# *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 $C_{\mathrm{BC}}$ 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 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 $\pi$, each party $P_i$ locally chooses a private random permutation $\pi_i$. After securely applying each permutation $\pi_i$ 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 $\pi_i$ to the data, the parties typically rely on an MPC permutation protocol $\Pi_{\text{perm}}$ as a primitive. As will be discussed in more detail in Section 2, this simple idea and the primitive $\Pi_{\text{perm}}$ 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 $\Pi_{\text{perm}}$ $n$ 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 $\Pi_{\text{perm}}$ 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. 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.

[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.

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 $C_{\mathrm{BC}}$ 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 $\Pi_{\mathrm{perm}}$, enabling future work to optimize permutation protocols independently.

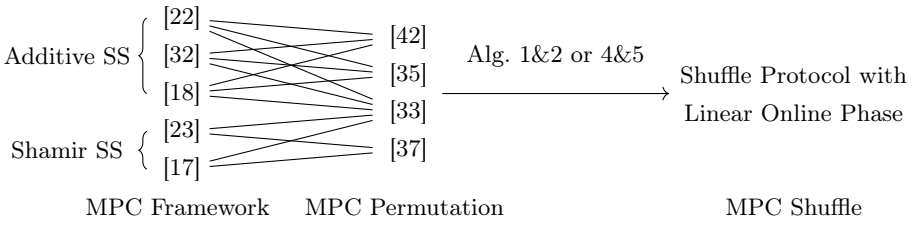


**Fig. 1.** Illustration of our transformation: any compatible pair consisting of an MPC framework and a permutation protocol can be transformed into an MPC shuffle protocol with linear online complexity.

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.
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

[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.

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 $\mathcal{F}_{\text{MPC}}$-hybrid model, guaranteeing composability across MPC frameworks. Here $\mathcal{F}_{\text{MPC}}$ 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. The full formal proofs are provided in Appendices B and C, and Shamir secret sharing-based experiments in Appendix E. We extend our construction to $\mathbb{Z}_{2^k}$ ring in Appendix F.

## 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, $C_{\mathrm{BC}}$ 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 Appendix A.3. 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 $\kappa$. 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 $\pi_2, \pi_1$ 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 $(\text{id}, x)$. "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 $(\text{id}, y)$.
- $\Pi_{\text{rand}}(\text{id})$, which draws a uniform random value $r \in \mathbb{F}$ and stores it as $(\text{id}, r)$.
- $\Pi_{\text{add}}(\text{id}, \text{id}_1, C)$, which retrieves $(\text{id}_1, x)$ 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 $(\text{id}_1, x_1)$ and $(\text{id}_2, x_2)$ from the memory and stores $(\text{id}, x_1 + x_2)$.
- $\Pi_{\text{mul}}$ works the same as $\Pi_{\text{add}}$, except it computes multiplications.
- $\Pi_{\text{open}}(\text{id})$, which retrieves $(\text{id}, x)$ 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 $(\text{id}, x)$ 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 $(\text{id}_j, x_j)$ 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 $C_{\text{BC}}$ 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 $\Pi_{\text{mul}}$. $\mathcal{F}_{\text{MPC}}$ 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 $\Pi_{\text{perm}}$ 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 $\Pi_{\text{abort}}$ 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 $\pi$, 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 $\Pi_{\text{perm}}$, 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 $\pi_i$, 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 $\pi$ is not known to any party. In addition, as long as at least one honest party $P_i$ has chosen its permutation $\pi_i$ uniformly at random, the resulting $\pi$ will be uniformly random. Hence, due to the adversary lacking access to the $\pi_i$ chosen by honest parties, the resulting $\pi$ 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 $\Pi_{\text{perm}}$ 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 $\Pi_{\text{perm}}$, 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 $\pi_i$ 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, $\pi_i$ 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 Section C.

