protocols to the n-party case would also incur over n^2m online complexity, by performing $O(n^2)$ pairwise two-party shuffles, as is the case for [35], which generalizes [10] to n-party.

It thus remains an open problem to construct MPC shuffle protocols for malicious security with linear online communication and computation, and to generalize the semi-honest results by [12, 20] to other secret sharing schemes.

In this paper, we resolve this problem by presenting novel constructions that transform any semi-honest or maliciously secure MPC permutation protocol into an MPC shuffle protocol with matching security and with only $O(C_{\mathrm{BC}} + nm)$ online communication and computation, where C_{BC} is the cost for one parallel broadcast of constant-length message. The bound is linear in the shuffled data when $C_{\mathrm{BC}} = O(nm)$. Crucially, our constructions treat the permutation protocol as a black box, ensuring broad applicability across different MPC frameworks and secret sharing schemes. As suggested by the title, our transformation is *for free* in two senses: it preserves the overall asymptotic complexity[4] necessary for performing n MPC permutations and yields linear online complexity, and it *frees* the online phase from depending on the specific implementation of Π_{perm}, enabling future work to optimize permutation protocols independently.

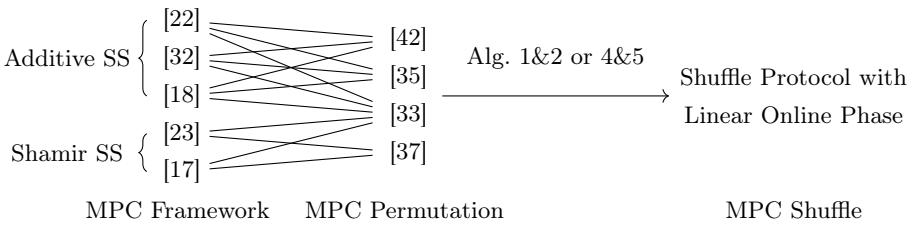

Fig. 1. Illustration of our transformation: any compatible pair consisting of an MPC framework and a permutation protocol can be transformed into an MPC shuffle protocol with linear online complexity.

Besides preserving asymptotic communication and computation complexity, our transformations also enjoy broad applicability. As illustrated in Figure 1, any compatible pair consisting of an MPC framework and a permutation protocol can be transformed into a shuffle protocol with linear online complexity, while inheriting the corresponding security level and offline complexity.

Our contributions are summarized as follows.

1. We formalize the permute-in-turn paradigm and propose generic transformations from MPC permutation protocols to MPC shuffle protocols with linear online complexity for both semi-honest and malicious security. Since the permute-in-turn paradigm underlies most existing MPC shuffle protocols, our transformations are widely applicable.

[4] Our maliciously secure shuffle doubles the amount of data input to the permutation protocol, e.g. changing from permuting m elements to permuting m length-2 vectors. Given that the offline phase can be precomputed, we believe this increase is moderate and well justified by the significant online efficiency gain.

2. Instantiating our constructions with state-of-the-art permutation protocols, we derive the first maliciously secure shuffles with $O(C_{\mathrm{BC}}+nm)$ online complexity for both additive and Shamir secret sharing. This improves prior bounds of $O(Bn^2m)$ [42] and $O(n^2m\log m)$ [33], respectively, where B is an additional parameter introduced by the cut-and-choose technique used in [42]. Note also that there was no known semi-honest MPC shuffle protocol for Shamir secret sharing with $O(nm)$ online complexity prior to our work.
3. We prove semi-honest and universally composable (UC) security under the $\mathcal{F}_{\mathrm{MPC}}$-hybrid model, guaranteeing composability across MPC frameworks. Here $\mathcal{F}_{\mathrm{MPC}}$ is an ideal functionality that provides black-box access to basic MPC operations and an MPC permutation protocol.
4. Experimental results confirm our theoretical analysis. Our prototype shows an over 60× online speedup compared to state-of-the-art constructions, with only a moderate increase in offline overhead.

The rest of this paper is organized as follows. Sections 2–4 cover related work, preliminaries, and the technical overview. Sections 5 and 6 present the semi-honest and malicious constructions, respectively, followed by experiments in Section 7. Section 8 discusses practical instantiations and extensions. Due to the page limit, the full formal proofs and Shamir secret sharing-based experiments are provided in the full version.

2 Related Work

The main focus of this section is to provide an overview of existing work on MPC shuffle protocols, as well as to argue the broad existence of the *permute-in-turn paradigm*. To this end, we start by reviewing classical shuffle protocols, then move on to MPC-based shuffle protocols, and finally discuss the latest advances and ideas in developing MPC shuffle protocols.

Classical shuffles and mix-nets. The first shuffle protocol dates back to the seminal work of Chaum [11], which introduces the concept of mix-nets and implements anonymous communication in a server-aided scheme, assuming semi-honest servers. The work also inspires a series of follow-up studies on the concept of "decryption shuffle", which involves a sender encrypting its messages with a sequence of public keys, each belonging to one of the servers. The servers then decrypt and permute the messages in sequence, and finally output the shuffled plaintext. The construction requires linear communication overhead and a moderate number of public-key encryption and decryption operations.

This result is later extended to malicious security by a number of works [1, 2, 24, 28]. The most commonly used primitive for the enhancement is zero-knowledge proofs, where each server proves that it has permuted the ciphertext honestly, while hiding the applied permutation. This approach, while being effective, is generally computationally intensive.

MPC-based shuffle protocols. In the context of MPC, Chase et al. [10] design a shuffle protocol for the semi-honest two-party setting, which inspires a series of follow-up efforts towards further optimizing MPC shuffle protocols. At the core of their construction is a protocol that generates correlated randomness between the two parties, requiring only $O(m \log m)$ communication for m items. This generation protocol is concretely efficient, as the only public-key primitive involved is the oblivious transfer extension (OTE).

While Chase's protocol is two-party, it is later extended to the n-party setting. Laud [35] proposes an extension to an n-party shuffle protocol with $O(n^2m)$ online communication in the malicious security setting. Also building upon [10], Eskandarian and Boneh [20] and Chen et al. [12] both construct n-party shuffle protocols with only $O(nm)$ online complexity. Eskandarian and Boneh further enhance their construction, arguing that the improved version achieves malicious security. Thus, these works suggest that a maliciously secure MPC shuffle with linear online complexity is already within reach.

Malicious security and the quadratic barrier. However, a recent study by Song et al. [42] shows that the constructions proposed in [35] and [20] are in fact not secure against malicious adversaries. Through a selective-abort attack, Song et al. show that a malicious adversary can bypass correctness checks of [35] and [20] with non-negligible probability and learn partial information about the applied permutation. This makes the shuffle non-uniform in the view of a malicious adversary, thereby breaking the security of the protocols.

Such an attack creates a dilemma for the construction of maliciously secure MPC shuffle protocols: on the one hand, a malicious adversary can tamper with the messages in an attempt to break security, which must be prevented; on the other hand, it is not clear how such tampering can be detected, let alone how it can be detected with linear online complexity. To at least address one side of the dilemma, Song et al. design a maliciously secure shuffle protocol for MPC with $O(Bn^2m)$ online communication complexity, where B is a parameter arising from the cut-and-choose technique. Thus, all currently known maliciously secure MPC shuffle protocols have at least quadratic n^2m online communication and computation complexities.

These results are summarized in Table 1.

Correlated randomness and recent advances. In more recent studies, the correlated randomness used in [10] has been identified and formalized as "correlated oblivious permutation" (COP) [27], or "permutation correlation" [40]. Building upon this concept, these works further improve the efficiency of generating such randomness. Han et al. [27] further identify redundancy in generating the correlation using random OT (ROT) and halve the communication cost. Peceny et al. [40] propose a new construction for generating correlated randomness with $O(n)$ communication, based on an MPC-friendly PRF and the learning parity with noise (LPN) assumption. These two constructions are built for semi-honest two-party computation, and it remains an open question whether they can be efficiently extended to n-party malicious security.

Table 1. Comparison of MPC shuffle protocols. Bounds cover both communication and computation, which coincide online.

Protocol	Offline	Online	Security	Framework
Laur et al. [37]	$O(1)$	$O(2^n n^{1.5} m \log m)$	malicious	Threshold
Keller–Scholl [33]	$O(n^2 m \log m)$	$O(n^2 m \log m)$	malicious	Arbitrary
Laud [35]	$O(n^2 m \log m)$	$O(n^2 m)$	semi-honest	SPDZ
Clarion [20]	$O(n^2 m \log m + n^3 m)$	$O(nm)$	semi-honest	SPDZ
Song et al. [42]	$O(Bn^2 m \log m)$	$O(Bn^2 m)$	malicious	SPDZ
Ours + Keller–Scholl	$O(n^2 m \log m)$	$O(C_{\mathrm{BC}} + nm)$	malicious	Arbitrary
Ours + Song et al.	$O(Bn^2 m \log m)$	$O(C_{\mathrm{BC}} + nm)$	malicious	SPDZ

n is the number of parties, m the number of items, C_{BC} the cost of one ideal authenticated parallel broadcast of a field element, and B the cut-and-choose parameter of [42]. For simplicity, we omit the $\log |\mathbb{F}|$ factor for all constructions. See Section 8.1 for concrete broadcast costs.

Offline multiplication costs $O(n)$ in SPDZ [22]. Instantiating ours with Keller–Scholl gives $O(n^2 m \log m)$ offline complexity, assuming a constant fraction of corrupted parties.

The permute-in-turn paradigm. We note that all the above constructions follow the same structure for building MPC shuffle protocols, which we call the *permute-in-turn paradigm*. That is, in each such construction, one can identify a component that is responsible for applying a permutation to the secret-shared data (i.e., a permutation protocol), where the permutation is chosen by one of the parties. The final construction thus consists of n such components, each corresponding to one party, executed in sequence. All the constructions mentioned so far, including early works [1, 2, 11, 24, 28] and more recent ones [10, 12, 20, 27, 35, 40, 42], follow this paradigm, even though many were not originally designed for multi-party computation. Although [37] adopts a slightly different approach, it can be viewed as a variant of the paradigm, where each group of parties permutes the data in turn; a detailed case study is provided in the full version. However, to the best of our knowledge, all current maliciously secure shuffle protocols [33, 37, 42] execute the online phase of the underlying permutation protocol directly, where parties repeatedly form valid shares of the m permuted secrets n times among n parties, resulting in at least $n^2 m$ communication and computation. This makes the online phase both communication- and computation-heavy, which is undesirable for real-world applications that require quick responses.