**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 $\Pi_{\text{Shuffle}}$. This command takes an agreed shared vector $\langle \mathbf{x} \rangle$ and a private $m$-permutation $\pi_i$ 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* $\Pi_{\text{Shuffle}}$.

*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* $\Pi_{\text{perm}}$ *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 in Section C.*

$\mathcal{E}$ *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* $\Pi_{\text{Shuffle}}$, *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 $\pi_i$ or the composed permutation $\pi$. 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 Section C.

# 4 Technical Overview

## 4.1 High-Level Idea

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

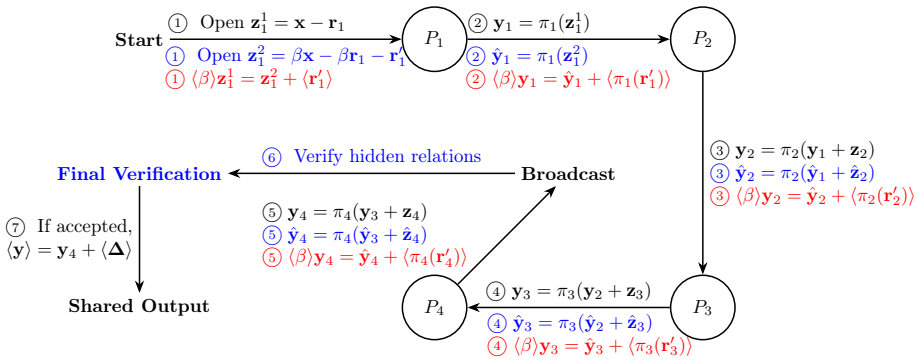


**Fig. 2.** Overview of online phases for 4 parties. Black steps are common to both settings; blue steps are added for malicious security; red text marks the hidden relation expected between $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$. In the malicious setting, the shared output is produced only after the 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 $\pi_i$ 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 $\beta$ 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 $\Pi_{\text{perm}}$?
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 $\pi_i$. 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 $\beta$-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 $\beta$ 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 $\beta$ correctly, which succeeds with probability at most $1/|\mathbb{F}|$. Section 6.6 gives a security proof sketch; the full proof is provided in Appendix C.

## 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. *$\pi_i$ 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 $\pi_i$.
**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 Appendix B.

## 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. *$\pi_i$ 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 $\beta$ plays a role similar to that of the MAC key in the SPDZ framework (which is usually denoted by "$\alpha$" 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 $\Pi_{\mathrm{perm}}$, 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 $\pi_i$.
**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 $\Pi_{\text{mul}}$.
    $\langle d_i \rangle \leftarrow \sum_{j=1}^m \langle c_i^{j-1} \rangle \cdot \langle \pi_{i-1}(\mathbf{r}'_{i-1})(j) \rangle$
  **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 $\Pi_{\text{mul}}$ 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 $\pi_i$ means permuting both halves of the $2m$-length vector by $\pi_i$.

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 $\Pi_{\text{open}}$ 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 $\Pi_{\text{rand}}$ (resp. $\Pi_{\text{mul}}$), and $P$ the total cost of $\Pi_{\text{perm}}$. All randomness is preprocessed ($R_{\text{on}} = 0$); online $\Pi_{\text{rand}}$ 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 $\Pi_{\text{perm}}$. 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 $\rho_{\text{perm}}$ be the round complexity of a batched $\Pi_{\text{perm}}$, 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 Appendix C. Here we provide a proof sketch to offer intuition. We note that the security of input $\mathbf{x}$ is rather straightforward,

[9] In some constructions of $\Pi_{\text{perm}}$ (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.

as it is only used in the online phase, and is hidden behind fresh random masks. The main challenge is to show that the secret permutation $\pi_i$ 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 $\beta$, even if corrupted parties send arbitrary messages. Without knowing $\beta$, 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, $\beta$ 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 $\beta$ 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