Summary. In short, besides all following the permute-in-turn paradigm, existing MPC shuffle protocols either (i) achieve linear online complexity only with semi-honest security and additive secret sharing [12, 20], or (ii) support malicious security at the cost of quadratic online overhead [33, 37, 42]. This leaves open the central question we address in this paper: *how to design a semi-honest (resp. malicious) secure MPC shuffle protocol with linear online complexity and compatibility across secret-sharing schemes?*

3 Preliminaries

3.1 Basic Notations

Let there be n parties $P_1, P_2, ..., P_n$, and m field elements to shuffle. Throughout this paper, it is assumed that all elements and operations in multiparty computation protocols are over the field $\mathbb{F}$, with $|\mathbb{F}| = \Omega(nm2^\kappa)$ for any fixed statistical security parameter κ. Denote by

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

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

We use lowercase letters (e.g. x, y, z) for integers, and bold letters (e.g. $\mathbf{x}, \mathbf{y}, \mathbf{z}$) for vectors. We denote by $\mathbf{x}(i)$ or x_i the i-th entry of the vector $\mathbf{x}$. The notation $\mathbf{x}(i)$ is useful when handling vectors with indices, e.g. vectors $\mathbf{x}_1$ and $\mathbf{x}_2$. As we are mostly dealing with vectors of length m, we assume all vectors are of length m, unless otherwise specified. 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 a linear operation, i.e.,

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

where the addition of two vectors is defined component-wise.

The composition of two permutations π_2, π_1 is another permutation, denoted by $\pi_2 \circ \pi_1$ and satisfies

$$\pi_2 \circ \pi_1(\mathbf{x}) = \pi_2(\pi_1(\mathbf{x})).$$

For convenience in later notation, for the composition of 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:

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

where $\mathit{\Pi}$ 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 the meaning of "stored at $\mathcal{F}_{\text{MPC}}$" in the next subsection.

3.2 Primitives

We assume several secure MPC primitives, including inputting a secret, opening a secret, generating random values, and computing additions and multiplications over $\mathbb{F}$. We also assume a secure MPC permutation protocol, which allows one

of the parties to input a secret permutation and securely permute a vector. We also assume secure pairwise channels and an authenticated broadcast channel, which is a standard practice in the MPC literature [23].

To formalize, we follow the approach of the arithmetic black-box (ABB) model adopted by Escudero et al. [19], where an ideal arithmetic MPC functionality denoted by $\mathcal{F}_{\text{MPC}}$ is viewed as a Turing machine with internal state. The functionality interacts honestly with all parties, taking inputs and (restricted) commands from them and updating its internal state accordingly. We assume that the ideal arithmetic MPC functionality $\mathcal{F}_{\text{MPC}}$ supports the 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 is agreed upon by all parties, 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 the same as Π_{add}, except it computes multiplications.
- $\Pi_{\text{open}}(\text{id})$, which retrieves (id, x) and outputs x to the adversary. If the adversary replies with "continue", then $\mathcal{F}_{\text{MPC}}$ also sends 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.
- $\Pi_{\text{broadcast}}((P_i : x_i)_{i \in S})$, which takes x_i from every P_i in a nonempty set S and broadcasts all these values to all parties in parallel. The case $|S| = 1$ is ordinary single-sender broadcast.
- $\Pi_{\text{perm}}(P_i : \pi, (\text{id}_\text{j})_{j=1}^m, (\text{id}'_\text{j})_{j=1}^m)$, which takes a secret m-permutation from P_i, retrieves (id_j, x_j) and stores $(\text{id}'_j, x_{\pi(j)})$.
- $\Pi_{\text{abort}}()$, which sends "abort" to all parties. This command is invoked by the adversary.

Throughout the paper, $\Pi_{\text{broadcast}}$ denotes an ideal authenticated-broadcast functionality. Its concrete realization and cost under different corruption and setup assumptions are discussed in Section 8.1. For analytical purposes, let C_{BC} upper-bound both the communication and computation costs of one such parallel-broadcast invocation in which every party broadcasts one field element. Broadcasting a constant number of elements costs $O(C_{\text{BC}})$, while broadcasting a length-m vector costs $O(C_{\text{BC}} + nm)$.

In practice, under the semi-honest setting, we do not need Π_{mul}. $\mathcal{F}_{\text{MPC}}$ can thus be instantiated with Shamir secret sharing or additive secret sharing without MAC. For the malicious setting, $\mathcal{F}_{\text{MPC}}$ may be instantiated with additive secret sharing (e.g. [18, 31, 32]), Shamir's secret sharing (e.g. [7, 17]), replicated secret sharing [15, 38], or other MPC frameworks, based on specific security requirements and application scenarios.

Note that we assume that $\mathcal{F}_{\text{MPC}}$ checks the correctness of the arguments. For example, if Π_{perm} is invoked with a non-permutation, $\mathcal{F}_{\text{MPC}}$ will output "abort" to all parties. During the execution, the malicious adversary can also invoke Π_{abort} at any time to abort the protocol.

For notational simplicity, we denote by $[\![x]\!]$ a secret value x stored by $\mathcal{F}_{\text{MPC}}$. This can also be understood as "sharing x among all parties", since $\mathcal{F}_{\text{MPC}}$ is implemented by secure multiparty computation in practice. To distinguish between semi-honest and malicious settings, we write $\langle x \rangle$ for a secret stored by the maliciously secure ideal functionality. We will use $[\![\cdot]\!]$ only in the semi-honest setting and $\langle \cdot \rangle$ only in the malicious setting. We also use

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

to denote the process of computing $a \cdot b + c$ and storing it in the ideal functionality. Note that each symbol corresponds to a distinct identifier for $\mathcal{F}_{\text{MPC}}$, which allows us to omit the identifier in later descriptions. For vector $\mathbf{x} = (x_1, x_2, ..., x_m)$, we use $[\![\mathbf{x}]\!]$ to denote storing each entry separately in the ideal MPC functionality. Hence, we also use

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

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

Similarly, we write

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

where

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

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

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

where $\langle \mathbf{x}_1, ..., \mathbf{x}_t \rangle$ is an abbreviation for $(\langle \mathbf{x}_1 \rangle, ..., \langle \mathbf{x}_t \rangle)$. This is a natural requirement for a permutation protocol, since in real-world applications, what is to be shuffled is usually a list of vectors (e.g., rows of the database) rather than a list of field elements. As concrete examples, all protocols in [10,20,35,37,42] support such an operation.

3.3 Permute-in-Turn Paradigm

To shuffle a secret-shared vector $[\![\mathbf{x}]\!]$ into some $[\![\pi(\mathbf{x})]\!]$ with a secret, uniformly random π, most previous works implicitly follow a permute-in-turn paradigm and build MPC shuffle protocols from MPC permutation protocols.

To illustrate, suppose we have built a protocol Permute that securely implements Π_{perm}, i.e.,

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

Then the shuffle protocol can be implemented via sequential invocations of Permute, where each party P_i selects a random permutation π_i, and parties compute sequentially for $i = 1, 2, \ldots, n$

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

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 not known to any party. In addition, as long as at least one honest party P_i has chosen its permutation π_i uniformly at random, the resulting π will be uniformly random. Hence, due to the adversary lacking access to the π_i chosen by honest parties, the resulting π is independent of the adversary's view, which accomplishes the security goal of the shuffle protocol.

We formalize such an approach as the *permute-in-turn paradigm*: to build a shuffle protocol, we first implement a permutation protocol, and then use it to construct the shuffle protocol. Former works mostly concentrated on the design of the permutation protocol [10,33,35,37,42], while the paradigm itself is scarcely discussed or questioned.

Although the above construction works effectively, its online complexities are at least n times those of Permute, since there are n sequential invocations. This problem becomes more severe when the online overhead of Permute is itself super-linear. As concrete examples, the implementation in [33] incurs $O(nm \log m)$ online overhead and [42] incurs $O(Bnm)$, which is then translated to $O(n^2 m \log m)$ and $O(n^2 Bm)$ online overhead in the shuffle protocol, respectively.

Our first step is to rewrite the above naive permute-in-turn paradigm as two phases, an offline phase $\text{Shuffle}_{\text{off}}$ and an online phase $\text{Shuffle}_{\text{on}}$, and moving all invocations to Π_{perm} to the offline phase. The online complexities of our constructions, besides now becoming $O(C_{\text{BC}} + nm)$, are also therefore independent of the implementation of Π_{perm}, while most constructions in previous works are not.

3.4 Security Model

Throughout this paper, we consider a static adversary, i.e. the corrupted parties are chosen and fixed before the protocol starts. We consider the case of dishonest majority, where an adversary can corrupt up to $n - 1$ parties.[5]

[5] Note that assuming dishonest majority here is without loss of generality, as our security proof does not rely on the specific number t of corrupted parties, besides the trivial requirement that $0 \leq t < n$. If we want to instantiate $\mathcal{F}_{\text{MPC}}$ with Shamir secret sharing under honest majority, setting $t = \lfloor (n-1)/2 \rfloor$ will do. In practice, the only concern is that the implementation of $\mathcal{F}_{\text{MPC}}$ must match the threat model.

Semi-honest security. Our first construction considers semi-honest security, under the $\mathcal{F}_{\text{MPC}}$-hybrid model, where $\mathcal{F}_{\text{MPC}}$ is semi-honest secure. In the semi-honest setting, the adversary corrupting some parties is assumed to follow the protocol honestly, but nevertheless may do some extra computations to gain information about other parties' inputs.

Definition 1 (Semi-honest Secure Shuffle). *Let $\mathcal{F}_{\text{shuffle}}$ take a shared vector $[\![\mathbf{x}]\!]$ and a private m-permutation π_i from each P_i, set $\pi = \pi_n \circ \cdots \circ \pi_1$, and store $[\![\pi(\mathbf{x})]\!]$ without revealing anything else.*

For a static set $T \subsetneq [n]$, let $\text{REAL}^{\mathcal{F}_{\text{MPC}}}_{\text{Shuffle},\mathcal{A},T}$ denote the output of the environment and a semi-honest adversary $\mathcal{A}$ corrupting T in the real execution of Shuffle*. Let $\text{IDEAL}_{\mathcal{F}_{\text{shuffle}},\mathcal{S},T}$ denote the corresponding ideal execution with a simulator $\mathcal{S}$, which receives only the corrupted parties' inputs and prescribed outputs. Protocol* Shuffle *securely realizes $\mathcal{F}_{\text{shuffle}}$ in the $\mathcal{F}_{\text{MPC}}$-hybrid model if, for every probabilistic polynomial-time semi-honest $\mathcal{A}$, there exists a probabilistic polynomial-time $\mathcal{S}$ such that, for every input and auxiliary input, the ensembles*

$$\text{REAL}^{\mathcal{F}_{\text{MPC}}}_{\text{Shuffle},\mathcal{A},T} \approx \text{IDEAL}_{\mathcal{F}_{\text{shuffle}},\mathcal{S},T}$$

are indistinguishable.

In both security models, π_i is a private input supplied by party P_i before correlation generation. Security is defined for arbitrary valid permutation inputs; the environment supplies the honest parties' inputs in the security experiment. A uniformly random shuffle is obtained by choosing these inputs independently and uniformly (it suffices that one honest permutation is uniform and independent of the other inputs). Permutation inputs are distinct from the fresh randomness used to generate each correlation. In the security proof, we will prove semi-honest security for the protocol

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

i.e., the composition of the two phases of the shuffle protocol is secure.

Malicious security. Our second construction considers security against a malicious adversary, who now can arbitrarily deviate from the protocol specification. Below is a sketch of the universally composable (UC) security definition for a maliciously secure shuffle protocol. The full definition and proof appear in the full version.

Definition 2 (Universally Composable Security [9], Sketch). *Let $\mathcal{E}$ be an environment and $\mathcal{A}$ be an adversary controlling the corrupted parties $P_i \in T \subsetneq [n]$. Let the ideal functionality $\mathcal{F}$ be $\mathcal{F}_{\text{MPC}}$ equipped with an additional ideal command Π_{Shuffle}. This command takes an agreed shared vector $\langle \mathbf{x} \rangle$ and a private m-permutation π_i from each party P_i. Invalid inputs cause a common abort. Otherwise it sets $\pi = \pi_n \circ \cdots \circ \pi_1$ and, unless the adversary requests abort, returns a fresh handle for $\langle \pi(\mathbf{x}) \rangle$. An abort returns no output handle and reveals*

no input or honest permutation. The output remains shared unless subsequently opened. Let protocol Shuffle *be an implementation of the command* Π_{Shuffle}.

Consider the following two games. One takes place among $\mathcal{E}$, $\mathcal{A}$, $\mathcal{F}_{\text{MPC}}$, *and the honest parties* $P_i \in \bar{T} = \{P_i \notin T\}$, *which execute the real protocol* Shuffle. *The other takes place among* $\mathcal{E}$, *the simulator* $\mathcal{S}$, *and the ideal functionality* $\mathcal{F}$, *with* $\mathcal{S}$ *simulating the adversary's view in the ideal execution. In each game,* $\mathcal{E}$ *chooses* $\mathbf{x}$ *and the private permutations of the honest parties, and sends them either to the honest parties in the real execution or directly to* $\mathcal{F}$ *in the ideal execution. 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-world execution, if the protocol does not abort,* $\mathcal{E}$ *receives the outputs of the honest parties from those parties.*

In the ideal world, $\mathcal{S}$ *receives neither* $\mathbf{x}$ *nor the honest parties' permutation inputs. It extracts the corrupted parties' permutations from their calls to* Π_{perm} *and sends them to* $\mathcal{F}$. *If* $\mathcal{E}$ *does not demand* $\mathcal{S}$ *to abort, and the protocol ends without abort,* $\mathcal{S}$ *sends "*continue*" to* $\mathcal{F}$, *which then returns the shared output handle. In the comparison experiment,* $\mathcal{E}$ *may obtain the vector through a subsequent opening, as formalized in the full version.*

$\mathcal{E}$ *continues 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 as to which game it is playing.*

The protocol Shuffle *securely implements* Π_{Shuffle}, *if for every adversary* $\mathcal{A}$ *there exists a probabilistic polynomial-time simulator* $\mathcal{S}$ *such that every environment* $\mathcal{E}$ *has distinguishing advantage bounded by*

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

Intuitively, this definition guarantees that under the $\mathcal{F}_{\text{MPC}}$-hybrid model, the adversary learns nothing about $\mathbf{x}$ or the honest parties' permutation inputs beyond what the ideal functionality reveals, whether the protocol completes or aborts. In particular, an abort must not reveal either an honest party's π_i or the composed permutation π. Section 8.1 discusses the integrity checks required to realize this stronger functionality in concrete MPC frameworks.

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", since $\mathcal{E}$ could perform all computations and decide the (malicious) actions. Also, the definition does not limit the computational resources of the environment, and hence achieves statistical security. Note that since we are working under the $\mathcal{F}_{\text{MPC}}$-hybrid model, this level of security is achievable with dishonest majority. The formal definition and full proof appear in the full version.

4 Technical Overview

4.1 High-Level Idea

Figure 2 below illustrates the idea behind our shuffle protocols, assuming there are four parties.

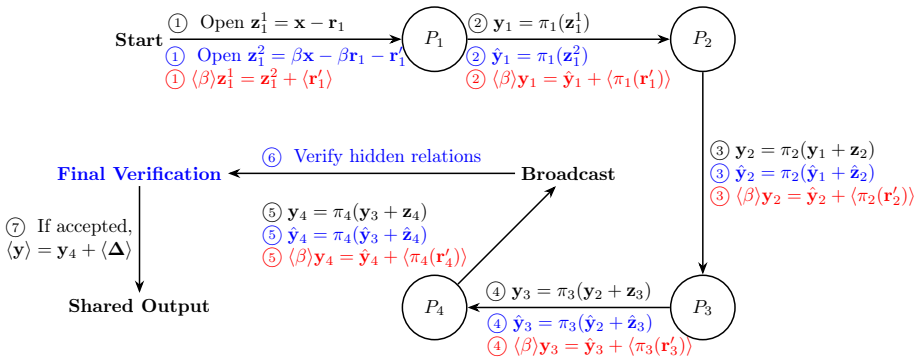

Fig. 2. Overview of online phases for 4 parties. Black steps are common to both settings; blue steps are added for malicious security; red text marks the hidden relation expected between $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$. In the malicious setting, the shared output is produced only after the final verification accepts.

Our basic idea is similar to the decryption shuffle by Chaum [11], where each server randomly permutes and decrypts with its own private key in turn. In our protocol, the parties first open a masked input $\mathbf{z}_1 = \mathbf{x} - \mathbf{r}_1$ to P_1, which computes $\mathbf{y}_1 = \pi_1(\mathbf{z}_1)$. Each subsequent party P_i applies its own secret permutation π_i and adds a one-time pad $\mathbf{z}_i$ that hides the permutation. For $i = 2, \ldots, n$, P_i sends

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

to the next party. At the end, P_n broadcasts its $\mathbf{y}_n$ and all parties jointly decrypt the message by adding the correction $\boldsymbol{\Delta}$ and share the shuffled vector $\mathbf{y} = \pi_n \circ \cdots \circ \pi_1(\mathbf{x})$, where

$$[\![\boldsymbol{\Delta}]\!] = [\![\pi_n(\mathbf{r}_n)]\!] \tag{2}$$

is computed in the offline phase.[6]

[6] In the MPC literature, such pre-shared random values are commonly referred to as *correlated randomness* or *correlation*. We will formalize the requirements over $\mathbf{z}_i$ and $\boldsymbol{\Delta}$ in Definition 3, where we define semi-honest shuffle correlation.

To achieve malicious security, we require each party P_i to send an additional message $\hat{\mathbf{y}}_i$ along with $\mathbf{y}_i$, implementing a hidden relation between $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$. By design, we require that

$$\langle\beta\rangle\mathbf{y}_i = \hat{\mathbf{y}}_i + \langle\pi_i(\mathbf{r}'_i)\rangle, \tag{3}$$

for a secret $\langle\beta\rangle$ and a secret random vector $\langle\pi_i(\mathbf{r}'_i)\rangle$. After $\mathbf{y}_n$ is fixed, the parties jointly verify that the received pairs $(\mathbf{y}_{i-1}, \hat{\mathbf{y}}_{i-1})$ satisfy Equation 3 before producing the output.

Intuitively, since β and $\pi_i(\mathbf{r}'_i)$ are hidden behind the curtain of $\mathcal{F}_{\text{MPC}}$ from all parties, no malicious party can learn them and forge $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$ that satisfy Equation 3, while all parties can jointly check Equation 3 with access to $\mathcal{F}_{\text{MPC}}$.

This blueprint leaves three challenges[7]:

1. How to generate the shuffle correlation, in particular $\boldsymbol{\Delta}$, efficiently with Π_{perm}?
2. How to check the hidden relation (Equation 3) efficiently with low communication and computation?
3. Why does checking Equation 3 alone guarantee security against malicious adversaries?

4.2 Generating Shuffle Correlation Efficiently

We sample independent masks $\mathbf{r}_1, \ldots, \mathbf{r}_n$ and set

$$\mathbf{z}_i = \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i, \quad i = 2, \ldots, n, \quad \Longrightarrow \quad \boldsymbol{\Delta} = \pi_n(\mathbf{r}_n), \tag{4}$$

with the initial mask $\mathbf{z}_1 = \mathbf{x} - \mathbf{r}_1$ generated online. This cancels masks successively: starting from $\mathbf{y}_1 = \overline{\pi}_1(\mathbf{x}) - \pi_1(\mathbf{r}_1)$, for every $i = 2, \ldots, n$ we have

$$\begin{aligned} \mathbf{y}_{i-1} + \mathbf{z}_i &= \overline{\pi}_{i-1}(\mathbf{x}) - \pi_{i-1}(\mathbf{r}_{i-1}) + \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i \\ &= \overline{\pi}_{i-1}(\mathbf{x}) - \mathbf{r}_i \\ \Longrightarrow\ \mathbf{y}_i &= \overline{\pi}_i(\mathbf{x}) - \pi_i(\mathbf{r}_i), \end{aligned}$$

where $\overline{\pi}_i = \pi_i \circ \cdots \circ \pi_1$. As $\mathbf{y}_n = \overline{\pi}_n(\mathbf{x}) - \pi_n(\mathbf{r}_n)$, $\boldsymbol{\Delta} = \pi_n(\mathbf{r}_n)$.