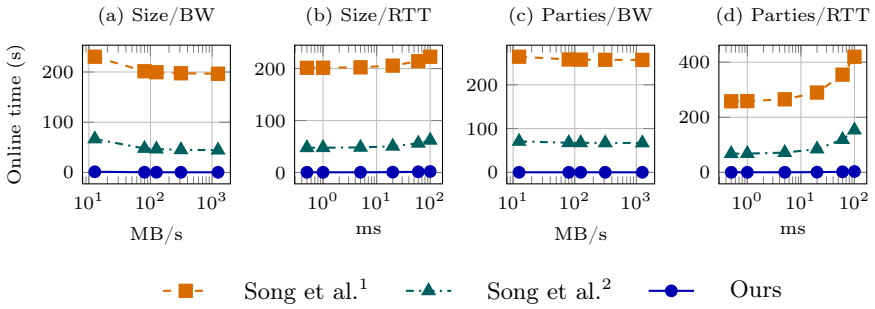


**Fig. 3.** SPDZ online time under varying network conditions. Bandwidth panels fix the RTT to 0.5 ms; RTT panels fix bandwidth to 80 MB/s. Song et al.[1] is optimized for total time, while Song et al.[2] is optimized for online time.

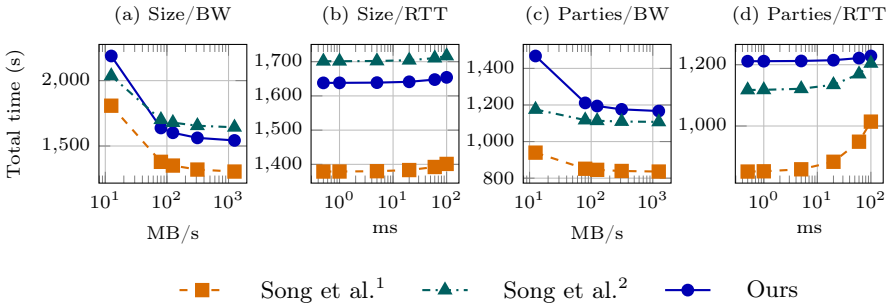


**Fig. 4.** SPDZ total time under varying network conditions. Bandwidth panels fix the RTT to 0.5 ms; RTT panels fix bandwidth to 80 MB/s. Song et al.[1] is optimized for total time, while Song et al.[2] is optimized for online time.

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 $\Pi_{\text{perm}}$,

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 $\Pi_{\mathrm{abort}}$. 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 $\Pi_{\mathrm{open}}$ 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 $\Pi_{\text{check}}$ 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 $\Pi_{\text{check}}$.

The first $\Pi_{\text{check}}$ guarantees that the unrevealed $f$ is computed correctly. If it passes, $f$ is not manipulated, which makes its disclosure safe. The last $\Pi_{\text{check}}$ 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 $\pi$ multiple times in sequence. They may also require shuffling with the inverse permutation $\pi^{-1}$, 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 $\pi$ is not leaked upon abort.

Second, the parties must be able to generate correlations for the same $\pi$. Assume each party $P_i$ samples a secret random permutation $\pi_i$. As $\Pi_{\text{perm}}$ 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 $\pi$.

If the parties must be able to generate shuffle correlations for $\pi$ 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 $\pi_i$ across sessions. When every party uses the same $\pi_i$ across sessions, the generated correlations correspond to the same $\pi$. This can be used to test whether two batches of correlations correspond to the same permutation $\pi$, allowing correlations for $\pi$ to be generated persistently.

Finally, if correlations of $\pi^{-1}$ are also needed, they can be generated as follows. The parties first generate a correlation for $\pi$, 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 $\pi^{-1}$, where each party $P_i$ uses $\pi_i^{-1}$ 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 $\pi_i^{-1}$ in the second session. Moreover, opening $\mathbf{v}'_i$ cannot reveal the secret permutation $\pi_i$ 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 $\pi$, 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 $\beta$ 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

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

Appendix F 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 $\Pi_{\mathrm{perm}}$ 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

# A Case Studies

In this section, we present several concrete protocols generated by instantiating our construction with existing MPC permutation protocols from [42], [33] and [37], respectively. The first construction applies only to additive secret sharing, and is highly efficient, due to utilizing possibly currently most efficient MPC permutation protocol. The second construction works for arbitrary MPC framework supporting MPC multiplication and has a relatively simple implementation. The third construction may be adapted to any threshold secret sharing scheme, but requires ZK proofs in its offline phase to achieve malicious security.