4.3 Checking the Hidden Relation Efficiently

Every P_{i+1} retains its received pair $(\mathbf{y}_i, \hat{\mathbf{y}}_i)$. We use a preprocessed random challenge c_{i+1}, which is opened to P_{i+1} after all online messages are fixed.[8] The compressed hidden relation is

$$\langle\beta\rangle \underbrace{\sum_{j=1}^{m} c_{i+1}^{j-1}\mathbf{y}_i(j)}_{u_{i+1}} = \underbrace{\sum_{j=1}^{m} c_{i+1}^{j-1}\hat{\mathbf{y}}_i(j)}_{v_{i+1}} + \underbrace{\sum_{j=1}^{m} c_{i+1}^{j-1}\langle\pi_i(\mathbf{r}'_i)(j)\rangle}_{\langle d_{i+1}\rangle}. \tag{5}$$

[7] We omit here for brevity the problem of implementing the hidden relation. See Definition 4 for the formal definition and requirements over these variables.

[8] The security does not depend on the adversary not knowing c_{i+1}. It requires only that the adversary does not know c_{i+1} *before* $\mathbf{y}_i$ is fixed.

Note that d_{i+1} is input-independent, which can be preprocessed offline. Thus P_{i+1} broadcasts only (u_{i+1}, v_{i+1}). After all such values are fixed, the parties combine the resulting scalar residuals and open one blinded aggregate. This costs $O(nm)$ local processing and one parallel-broadcast invocation, for total online complexity $O(C_{\mathrm{BC}} + nm)$. Algorithm 3 specifies the check.

4.4 Intuition behind Security

Before the final verification, fresh masks make the message-phase view independent of the input $\mathbf{x}$ and every honest permutation π_i. The hidden $\langle\beta\rangle$-relation prevents a malicious party from producing a different valid pair. If a different forged pair $(\mathbf{y}'_i, \hat{\mathbf{y}}'_i)$ also satisfies the β-relation, then

$$\beta\big(\mathbf{y}_i - \mathbf{y}'_i\big) = \hat{\mathbf{y}}_i - \hat{\mathbf{y}}'_i. \tag{6}$$

If the first components differ, this equation determines β from any coordinate at which $\mathbf{y}_i$ and $\mathbf{y}'_i$ differ; otherwise, it forces the second components to be equal as well. Thus producing a different valid pair requires guessing β correctly, which succeeds with probability at most $1/|\mathbb{F}|$. Section 6.6 gives a security proof sketch; the full proof is provided in the full version.

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 the operation

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

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 essentially shuffling random values in order to generate a shuffle correlation. The online phase takes as input a length-m array $[\![\mathbf{x}]\!]$, consumes a fresh shuffle correlation, and outputs shuffled $[\![\pi(\mathbf{x})]\!]$.

5.2 Semi-honest Shuffle Correlation

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

Definition 3 (Semi-honest Shuffle Correlation). *The shuffle correlation for the semi-honest setting is defined as*

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

where

1. *π_i is the private m-permutation input of party P_i.*
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. The vectors $\mathbf{z}_i$ are sampled uniformly at random, subject to the constraint*

$$\mathbf{s} = \pi_n(\cdots \pi_3(\pi_2(\pi_1(\mathbf{r}) - \mathbf{z}_2) - \mathbf{z}_3) \cdots - \mathbf{z}_n).$$

5.3 Offline Phase

As is discussed in Section 4, we decouple the correlation between $\mathbf{r}$ and $\mathbf{s}$ by letting

$$\begin{aligned}
&[\![\mathbf{r}_i]\!] \leftarrow \Pi_{\text{rand}}(),\\
&[\![\mathbf{z}_i]\!] \leftarrow \pi_{i-1}([\![\mathbf{r}_{i-1}]\!]) - [\![\mathbf{r}_i]\!], \text{ for } i = 2, 3, \ldots, n,\\
&[\![\mathbf{s}]\!] \leftarrow [\![\pi_n(\cdots \pi_3(\pi_2(\pi_1(\mathbf{r}_1) - \mathbf{z}_2) - \mathbf{z}_3) \cdots - \mathbf{z}_n)]\!] = [\![\pi_n(\mathbf{r}_n)]\!].
\end{aligned}$$

One can rigorously show that these two definitions are equivalent, by constructing a bijection and showing their distributions are the same. This definition also offers an immediate solution for generating the shuffle correlation, which is formally presented in Algorithm 1.

Algorithm 1 cor $\leftarrow \text{Shuffle}_{\text{off}}(m)$

Require: Each P_i holds a private m-permutation input π_i.
Ensure: Outputs a new shuffle correlation for these inputs.

```
for i = 1 to n do parallel
    Generate [[r_i]] of length m with Π_rand.
    [[π_i(r_i)]] ← Π_perm(P_i : π_i, [[r_i]]).
end for
for i = 2 to n do parallel
    [[z_i]] ← [[π_{i-1}(r_{i-1})]] − [[r_i]]
    Open [[z_i]] to P_i.
end for
Output {(π_1, ..., π_n), [[r_1]], [[π_n(r_n)]], (∅, z_2, z_3, ..., z_n)}.
```

5.4 Online Phase

As discussed in Section 4, in the online phase, each party P_i computes and sends to P_{i+1}

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

where $\mathbf{y}_0 = \mathbf{0}$ and $\mathbf{z}_1 \coloneqq \mathbf{x} - \mathbf{r}_1$ is opened to P_1. The last party P_n broadcasts $\mathbf{y}_n$, and all parties jointly remove the accumulated mask $-\pi_n(\mathbf{r}_n)$ to obtain the final result. The complete protocol is formalized in Algorithm 2.

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

Require: $\{(\pi_1, \ldots, \pi_n), [\![\mathbf{r}_1]\!], [\![\pi_n(\mathbf{r}_n)]\!], (\emptyset, \mathbf{z}_2, \mathbf{z}_3, \ldots, \mathbf{z}_n)\}$ from offline phase.
Ensure: Outputs $[\![\pi(\mathbf{x})]\!]$, 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.
$[\![\mathbf{x}']\!] \leftarrow \mathbf{y}_n + [\![\pi_n(\mathbf{r}_n)]\!]$
Output $[\![\mathbf{x}']\!]$.

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

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

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

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

The full security proof is provided in the full version.

6 Maliciously Secure Shuffle

As discussed in Section 4, our malicious shuffle protocol requires an additional MAC and a checking procedure. Hence, in this section, we first define the malicious shuffle correlation, then present the batched MAC checking protocol, and finally present the offline and online phases of the malicious shuffle protocol.

6.1 Malicious Shuffle Correlation

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

Definition 4 (Extended Malicious Shuffle Correlation). *The shuffle correlation for the malicious setting is defined as*

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

where

1. *π_i is the private m-permutation input of party P_i.*
2. *$\langle\beta\rangle$ is a secret-shared random variable. It plays the role of the authentication key, i.e., the MAC key.*
3. *The vectors $\mathbf{r}_i, \mathbf{r}'_i$ are independently uniform in $\mathbb{F}^m$. The displayed shared values are retained for the online phase and the final verification.*
4. *For $i = 2, \ldots, n$, c_i is uniformly random, with*
$$d_i = \sum_{j=1}^{m} c_i^{j-1} \pi_{i-1}(\mathbf{r}'_{i-1})(j).$$
5. *$\mathbf{z}_i = (\mathbf{z}_i^1, \mathbf{z}_i^2)$, where each $\mathbf{z}_i^j$ is a vector of length m. For $i = 2, \ldots, n$, they satisfy*
$$\mathbf{z}_i^1 = \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i,$$
$$\mathbf{z}_i^2 = \beta\mathbf{z}_i^1 + \pi_{i-1}(\mathbf{r}'_{i-1}) - \mathbf{r}'_i.$$

Remark. *For readers familiar with SPDZ [18], we note that although β plays a role similar to that of the MAC key in the SPDZ framework (which is usually denoted by "α" in this context), the two keys are different and should not be confused. For example, when instantiated with SPDZ, a maliciously secure shared value, say $\langle\mathbf{r}\rangle$, is essentially $([\![\mathbf{r}]\!], [\![\alpha\mathbf{r}]\!])$, i.e., the value and its SPDZ MAC. Our MAC for shuffle is then $\langle\beta\mathbf{r}\rangle = ([\![\beta\mathbf{r}]\!], [\![\alpha\beta\mathbf{r}]\!])$, which is orthogonal and additional to the SPDZ MAC $[\![\alpha\mathbf{r}]\!]$.*

6.2 Batched Correlation Check

Protocol Verify checks all intermediate and final correlations together after every online message has been fixed. Each P_i, for $i = 2, \ldots, n$, retains the message $\mathbf{y}_{i-1}$ that it received. The protocol opens only one blinded aggregate residual, and therefore does not reveal an individual check result or a failing index.

The protocol is formally presented in Algorithm 3.

6.3 Offline Phase

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

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

The offline phase is formally described in Algorithm 4.

Algorithm 3 Verify$((P_i : \mathbf{y}_{i-1})_{i=2}^n, \mathbf{y}_n, \text{cor})$

Require: Fresh cor from Algorithm 4, and all $\mathbf{y}_1, \ldots, \mathbf{y}_n$ are fixed.
Ensure: Abort w.h.p. if any intermediate or final correlation is invalid.

for $i = 2$ to n **do parallel**
  Parties open $\langle c_i \rangle$ to P_i.
  P_i computes $w_{i,b} \coloneqq \sum_{j=1}^m c_i^{j-1} \mathbf{y}_{i-1}^b(j)$, $b \in \{1, 2\}$.
  P_i broadcasts $(w_{i,1}, w_{i,2})$.
  $\langle \eta_i \rangle \leftarrow w_{i,1}\langle \beta \rangle - w_{i,2} - \langle d_i \rangle$.
end for
Wait until all $(w_{i,1}, w_{i,2})$ are fixed.
Draw and open $\langle e \rangle \leftarrow \Pi_{\text{rand}}()$.
$\langle \eta_{n+1} \rangle \leftarrow \sum_{j=1}^m e^{j-1}\big(\mathbf{y}_n^1(j)\langle \beta \rangle - \mathbf{y}_n^2(j) - \langle \pi_n(\mathbf{r}'_n)(j) \rangle\big)$.
Draw and open $\langle \rho \rangle \leftarrow \Pi_{\text{rand}}()$.
$\langle s \rangle \leftarrow \sum_{i=2}^{n+1} \rho^{i-2} \langle \eta_i \rangle$.
$\langle \gamma \rangle \leftarrow \Pi_{\text{rand}}()$.
$\langle f \rangle \leftarrow \langle \gamma \rangle \cdot \langle s \rangle$.
Open f. Abort if $f \neq 0$.

Algorithm 4 cor $\leftarrow$ Shuffle$_{\text{off}}(m)$

Require: Each P_i holds a private m-permutation input π_i.
Ensure: Outputs a shuffle correlation for these inputs, or aborts.