Despite all these differences, all three constructions are actively secure and have $O(C_{\mathrm{BC}} + nm)$ online complexity, due to our malicious-secure shuffle correlation.

## A.1 Shuffle Protocol for Additive Secret Sharing

The construction of Song et al. [42] is based on SPDZ framework [18], where a variable stored at MPC is additively secret-shared among all parties, along with an additional MAC. That is, the statement "a variable $a$ is secret-shared as $\langle a \rangle$" means that each party $P_i$ holds locally a tuple $(a_i, \gamma_i(a))$ such that

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

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

Basically, this two-party permutation protocol is built from repeating basic CGP protocol [10] for $B$ times with additional correctness checks, while $B$ is a parameter required for applying cut-and-choose technique. If the receiver has chosen permutation $\pi_1, ..., \pi_B$ respectively in these sessions, the overall effect is that the array is permuted by a single permutation $\pi$:

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

This two-party permutation protocol is then extended to the case of $n$-party by letting a fixed party $P_i$ act as the receiver in all $(n-1) \times B$ sessions, which can be interpreted as permuting each party's shares with the same set of $B$ permutations. Of course, additional steps are taken in [42] to prevent $P_i$ from acting inconsistently in sessions with different parties, e.g. choosing different $B$ permutations when the sender is different. The protocol then checks if each session of the CGP protocol finishes correctly, to guarantee that no misbehavior happens so far.

To conclude, the work of Song et al. [42] provides us with exactly the functionality

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

with an overhead of $O(Bnmt \log m)$ communication and computation complexity. By instantiating with the SPDZ framework of [22] and fitting above permutation protocol into Algorithm 4, we obtain a shuffle protocol with $O(Bn^2 m \log m)$ offline communication and computation complexity, whose online complexity is $O(C_{\mathrm{BC}} + nm)$.

One intricate detail regarding implementation is that at the end of the shuffle protocol, it must be guaranteed that the array is correctly shuffled. This arises in applications where the shuffle is followed by a data opening, such as the Clarion anonymous communication system designed by [20], which opens all values after the shuffle. However, SPDZ itself does not guarantee safe opening, i.e. before the MAC check passes, the correctness of all opened values is not guaranteed. This means that $\Pi_{\mathrm{open}}$ does not guarantee a safe open (which we have assumed throughout), which means that all values during the shuffle protocol, including those used in verification, could be wrong, and the correctness is not guaranteed even if the protocol doesn't abort. Consequently, a concrete SPDZ instantiation must follow the required authentication order of Section 8.1: it computes $\langle f \rangle$, batch-authenticates all prior tentative operations, and only then invokes an authenticated opening of $f$ and acts on its value. Our $O(C_{\mathrm{BC}} + nm)$ bound is stated in the $\mathcal{F}_{\mathrm{MPC}}$-hybrid model; a concrete SPDZ implementation must additionally account for these backend authentication steps.

We note that this gives the first maliciously secure shuffle protocol for additive secret sharing with $O(C_{\mathrm{BC}} + nm)$ online complexity.

### A.2 Shuffle Protocol for Shamir Secret Sharing

The construction of permutation protocol in [33] works for arbitrary MPC framework supporting MPC multiplication. Assuming MPC multiplication can be done within $O(n)$ communication and computation, its implementation of $\Pi_{\mathrm{perm}}$ requires $O(nm \log m)$ communication and computation in total.