Generate random value $\langle \beta \rangle$.
for $i = 1$ to n **do parallel**
  Generate random vectors $\langle \mathbf{r}_i \rangle$ and $\langle \mathbf{r}'_i \rangle$.
  if $i = 1$ **then**
    $\langle \beta \mathbf{r}_1 \rangle \leftarrow \Pi_{\text{mul}}(\langle \mathbf{r}_1 \rangle, \langle \beta \rangle)$
  end if
  $\langle \pi_i(\mathbf{r}_i), \pi_i(\mathbf{r}'_i) \rangle \leftarrow \Pi_{\text{perm}}\,(P_i : \pi_i, \langle \mathbf{r}_i, \mathbf{r}'_i \rangle)$
  if $i \geq 2$ **then**
    $\langle \mathbf{z}_i^1 \rangle \leftarrow \langle \pi_{i-1}(\mathbf{r}_{i-1}) \rangle - \langle \mathbf{r}_i \rangle$
    $\langle \mathbf{z}_i^2 \rangle \leftarrow \langle \pi_{i-1}(\mathbf{r}'_{i-1}) \rangle - \langle \mathbf{r}'_i \rangle + \Pi_{\text{mul}}(\langle \mathbf{z}_i^1 \rangle, \langle \beta \rangle)$
    $\langle \mathbf{z}_i \rangle \coloneqq (\langle \mathbf{z}_i^1 \rangle, \langle \mathbf{z}_i^2 \rangle)$
    $\langle c_i \rangle \leftarrow \Pi_{\text{rand}}()$
    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

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

Remark. *Computing $\langle c_i^j \rangle$ for all $j = 2, 3, \ldots, m-1$ can be done efficiently with a divide-and-conquer strategy, costing $O(m)$ calls to Π_{mul} and $O(\log m)$ rounds. The round complexity can be further reduced to $O(1)$ if given access to sampling random $\langle u \rangle$ and $\langle u^{-1} \rangle$, as shown in [16].*

To understand the process, note that the online phase consists essentially of two semi-honest shuffle protocols, one for shuffling $\mathbf{x}$ and the other for $\beta\mathbf{x}$. Naturally, the mask for shuffling $\mathbf{x}$ is the same as in the semi-honest case, which is

$$\mathbf{z}_i^1 = \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i. \tag{7}$$

As for the mask for $\beta\mathbf{x}$, we need to ensure that

$$\beta\mathbf{y}_i^1 = \mathbf{y}_i^2 + \mathbf{r}'_i, \tag{8}$$

which is consistent with the integrity check to be carried out in the online phase. Further, note that

$$\mathbf{y}_i^1 = \pi_i(\mathbf{y}_{i-1}^1 + \mathbf{z}_i^1), \tag{9}$$
$$\mathbf{y}_i^2 = \pi_i(\mathbf{y}_{i-1}^2 + \mathbf{z}_i^2) = \pi_i(\beta\mathbf{y}_{i-1}^1 - \mathbf{r}'_{i-1} + \mathbf{z}_i^2). \tag{10}$$

By plugging the Equations 9 and 10 into Equation 8, we conclude that

$$\beta\pi_i(\mathbf{y}_{i-1}^1 + \mathbf{z}_i^1) = \pi_i(\beta\mathbf{y}_{i-1}^1 - \mathbf{r}'_{i-1} + \mathbf{z}_i^2) + \mathbf{r}'_i \tag{11}$$
$$\implies \beta\mathbf{y}_{i-1}^1 + \beta\mathbf{z}_i^1 = \beta\mathbf{y}_{i-1}^1 - \mathbf{r}'_{i-1} + \mathbf{z}_i^2 + \pi_i^{-1}(\mathbf{r}'_i) \tag{12}$$
$$\implies \mathbf{z}_i^2 = \beta\mathbf{z}_i^1 + \mathbf{r}'_{i-1} - \pi_i^{-1}(\mathbf{r}'_i). \tag{13}$$

To avoid computing $\pi_i^{-1}(\mathbf{r}'_i)$, we simply replace each $\mathbf{r}'_i$ term by $\pi_i(\mathbf{r}'_i)$, which finally leads to

$$\mathbf{z}_i^2 = \beta\mathbf{z}_i^1 + \pi_{i-1}(\mathbf{r}'_{i-1}) - \mathbf{r}'_i. \tag{14}$$

This explains the process formalized in Algorithm 4.

Similar to the semi-honest case, it is also straightforward to show that the output of Algorithm 4 is consistent with Definition 4.

6.4 Online Shuffle

As also mentioned in the previous subsection, the online phase consists essentially of two concurrent sessions of the semi-honest shuffle protocol, i.e., each P_i sends to P_{i+1}

$$\mathbf{y}_i^1 \leftarrow \pi_i(\mathbf{y}_{i-1}^1 + \mathbf{z}_i^1) = \overline{\pi}_i(\mathbf{x}) - \pi_i(\mathbf{r}_i),$$
$$\mathbf{y}_i^2 \leftarrow \pi_i(\mathbf{y}_{i-1}^2 + \mathbf{z}_i^2) = \overline{\pi}_i(\beta\mathbf{x}) - \pi_i(\beta\mathbf{r}_i + \mathbf{r}'_i).$$

We abbreviate this operation as $\mathbf{y}_i \leftarrow \pi_i(\mathbf{y}_{i-1} + \mathbf{z}_i)$, where each vector is of length $2m$. Applying π_i means permuting both halves of the $2m$-length vector by π_i.

To achieve malicious security, each P_i, for $i = 2, \ldots, n$, retains the message $\mathbf{y}_{i-1}$ that it receives. After all messages are fixed, the parties call Verify once to check simultaneously whether

$$\beta \mathbf{y}_{i-1}^1 = \mathbf{y}_{i-1}^2 + \pi_{i-1}(\mathbf{r}'_{i-1}) \quad \text{for every } i = 2, \ldots, n,$$

and whether the same correlation holds for the final broadcast $\mathbf{y}_n$. The individual residuals remain secret-shared and are compressed into one blinded aggregate before opening, so the verification reveals no individual check result or failing index.

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 from Algorithm 4.
Ensure: Outputs $\langle \overline{\pi}_n(\mathbf{x}) \rangle$ 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 $\mathbf{z}_1 := (\mathbf{z}_1^1, \mathbf{z}_1^2)$ to P_1.
P_1 computes and sends to P_2 $\mathbf{y}_1 = \pi_1(\mathbf{z}_1)$.
for $i = 2$ to $n - 1$ **do**
  P_i stores $\mathbf{y}_{i-1}$, then computes and sends to P_{i+1} $\mathbf{y}_i = \pi_i(\mathbf{z}_i + \mathbf{y}_{i-1})$.
end for
P_n stores $\mathbf{y}_{n-1}$, then computes and broadcasts $\mathbf{y}_n = \pi_n(\mathbf{z}_n + \mathbf{y}_{n-1})$.
Verify$((P_i : \mathbf{y}_{i-1})_{i=2}^n, \mathbf{y}_n, \text{cor})$.
$\langle \mathbf{y}_n^1 \rangle \leftarrow \Pi_{\text{input}}(\mathbf{y}_n^1)$
$\langle \mathbf{x}' \rangle \leftarrow \langle \mathbf{y}_n^1 \rangle + \langle \pi_n(\mathbf{r}_n) \rangle$
Output $\langle \mathbf{x}' \rangle$.

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

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

And the $\mathbf{y}_i$ sent by each party P_i to P_{i+1} is

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

Hence, if the parties input the first m entries of $\mathbf{y}_n$ as $\langle \mathbf{y}_n^1 \rangle$ and add $\langle \pi_n(\mathbf{r}_n) \rangle$ to it, the result is indeed $\langle \overline{\pi}_n(\mathbf{x}) \rangle$. Also, if all parties act honestly, every residual computed by Verify is zero, so the verification passes.

Remark. *We note that in real-world MPC implementations, values opened by Π_{open} may remain tentative until a backend authentication check. This differs from our arithmetic black-box model. Section 8.1 gives the required order for authenticating the computation and opening f before any branch or output.*

6.5 Complexity Analysis

Let $R_{\text{off}}, R_{\text{on}}$ (resp. $M_{\text{off}}, M_{\text{on}}$) denote the offline and online costs of Π_{rand} (resp. Π_{mul}), and P the total cost of Π_{perm}. All randomness is preprocessed ($R_{\text{on}} = 0$); online Π_{rand} calls consume fresh sharings.

The offline communication and computation complexity is

$$O(nP + nmR_{\text{off}} + nm(M_{\text{off}} + M_{\text{on}})).$$

Correlation generation executes both parts of its multiplications; preprocessing for online multiplications is also included. If $R_{\text{off}}, M_{\text{off}}, M_{\text{on}} = O(n)$ (e.g., [22, 23]), and $P = O(nm \log m)$ (e.g., [33, 42]), this is $O(nP)$, matching n calls to Π_{perm}. Permuting columns of an $\ell \times m$ matrix with $\ell = 2$ can nevertheless double the concrete permutation cost.[9]

For $M_{\text{on}} = O(n)$, the online communication and computation cost is

$$O(C_{\text{BC}} + nm + mM_{\text{on}}) = O(C_{\text{BC}} + nm).$$

Computing $\langle \beta \mathbf{x} \rangle$ costs $O(mM_{\text{on}})$. Computing and passing the length-$2m$ messages, broadcasting $\mathbf{y}_n$, and locally compressing residuals cost $O(C_{\text{BC}} + nm)$. In Verify, the $n-1$ values c_i are opened to the corresponding verifiers, whose $n-1$ two-field-element broadcasts form one parallel-broadcast invocation and contribute $O(C_{\text{BC}})$. The public challenge openings, blinding multiplication, and final opening add $O(C_{\text{BC}} + M_{\text{on}})$, giving the claimed bound.

For round complexity, let ρ_{perm} be the round complexity of a batched Π_{perm}, counting the other MPC primitives and ideal broadcast as $O(1)$ rounds. The n permutation calls and the power computations for different i run in parallel, giving $O(\rho_{\text{perm}} + \log m)$ offline rounds, or $O(\rho_{\text{perm}} + 1)$ if using a specialized protocol for computing c_i^{m-1}. Online messages pass sequentially from P_1 to P_n, after which the verifier broadcasts run in parallel and Verify takes $O(1)$ further ideal-hybrid rounds. The online round complexity is therefore $O(n)$.

The retained correlation contains $O(n)$ shared length-m vectors and $O(n)$ scalars; each party also stores an $O(m)$ local permutation and mask. Thus preprocessing storage is $O(nm)$ words per party and $O(n^2 m)$ words in aggregate.

6.6 Intuition behind Security

The full security proof is provided in the full version. Here we provide a proof sketch to offer intuition. We note that the security of input $\mathbf{x}$ is rather straightforward, as it is only used in the online phase, and is hidden behind fresh random

[9] In some constructions of Π_{perm} (e.g., [42]), the communication is $O(nm \log m + nm\ell)$. Hence, the actual growth of offline communication could be even milder, if the $nm \log m$ term dominates.

masks. The main challenge is to show that the secret permutation π_i of every honest party P_i remains protected even if the protocol aborts. The intuition is that the adversary's complete view before Verify is independent of all honest permutations and β, even if corrupted parties send arbitrary messages. Without knowing β, the adversary who has sent wrong messages to honest parties will fail the final check with high probability. Moreover, the final check opens only one blinded aggregate, so an abort reveals neither an individual residual nor its index.

Theorem 1. *Against a static adversary corrupting any set $T \subsetneq [n]$, the final protocol statistically UC-realizes the malicious shuffle functionality in the $\mathcal{F}_{\mathrm{MPC}}$-hybrid model. Except with probability $O(nm/|\mathbb{F}|)$, acceptance implies the prescribed shuffled output; whether the protocol accepts or aborts, its view reveals no information about the input or any honest party's permutation beyond what the ideal functionality reveals.*

Proof sketch. The simulator samples the offline correlation using a dummy zero input and identity permutations for honest parties. Fresh masks make the entire message-phase view identical to the real one, even under arbitrary corrupted-party messages.

After all messages and verifier broadcasts are fixed, β remains uniform from the adversary's perspective. An invalid hidden relation survives the two polynomial compressions only with probability $O((m+n)/|\mathbb{F}|)$. A different message or verifier broadcast that nevertheless satisfies its hidden relation determines β and therefore succeeds with probability at most $1/|\mathbb{F}|$. A union bound gives $O(nm/|\mathbb{F}|)$.

Outside this event, acceptance enforces the prescribed messages at every honest boundary and the prescribed final output. On rejection, the opened value is uniform nonzero and leaks no failing index. The simulator therefore reproduces both accepting and aborting executions, and the remaining view is the semi-honest one-time-pad simulation. □

7 Experiments

7.1 Experimental Setup

We implement our shuffle protocol using the permutation protocol of Song et al. [42] and the MP-SPDZ [30] MPC framework. The MP-SPDZ framework supports the Mascot protocol [31], an arithmetic MPC protocol based on additive secret sharing. Mascot supports the multiplication operations required by our protocol, and is secure against malicious adversaries corrupting a majority of parties. We implement the permutation protocol and shuffle protocol of Song et al. [42] (called the "basic shuffle protocol" below), then implement our shuffle protocol based on it and compare the two. The goal of our experiment is to evaluate whether our construction significantly improves the online communication and computation complexity compared to the basic shuffle protocol, as suggested by our theoretical analysis.

We note that we use Mascot only as a representative backend; both protocols are implemented in the same framework for a fair comparison.

The reported experiments follow the authentication order in Section 8.1, which hides the permutation even upon abort.

We run our experiments on Ubuntu 24.04 with 32 physical CPU cores (64 hardware threads) on two Intel(R) Xeon(R) Gold 6326 processors at 2.90 GHz. Each party is simulated as a separate process over loopback. From one local execution per configuration, we model the running time as local computation time plus communication time and round-trip latency. Unless stated otherwise, we use 80 MB/s bandwidth and 60 ms RTT.

We set the security parameter to $\kappa = 40$ and choose a 64-bit prime field for the Mascot protocol. The basic shuffle protocol of [42] has a parameter trading communication for computation, due to the sub-permutation decomposition of [10]. We therefore report two configurations: one optimized for minimal online running time and the other for minimal total running time. Since the online phase of our protocol is unaffected by the choice of basic permutation protocol, we report results for the configuration with the minimal overall running time.

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

7.2 Experiments with Number of Parties

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	$[42]^1$	160	395	728	1000	1213	30.0	110.0	220.9	398.8	594.0
	$[42]^2$	133	317	491	639	813	46.4	168.3	381.3	655.6	1050.5
	ours	451	1178	2007	2873	3798	56.5	200.2	432.6	791.6	1219.8
Online	$[42]^1$	11.3	27.9	55.1	75.7	91.8	24.1	63.1	157.1	242.4	354.8
	$[42]^2$	7.47	17.7	27.3	35.3	45.0	14.9	35.8	61.6	87.8	118.9
	ours	0.263	0.329	0.351	0.362	0.369	1.05	1.25	1.43	1.62	1.83

n is the number of parties. The number of items is $m = 2^{12}$.
$[42]^1$ is optimized to minimize total running time.
$[42]^2$ is optimized to minimize online running time.

We first evaluate the protocols with $n = 3, 6, 9, 12, 15$ parties. This test aims to verify that 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 this experiment are listed in Table 2.

From the table, it can be seen that both the online communication and running time of our protocol grow significantly more slowly than those of the basic protocol. One significant phenomenon is that the online communication overhead per party is almost invariant for our protocol, which aligns with the

theoretical result that the online communication should be $O(C_{\mathrm{BC}} + nm)$. It can also be seen that the basic protocol's trade-off has inherent limitations, in the sense that even if we choose a parameter that completely ignores the offline cost, its online running time remains slower than that of our protocol. This is because the basic shuffle protocol has inherent complexity $\Theta(Bn^2m)$, which cannot be mitigated by increasing offline computation overhead.

Against the basic protocol optimized for total time, our total time ratio is 1.06×–1.29× across $n = 3, \ldots, 15$. Against its online-optimized configuration, our online speedup grows from 14.2× to 65.0×, reflecting the reduction from $\Theta(Bn^2m)$ to $O(C_{\mathrm{BC}} + nm)$ online work.

7.3 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 to evaluate the scalability of the protocol with respect to increasing data volume. The results are listed in Table 3.

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	[42]^1	21.2	67.7	336	1692	5932	5.18	23.1	60.2	222.1	1178.1
	[42]^2	20.8	67.7	262	1243	4684	8.83	23.0	152.4	554.0	1654.2
	ours	53.5	220	559	2399	8142	12.8	29.4	100.8	364.4	1646.3
Online	[42]^1	1.11	3.80	25.7	134	428	9.14	8.11	25.9	102.5	214.2
	[42]^2	0.951	3.80	13.9	79.7	281	7.80	8.11	8.24	39.3	56.3
	ours	0.0493	0.197	0.787	3.15	12.6	0.973	0.980	0.995	1.08	1.43

m is the number of items. The number of parties is $n = 2$.
[42]^1 is optimized to minimize total running time.
[42]^2 is optimized to minimize online running time.
At $m = 2^{18}$, one candidate parameter for our protocol was skipped due to memory exhaustion; the best completed candidate is reported.

From the table, we can observe that both online communication and running time of our protocol are significantly lower than those of the basic protocols. From $m = 2^{16}$ to $m = 2^{18}$, an $O(m \log m)$ offline term with fixed decomposition parameters predicts a 4.5× increase. Optimizing the decomposition parameter separately at each point yields measured increases of 3.39×–3.77× across the three configurations. Our online communication grows by exactly 4×, as expected from its dominant $O(nm)$ term for fixed n. Across the size sweep, our total time ratio to the total-time optimized basic protocol is 0.96×–1.18×, while our online speedup over the online-optimized configuration reaches 39.33×. This advantage arises from the factor B introduced by the cut-and-choose technique used in the basic protocol. The performance gap becomes more pronounced as

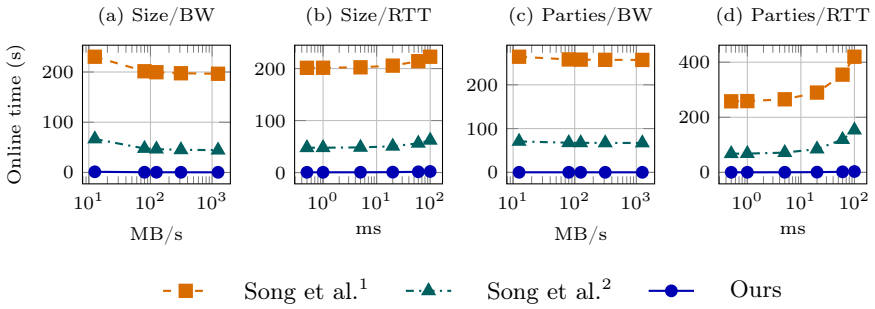

Fig. 3. SPDZ online time under varying network conditions. Bandwidth panels fix the RTT to 0.5 ms; RTT panels fix bandwidth to 80 MB/s. Song et al.[1] is optimized for total time, while Song et al.[2] is optimized for online time.

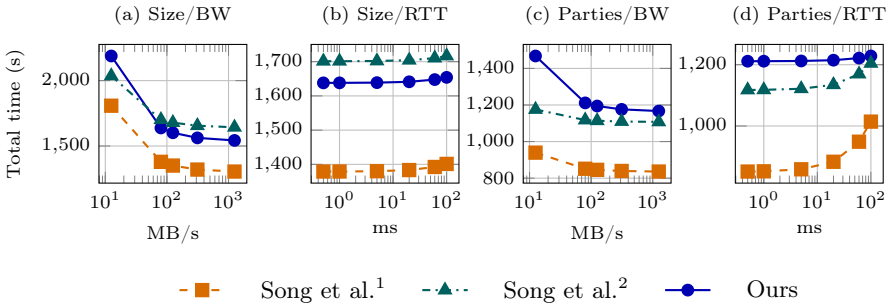

Fig. 4. SPDZ total time under varying network conditions. Bandwidth panels fix the RTT to 0.5 ms; RTT panels fix bandwidth to 80 MB/s. Song et al.[1] is optimized for total time, while Song et al.[2] is optimized for online time.

the number of items grows. For small inputs, computation rather than communication remains the primary bottleneck; as m grows, the communication gap becomes more pronounced.

7.4 Sensitivity to Network Conditions

We here present results under varying bandwidth and RTT. The size panels use $(n, m) = (2, 2^{18})$ and the party panels $(n, m) = (15, 2^{12})$; each point uses the decomposition parameter that minimizes the stated target.