The permutation protocol developed in [33] utilizes a specific permutation network, which is a switching network of size $O(m \log m)$ and of depth $O(\log m)$. Such a network consists of $O(\log m)$ layers of switches, each switch randomly swapping two entries of the vector depending on an extra control bit. As a permutation network is capable of representing any permutation by $O(m \log m)$ switches, this allows parties to represent a permutation by $O(m \log m)$ bits.

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

Such an implementation matches exactly our requirements for $\Pi_{\mathrm{perm}}$. By instantiating $\Pi_{\mathrm{perm}}$ in Algorithm 4 with such implementation, we immediately

obtain a shuffle protocol for arbitrary MPC framework supporting MPC multiplication. The offline phase consists of $O(nm \log m)$ many multiplications, which results in $O(n^2 m \log m)$ communication, computation and $O(\log m)$ rounds, assuming the communication and computation overhead of each multiplication are both linear. By instantiating with the Shamir secret sharing scheme in [23], we obtain a concrete construction that meets above complexities. The above implementation can also be instantiated with SPDZ framework of [22], with $O(n^2 m \log m)$ offline complexities that is asymptotically better than instantiating with the permutation protocol of [42]. Nevertheless, since the primitives used in [42] might be cheaper than MPC multiplication, which construction is more concretely efficient may depend on concrete implementation.

We note that this gives the first shuffle protocols for Shamir secret sharing with $O(C_{\mathrm{BC}} + nm)$ online complexity, for both semi-honest and malicious security.

### A.3 Shuffle Protocol instantiated with [37]

The construction of Laur et al. [37] works for threshold secret sharing, with possibly malicious adversaries. By its design, each party belongs to several groups, and each group of parties will agree on a common permutation and permute the array once. The size of each group is large enough, so that it is possible to reconstruct all secret with the shares of the group member, which allows the group to permute the array trivially by each member permuting the shares locally and re-distributing their shares. For $(n/2)$-threshold secret sharing, there will be asymptotically $O(2^n/\sqrt{n})$ many groups, and by design at least one of the groups consists of only honest parties. Hence, since this all-honest group permutes the array with a uniform random permutation known only to its group members, the overall permutation is hidden from the adversary. The advantage of this construction is that, when the number of parties are small and all parties are semi-honest, it does not utilize any public key primitives and is extremely fast. Also, most of the other MPC shuffle protocols works only with additive secret sharing (e.g. [10] [35] [20] [42]), while the construction of [37] works with threshold secret sharing, e.g. Shamir secret sharing.

It is possible to extend our construction, so that the shuffle protocol may be instantiated by the protocol of Laur et al. [37]. For clarity, suppose we are working in semi-honest case. Suppose we have $t$ such groups

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

by the design of [37], with $G_i \subsetneq \{P_i\}_{i \in [n]}$. All that needs to be done is to replace the $\Pi_{\mathrm{perm}}$ functionality by

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

where $G \in \mathcal{G}$, i.e. $\pi$ is known only to members of $G$. The rest of the protocol proceeds naturally by treating each $G_i$ as an individual participant, i.e. generating and permuting random vectors with each permutation $\pi_i$ known only to

members of $G_i$, opening $\mathbf{z}_i$ to only members of $G_i$, etc. The online phase hence consists of $O(2^n/\sqrt{n})$ rounds and has $O(2^n m/\sqrt{n})$ communication complexity. Note that instead of repeating this same process on all members of $G_i$, each group $G_i$ can elect a party in the group as its agent, who will permute the masked vector $\mathbf{y}_{i-1}$, add the vector $\mathbf{z}_i$ and send $\mathbf{y}_i$ to agent of the next group $G_{i+1}$.

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

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

we can sort it as

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

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

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

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

This is simplified to compute

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

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

This can be also extended to malicious security, with the same technique used earlier in Section 6. Note that we only need to check if the final result of agent $P_i$ is correct, since all other computations are done internal $P_i$. Hence, the malicious online communication and computation are $O(C_{\mathrm{BC}} + nm)$, including the final verifier broadcasts. Note that as the construction of [37] utilizes ZK proof for malicious security, it is likely that the construction instantiated with [42] or [33] would be more efficient in practice. Nevertheless, if one wishes to build their MPC application with Shamir secret sharing or replicated secret sharing (instead of additive secret sharing), the construction of [42] will not be available, and instantiating shuffle protocol with the construction of [37] might be a good choice.