The online advantage persists across all network settings and becomes especially pronounced with many parties. The total-time plots show the corresponding preprocessing trade-off. The plots varying bandwidth primarily reflect the protocols' communication volume, whereas those varying RTT primarily reflect their round complexity. Notably, our protocol is less sensitive to increasing RTT. This is because its shuffle correlations require only parallel invocations of Π_{perm},

allowing the online phases of these invocations to be executed in batch and thereby reducing the overall number of online rounds. In the implementation, as each party is simulated by one single-threaded process, executing the online phase in batch (instead of in parallel) reduces only the number of rounds and does not otherwise accelerate the execution.

8 Discussions and Extensions

8.1 Real-world Instantiation

Our protocol and security proof use the ideal $\mathcal{F}_{\mathrm{MPC}}$ and authenticated-broadcast functionalities. In the following we discuss their instantiation in practice.

Authenticated broadcast. Authenticated parallel broadcast with unanimous abort ensures that all honest parties either agree on a vector containing one message per party, preserving every honest sender's message, or unanimously abort. When all parties follow the protocol, they receive the message vector. This suffices for our protocols, whose ideal model permits adversarially triggered global abort through Π_{abort}. Protocol 5 uses one parallel broadcast in Verify and one single-sender broadcast of a length-$2m$ vector. The latter can be implemented by distributing the vector and certifying its digest, incurring $O(nm)$ payload communication and certification overhead asymptotically dominated by that of the parallel broadcast.

The cost of such a parallel broadcast primitive varies by setting. In a generic dishonest-majority setting where up to $n-1$ parties can be corrupted, assuming synchronous communication, a public-key infrastructure, and collision-resistant hashing, such a primitive can be implemented using $O(n^3)$ communication via the certificate reduction to Dolev–Strong broadcast [21, Proposition 2], suppressing security-parameter factors. In this case, our malicious protocol incurs $O(n^3 + nm)$ online communication.

Authenticated openings. Song et al. [41] show that a preliminary version of our protocol is vulnerable under a direct SPDZ-like instantiation. Their attack exploits the gap between the ideal Π_{open} and a tentative SPDZ opening [18]: after injecting an error into a protocol message, the adversary subtracts a guessed permutation-dependent error from its local share of the verification value. The tentative value opens to zero exactly when the guess is correct, even though the subsequent MAC check rejects in either case.

This attack concerns a concrete realization outside our $\mathcal{F}_{\mathrm{MPC}}$-hybrid proof, but it must be addressed in an implementation. At the end of Verify, an SPDZ-like backend must use the order

$$\begin{aligned}\langle f\rangle \text{ computed } &\longrightarrow \Pi_{\mathrm{check}} \longrightarrow \text{abort or proceed}\\ &\longrightarrow \text{open } \langle f\rangle \longrightarrow \text{abort or proceed}\\ &\longrightarrow \Pi_{\mathrm{check}} \longrightarrow \text{abort or proceed.}\end{aligned}$$

Here Π_{check} is one batched check covering every prior tentative opening and unchecked multiplication. If it fails, the protocol aborts without revealing f. Only after it succeeds do the parties open f and authenticate this opening with the final check. Failure detected in any of these three steps causes immediate abort. The opening of f must still be checked because a rushing adversary may alter its opening share after Π_{check}.

The first Π_{check} guarantees that the unrevealed f is computed correctly. If it passes, f is not manipulated, which makes its disclosure safe. The last Π_{check} is used to prevent the adversary from directly forging a null opening regardless of the real value of f. Together, these steps ensure that the disclosure of f is safe, and that the opened value correctly indicates whether there was misbehavior.

8.2 Reusing Permutations

As mentioned in Section 1, some real-world applications require reusing the same random permutation π multiple times in sequence. They may also require shuffling with the inverse permutation π^{-1}, so that the data are returned to their original positions. Examples of such applications include distributed graph analysis tasks [4, 8, 34] and distributed data analysis tasks [5].

This requires several enhancements to the shuffle protocol. First, the concrete backend must follow the required authentication order in Section 8.1, so that the secret random permutation π is not leaked upon abort.

Second, the parties must be able to generate correlations for the same π. Assume each party P_i samples a secret random permutation π_i. As Π_{perm} supports batched permutation, replacing the invocations

$$\Pi_{\text{perm}}(P_i : \pi_i, \langle \mathbf{r}_i, \mathbf{r}'_i \rangle) \text{ by } \Pi_{\text{perm}}(P_i : \pi_i, \langle \mathbf{r}_{i,0}, \mathbf{r}'_{i,0}, \ldots, \mathbf{r}_{i,b-1}, \mathbf{r}'_{i,b-1} \rangle)$$

and modifying the other corresponding parts in Algorithm 4 allow the parties to generate b shuffle correlations for the same π.

If the parties must be able to generate shuffle correlations for π persistently, e.g., because the maximum number of required shuffles cannot be determined a priori, we can append each $\Pi_{\text{perm}}(P_i : \pi_i, \ldots)$ invocation with a public vector $(0, 1, \ldots, m-1)$. These public vectors, which become secret-shared after shuffling, can be used to test each other and check whether party P_i has honestly used the same π_i across sessions. When every party uses the same π_i across sessions, the generated correlations correspond to the same π. This can be used to test whether two batches of correlations correspond to the same permutation π, allowing correlations for π to be generated persistently.

Finally, if correlations of π^{-1} are also needed, they can be generated as follows. The parties first generate a correlation for π, where each invocation of $\Pi_{\text{perm}}(P_i : \pi_i, \ldots)$ is appended with a public vector $(0, 1, \ldots, m-1)$[10]. After generation, we obtain a corresponding vector for P_i

$$\langle \mathbf{v}_i \rangle \leftarrow \pi_i(0, 1, \ldots, m-1),$$

[10] Any public vector with distinct entries will do.

which is secret-shared and protected. The parties then generate π^{-1}, where each party P_i uses π_i^{-1} as its input, and the parties apply the permutations in reverse order, i.e., $\pi'_n, \pi'_{n-1}, \ldots, \pi'_1$. Again, each invocation of $\Pi_{\text{perm}}(P_i : \pi'_i, \ldots)$ is appended with $\langle \mathbf{v}_i \rangle$, after which we obtain $\langle \mathbf{v}'_i \rangle$. Then the parties open all $\mathbf{v}'_i$ and check

$$\mathbf{v}'_i = \pi'_i(\pi_i((0, 1, \ldots, m-1))) \stackrel{?}{=} (0, 1, \ldots, m-1),$$

which must be true if P_i has honestly used π_i^{-1} in the second session. Moreover, opening $\mathbf{v}'_i$ cannot reveal the secret permutation π_i of an honest party P_i because the two permutations cancel each other out. In a real-world instantiation, these openings must follow the authenticated opening discipline above so that they do not create a selective-abort channel.

We note that this allows the parties to reuse π, but not shuffle correlations. Correlations must be treated as one-time pads and should be discarded after use.

8.3 Extending to Small Fields and Rings

Small fields. If $nm/|\mathbb{F}|$ is not negligible, the polynomial checks used in Section 6 no longer provide the desired concrete security. The parties can instead choose an extension field $\mathbb{K}/\mathbb{F}$ such that

$$|\mathbb{K}| \geq nm2^{\kappa},$$

embed the input in $\mathbb{K}$, and execute the shuffle protocol over $\mathbb{K}$. In particular, the shuffle authentication key, all masks and correlations, the multiplications, and the openings used by the checks are defined over $\mathbb{K}$. The total soundness error is then $O(nm/|\mathbb{K}|) = O(2^{-\kappa})$. Mask cancellation leaves the final result in the embedded copy of $\mathbb{F}$, so reducing it back to $\mathbb{F}$ yields the intended shuffled vector. This extension requires the underlying MPC functionality and permutation protocol to support arithmetic and permutation over $\mathbb{K}$.

Power-of-two rings. The construction can also be adapted to semantic values in $R = \mathbb{Z}_{2^k}$ by following the lifted authentication technique of SPDZ$_{2^k}$ [14]. Let s be the statistical security parameter and define the larger ring $\widehat{R} = \mathbb{Z}_{2^{k+s}}$. A semantic value $x \in R$ is represented by a lifted value $\widehat{x} \in \widehat{R}$ whose low k bits equal x. The shuffle authentication key is sampled as $\beta \leftarrow \mathbb{Z}_{2^s}$ and embedded in $\widehat{R}$. The hidden relation is maintained in the larger ring as

$$\beta \widehat{\mathbf{y}}_i^1 \equiv \widehat{\mathbf{y}}_i^2 + \pi_i(\widehat{\mathbf{r}}'_i) \pmod{2^{k+s}}.$$

All shuffle masks are sampled in $\widehat{R}$, and every correlation and verification equation is computed modulo 2^{k+s}. Openings are exact in $\widehat{R}$; the output denotes its reduction modulo 2^k, with its upper bits freshly randomized.

An output deviation is harmful only if it changes the value modulo 2^k. Factoring a harmful error into a power of two and an odd unit shows that a successful forgery determines β modulo 2^s, and therefore succeeds with probability at most

2^{-s}. We replace the field polynomial test by private random linear combinations and secretly input the verifier summaries. Combining with the lifted authentication argument, this gives error at most

$$2^{-s+\log_2(s+1)}.$$

The full version gives the modified protocol and proof, with $O(C_{\mathrm{BC}}+nm)$ online communication and computation.

We note that this extension requires that the underlying functionality supports addition, multiplication, authenticated opening, and permutation over $\widehat{R}$. In particular, this requirement is not automatically met by permutation protocols or Shamir-style backends designed only for fields.

8.4 Approximate Vector Length

When the input length is unknown during preprocessing, $\mathrm{Shuffle}_{\mathrm{off}}$ can use an upper bound m. The parties pad the input to length m and append to each element an indicator bit identifying dummy entries. They shuffle these pairs, open the shuffled indicators, and discard marked entries. This requires Π_{perm} to support vectors of elements, replacing vectors by matrices in the protocol. The same padding technique applies to arrays of vectors whose lengths are known only up to an upper bound.

9 Conclusion

We formalize the permute-in-turn paradigm and give generic transformations from semi-honest and malicious MPC permutation protocols to shuffle protocols with linear online communication and computation. Instantiating them with [42] and [33] yields the first maliciously secure linear-online shuffles for additive and Shamir secret sharing, respectively. We prove security in the $\mathcal{F}_{\mathrm{MPC}}$-hybrid model and experimentally confirm significant online savings over prior shuffle protocols.

Acknowledgments. This work was supported by the Natural Science Foundation on Frontier Leading Technology Basic Research Project of Jiangsu (No. BK20222001), the NSFC-62272215, AIQ funding of Nanjing University, and the 111 Center (No. B26023).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Abe, M.: Mix-networks on permutation networks. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 258–273. Springer (1999)
2. Adida, B., Wikström, D.: 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. pp. 555–574. Springer (2007). https://doi.org/10.1007/978-3-540-70936-7_30
3. Alexopoulos, N., Kiayias, A., Talviste, R., Zacharias, T.: Mcmix: Anonymous messaging via secure multiparty computation. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1217–1234 (2017)
4. Araki, T., Furukawa, J., Ohara, K., Pinkas, B., Rosemarin, H., Tsuchida, H.: Secure graph analysis at scale. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security. pp. 610–629. Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3460120.3484560, https://doi.org/10.1145/3460120.3484560
5. Asharov, G., Hamada, K., Ikarashi, D., Kikuchi, R., Nof, A., Pinkas, B., Takahashi, K., Tomida, J.: Efficient secure three-party sorting with applications to data analysis and heavy hitters. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 125–138. Association for Computing Machinery (2022). https://doi.org/10.1145/3548606.3560691, https://doi.org/10.1145/3548606.3560691
6. Bater, J., Elliott, G., Eggen, C., Goel, S., Kho, A., Rogers, J.: Smcql: Secure querying for federated databases (2017). https://doi.org/10.14778/3055330.3055334
7. Beerliová-Trubíniová, Z., Hirt, M.: Perfectly-secure mpc with linear communication complexity. In: Theory of Cryptography Conference. pp. 213–230. Springer (2008). https://doi.org/10.1007/978-3-540-78524-8_13
8. Brüggemann, A., Koti, N., Kukkala, V.B., Schneider, T.: MultiCent: Secure and scalable computation of centrality measures on multilayer graphs. Proceedings on Privacy Enhancing Technologies Symposium **2025**(4), 944–968 (2025). https://doi.org/10.56553/popets-2025-0166
9. Canetti, R.: Universally composable security: A new paradigm for cryptographic protocols. In: Proceedings 42nd IEEE Symposium on Foundations of Computer Science. pp. 136–145. IEEE (2001). https://doi.org/10.1109/SFCS.2001.959888
10. Chase, M., Ghosh, E., Poburinnaya, O.: 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. pp. 342–372. Springer (2020). https://doi.org/10.1007/978-3-030-64840-4_12
11. Chaum, D.L.: Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM **24**(2), 84–90 (1981). https://doi.org/10.1145/358549.358563
12. Chen, K., Pang, B., Li, Y., Wang, M.: Secure multi-party shuffling with optimal communication. The Computer Journal p. bxaf043 (2025)
13. Chow, S.S.M., Lee, J.H., Subramanian, L.: Two-party computation model for privacy-preserving queries over distributed databases. In: Proceedings of the Network and Distributed System Security Symposium. Internet Society (2009)

14. Cramer, R., Damgård, I., Escudero, D., Scholl, P., Xing, C.: SPDZ2k: Efficient MPC mod 2^k for dishonest majority. Cryptology ePrint Archive, Paper 2018/482 (2018), https://eprint.iacr.org/2018/482
15. Dalskov, A., Escudero, D., Keller, M.: Fantastic four: Honest-Majority Four-Party secure computation with malicious security. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 2183–2200. USENIX Association (Aug 2021), https://www.usenix.org/conference/usenixsecurity21/presentation/dalskov
16. Damgård, I., Fitzi, M., Kiltz, E., Nielsen, J.B., Toft, T.: Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. In: Theory of Cryptography Conference. pp. 285–304. Springer (2006). https://doi.org/10.1007/11681878_15
17. Damgård, I., Nielsen, J.B.: Scalable and unconditionally secure multiparty computation. In: Annual International Cryptology Conference. pp. 572–590. Springer (2007). https://doi.org/10.1007/978-3-540-74143-5_32
18. Damgård, I., Pastro, V., Smart, N., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: Annual Cryptology Conference. pp. 643–662. Springer (2012). https://doi.org/10.1007/978-3-642-32009-5_38
19. Escudero, D., Ghosh, S., Keller, M., Rachuri, R., Scholl, P.: 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. pp. 823–852. Springer (2020). https://doi.org/10.1007/978-3-030-56880-1_29
20. Eskandarian, S., Boneh, D.: Clarion: Anonymous communication from multiparty shuffling protocols. In: Network and Distributed System Security (NDSS) Symposium (2022). https://doi.org/10.14722/ndss.2022.24141
21. Fitzi, M., Gottesman, D., Hirt, M., Holenstein, T., Smith, A.: Detectable byzantine agreement secure against faulty majorities. In: Proceedings of the Twenty-First Annual Symposium on Principles of Distributed Computing. p. 118–126. PODC '02, Association for Computing Machinery, New York, NY, USA (2002). https://doi.org/10.1145/571825.571841, https://doi.org/10.1145/571825.571841
22. Gordon, S.D., Le, P.H., McVicker, D.: Linear communication in malicious majority mpc. In: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security. pp. 2173–2187. CCS '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3576915.3623162, https://doi.org/10.1145/3576915.3623162
23. Goyal, V., Song, Y., Zhu, C.: Guaranteed Output Delivery Comes Free in Honest Majority MPC. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology – CRYPTO 2020. pp. 618–646. Springer International Publishing, Cham (2020). https://doi.org/10.1007/978-3-030-56880-1_22
24. Groth, J.: A verifiable secret shuffle of homomorphic encryptions. Journal of Cryptology **23**(4), 546–579 (2010). https://doi.org/10.1007/s00145-010-9067-9
25. Hamada, K., Ikarashi, D., Chida, K., Takahashi, K.: Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation. Cryptology ePrint Archive (2014)
26. Hamada, K., Kikuchi, R., Ikarashi, D., Chida, K., Takahashi, K.: 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. pp. 202–216. Springer (2013). https://doi.org/10.1007/978-3-642-37682-5_15
27. Han, F., Lan, X., Liu, W., Zhang, L., Ren, H., Qu, L., Hong, Y.: Concretely efficient correlated oblivious permutation. Cryptology ePrint Archive (2025)

28. Hohenberger, S., Rothblum, G.N., Shelat, A., Vaikuntanathan, V.: Securely obfuscating re-encryption. In: Theory of Cryptography Conference. pp. 233–252. Springer (2007). https://doi.org/10.1007/978-3-540-70936-7_13
29. Kanagavelu, R., Li, Z., Samsudin, J., Yang, Y., Yang, F., Mong Goh, R.S., Cheah, M., Wiwatphonthana, P., Akkarajitsakul, K., Wang, S.: Two-phase multiparty computation enabled privacy-preserving federated learning. In: 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID). pp. 410–419 (2020). https://doi.org/10.1109/CCGrid49817.2020.00-52
30. Keller, M.: 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
31. Keller, M., Orsini, E., Scholl, P.: Mascot: faster malicious arithmetic secure computation with oblivious transfer. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. pp. 830–842 (2016). https://doi.org/10.1145/2976749.2978357
32. Keller, M., Pastro, V., Rotaru, D.: Overdrive: Making spdz great again. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 158–189. Springer (2018)
33. Keller, M., Scholl, P.: Efficient, oblivious data structures for MPC. In: Sarkar, P., Iwata, T. (eds.) Advances in Cryptology–ASIACRYPT 2014, Part II. Lecture Notes in Computer Science, vol. 8874, pp. 506–525. Springer (2014). https://doi.org/10.1007/978-3-662-45608-8_27
34. Koti, N., Kukkala, V.B., Patra, A., Raj Gopal, B.: Graphiti: Secure graph computation made more scalable. In: Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security. pp. 4017–4031. Association for Computing Machinery (2024). https://doi.org/10.1145/3658644.3670393, https://doi.org/10.1145/3658644.3670393
35. Laud, P.: Linear-time oblivious permutations for spdz. In: Cryptology and Network Security: 20th International Conference, CANS 2021, Vienna, Austria, December 13-15, 2021, Proceedings 20. pp. 245–252. Springer (2021). https://doi.org/10.1007/978-3-030-92548-2_13
36. Laud, P., Pettai, M.: Secure multiparty sorting protocols with covert privacy. In: Secure IT Systems: 21st Nordic Conference, NordSec 2016, Oulu, Finland, November 2-4, 2016. Proceedings 21. pp. 216–231. Springer (2016). https://doi.org/10.1007/978-3-319-47560-8_14
37. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: Information Security: 14th International Conference, ISC 2011, Xi'an, China, October 26-29, 2011. Proceedings 14. pp. 262–277. Springer (2011). https://doi.org/10.1007/978-3-642-24861-0_18
38. Mohassel, P., Rindal, P.: Aby3: A mixed protocol framework for machine learning. In: Proceedings of the 2018 ACM SIGSAC conference on computer and communications security. pp. 35–52 (2018)
39. Mugunthan, V., Polychroniadou, A., Byrd, D., Balch, T.H.: Smpai: Secure multiparty computation for federated learning. In: Proceedings of the NeurIPS 2019 Workshop on Robust AI in Financial Services. vol. 21. MIT Press Cambridge, MA, USA (2019)
40. Peceny, S., Raghuraman, S., Rindal, P., Shah, H.: Efficient permutation correlations and batched random access for two-party computation. In: IACR International Conference on Public-Key Cryptography. pp. 76–109. Springer (2025)

41. Song, X., Liang, X., Dong, Y., Bai, J., Duan, P., Dong, C., Zhang, T., Chang, E.C.: Secret-shared shuffle from authenticated correlations. In: IACR International Conference on Public-Key Cryptography. pp. 321–354. Springer (2026)
42. Song, X., Yin, D., Bai, J., Dong, C., Chang, E.C.: Secret-shared shuffle with malicious security. Network and Distributed System Security (NDSS) Symposium (2024). https://doi.org/10.14722/ndss.2024.24021
43. Sotthiwat, E., Zhen, L., Li, Z., Zhang, C.: Partially encrypted multi-party computation for federated learning. In: 2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid). pp. 828–835. IEEE (2021). https://doi.org/10.1109/CCGrid51090.2021.00101
44. Volgushev, N., Schwarzkopf, M., Getchell, B., Varia, M., Lapets, A., Bestavros, A.: Conclave: secure multi-party computation on big data. In: Proceedings of the Fourteenth EuroSys Conference 2019. pp. 1–18. EuroSys '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3302424.3303982