## B Security Proof for Semi-honest Secure Shuffle

Recall the simulation-based security definition from Definition 1. For a static corrupted set $T \subsetneq [n]$, let $\mathrm{view}_{\mathcal{A}}$ contain the corrupted parties' inputs and random tapes, all messages they receive, and their prescribed outputs in the execution of $\mathrm{Shuffle}_{\mathrm{off}}$ followed by $\mathrm{Shuffle}_{\mathrm{on}}$.

**Theorem 2.** *The semi-honest two-phase shuffle protocol* $\mathrm{Shuffle}_{\mathrm{off}}$ *and* $\mathrm{Shuffle}_{\mathrm{on}}$ *perfectly realizes* $\mathcal{F}_{\mathrm{shuffle}}$ *against static semi-honest adversaries corrupting up to* $n-1$ *parties in the* $\mathcal{F}_{\mathrm{MPC}}$*-hybrid model.*

*Proof.* We construct a simulator $\mathcal{S}$ that uses the corrupted parties' permutations and simulates the protocol with an arbitrary dummy input and arbitrary dummy permutations for the honest parties. Replacing $\mathbf{x}$ induces a bijection on the uniform masks: adjust $\mathbf{r}_1$ to preserve $\mathbf{x} - \mathbf{r}_1$, and then adjust each later $\mathbf{r}_i$ recursively to preserve $\mathbf{z}_i$. Hence it does not change the view distribution. The following bijection shows the analogous statement for the honest permutations.

The core idea is that, fixing the adversary's view and $\{\pi_i\}_{i\in\bar{T}}$ of permutations chosen by the honest parties, for any arbitrary permutations $\{\pi'_i\}_{i\in\bar{T}}$, there is exactly one assignment to $(\mathbf{r}_i)_{i\in[n]}$ and $(\mathbf{z}_i)_{i\in\bar{T}}$, such that the adversary's view is not altered, yet the output of the protocol is $\mathbf{x}$ shuffled with $\{\pi'_i\}_{i\in\bar{T}}$ instead of $\{\pi_i\}_{i\in\bar{T}}$.

For a clear demonstration, assume that the adversary corrupts exactly $n-1$ parties, leaving $P_k$ the only honest party. We even grant $\mathbf{x}$ to the adversary, which only strengthens its view. The adversary's view consists of

$$\mathrm{view}_{\mathcal{A}} = \left\{ \begin{array}{llllll} \mathbf{x} & & & & & \\ \pi_1 & \pi_2 & \cdots & \pi_{k-1} & \pi_{k+1} & \cdots & \pi_n \\ \mathbf{z}_1 & \mathbf{z}_2 & \cdots & \mathbf{z}_{k-1} & \mathbf{z}_{k+1} & \cdots & \mathbf{z}_n \\ \mathbf{y}_1 & \mathbf{y}_2 & \cdots & \mathbf{y}_{k-1} \; \mathbf{y}_k & \mathbf{y}_{k+1} & \cdots & \mathbf{y}_n \end{array} \right\}.$$

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

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

1. For each $i \neq k$, $\pi'_i = \pi_i$.
2. For each $i \neq k$, $\mathbf{z}_i = \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i = \pi'_{i-1}(\mathbf{r}'_{i-1}) - \mathbf{r}'_i = \mathbf{z}'_i$.
3. For $P_k$, $\mathbf{z}'_k = \pi'_{k-1}(\mathbf{r}'_{k-1}) - \mathbf{r}'_k$, which may differ from the original $\mathbf{z}_k$.
4. The output will be

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

Note that the above statements imply that the view of corrupted parties remains unchanged by replacing $\pi_k$ with arbitrary $\pi'_k$, and even remains consistent with its newly given semantics (i.e. shuffling with $\pi'_i$). Hence, since $\pi'_k$ is an arbitrary permutation, this indicates that any permutation $\pi'_k$ is equally possible in the view of adversary.

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

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

Since $\mathbf{y}_1$ appears in the view, it demands $\mathbf{z}'_1 = \pi_1'^{-1}(\mathbf{y}_1)$. Luckily, $\mathbf{z}_1$ and $\mathbf{z}'_1$ does not appear in the view, hence they can be different. This uniquely determines $\mathbf{r}'_1 = \mathbf{x} - \pi_1'^{-1}(\mathbf{y}_1)$. Also, observe that

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

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

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

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

which follows trivially from the fact that

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

and

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

and $\pi_2 = \pi'_2$.

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

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

And at the end, party $P_n$ obtains

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

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

Hence, if $P_1$ is honest, by seeing only the values in the view, $\pi_1$ can be any permutation $\pi'_1$. And the probabilities of $\pi_1$ being any $\pi'_1$ are equal, since there

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

The argument for $k \geq 2$ proceeds similarly. For $k \geq 2$, all $\mathbf{r}_i$ for $i < k$ are fully determined by the adversary's view and hence cannot be altered. However, since the adversary cannot observe $\mathbf{z}_k$, we retain the flexibility to adjust $\mathbf{r}_k$ and $\pi_k$ arbitrarily.

The general case where $\mathcal{A}$ corrupts an arbitrary subset $T \subsetneq [n]$ follows by analogous reasoning, which we omit here for brevity. The bijection is measure preserving because all masks are uniform. Therefore $\mathcal{S}$ can generate a perfectly distributed real-world view without the honest parties' permutations, while $\mathcal{F}_{\text{shuffle}}$ stores the prescribed shuffled output. Hence the real and ideal executions are identically distributed. □

# C Security Proof for Maliciously Secure Shuffle

## C.1 Roadmap and Ideal Functionality

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

First recall that the offline phase $\text{Shuffle}_{\text{off}}$ of the shuffle protocol essentially performs only randomness generation, multiplications, calls to $\Pi_{\text{perm}}$, and openings. Since all these operations are supported by $\mathcal{F}_{\text{MPC}}$, the security of offline phase is immediate.

In the online phase $\text{Shuffle}_{\text{on}}$, the potential security risk stems from message $\mathbf{y}_i$. Since $\mathbf{z}_i$ and $\pi_i$ are by design known only to party $P_i$, honest parties cannot tell locally whether what $P_i$ sends is indeed $\mathbf{y}_i = \pi_i(\mathbf{z}_i + \mathbf{y}_{i-1})$. Every receiver therefore retains its incoming message. After all $\mathbf{y}_i$ are fixed, one call to Verify checks all hidden $\beta$-correlations through a single blinded aggregate.

The proof first shows that the complete message-phase view is independent of the input and honest permutations, even when corrupted parties send arbitrary messages. It then proves that $\beta$ remains hidden until all verification broadcasts are fixed. Consequently, a different but correlation-valid message requires guessing $\beta$, while an invalid correlation is rejected by the aggregate check. The opened aggregate reveals no failing residual or index. Finally, a simulator reproduces the message phase and the single final accept-or-abort outcome, which gives UC security.

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

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

**Definition 5.** *(Ideal Functionality of* $\text{Shuffle}_{\text{off}}$*) Denote by* $\mathcal{F}_{\text{off}}$ *an ideal functionality, which is* $\mathcal{F}_{\text{MPC}}$ *equipped with an additional (ideal) command* $\Pi_{\text{off}}$*. This*

*command takes as input a public length $m$ and each $m$-permutation $\pi_i$ from $P_i$, aborts if $\pi_i$ is not a valid $m$-permutation or parties do not agree on $m$, and otherwise generates a shuffle correlation. That is,*

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

*which satisfies Definition 4.*

**Definition 6.** *(Ideal Functionality of Shuffle) Denote by $\mathcal{F}_{\mathrm{shuffle}}$ an ideal functionality, which is the ideal functionality $\mathcal{F}_{\mathrm{MPC}}$ equipped with an additional command $\Pi_{\mathrm{shuffle}}$. This command takes as input $\langle \mathbf{x} \rangle$ and a private $m$-permutation $\pi_i$ from each party $P_i$. It checks that the parties agree on the input handle and length $m$ and that each $\pi_i$ is valid; otherwise it sends a common abort. For valid inputs it sets $\pi = \pi_n \circ \cdots \circ \pi_1$. The adversary may request a common abort before completion, in which case no output handle is returned and no input or honest permutation is revealed. Otherwise the functionality stores $\pi(\mathbf{x})$ under a fresh handle and returns this shared output handle. Its value is revealed only by a subsequent opening.*

**Remark.** *As $\pi_i$ is a private input of $P_i$, exposing it is unacceptable if $P_i$ is honest, even if the shuffle protocol aborts.*

We first establish the security of the offline phase and the masking property needed for deferred verification. We then analyze the soundness of the verification protocol.

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

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

*Note that since $\mathbf{z}_1$ is not included in the offline phase, it should be understood as a null value if $P_1$ is corrupted.*

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

First, we note that the protocol $\mathrm{Shuffle}_{\mathrm{off}}$ correctly implements ideal functionality $\mathcal{F}_{\mathrm{off}}$ by outputting a correlation that satisfies Definition 4. This follows directly: the protocol samples $\beta, (\mathbf{r}_i, \mathbf{r}'_i)_{i=1}^n$, and $(c_i)_{i=2}^n$ with the distributions in Definition 4, and computes $d_i$ and $\mathbf{z}_i$ by the equations stated there.

Consider a simulator $\mathcal{S}$ that acts as follows. At the start of the game, $\mathcal{E}$ first sends to $\mathcal{F}_{\mathrm{off}}$ the input of honest parties for command $\Pi_{\mathrm{off}}$, which is $\pi_i$ for $P_i$. $\mathcal{S}$ simply follows the protocol, since $\mathcal{A}$ will obtain no output by calling to ideal

$\mathcal{F}_{\text{MPC}}$ in the real world. In the step of calling $\Pi_{\text{perm}}(P_i : \pi_i, ...)$ for corrupted party $P_i$, since in the real world execution $\mathcal{A}$ will need to send $\pi_i$ to $\Pi_{\text{perm}}$, $\mathcal{S}$ will receive (purported) $\pi_i$ from $\mathcal{E}$. Hence, $\mathcal{S}$ can send $\pi_i$ to $\mathcal{F}_{\text{off}}$, as part of the input required by command $\Pi_{\text{off}}$.

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

Note also that during the process, whenever $\mathcal{E}$ demands $\mathcal{S}$ to abort, $\mathcal{S}$ simply aborts as $\mathcal{A}$ does. If the protocol does not eventually abort, $\mathcal{S}$ sends "continue" to $\mathcal{F}_{\text{off}}$, which then returns the shared correlation handles and delivers only the prescribed private correlation values to their recipients. This shuffle correlation is of course consistent with the view of adversary $\mathcal{A}$, as the $\mathbf{z}_i$ and $\pi_i$ are the same. Hence, the above process is perfect, i.e. is identical with the execution in the real world.

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

For a static corrupted set $T$, let $\text{view}_{\mathcal{A}}^{\text{msg}}$ denote the adversary's view immediately after $\mathbf{y}_n$ is fixed and before Verify. It consists of $\{\pi_i, \mathbf{z}_i\}_{i\in T}$, all messages received or sent by corrupted parties, and all public broadcasts up to this point. The adversary's auxiliary input, random tape, and internal state are included implicitly.

**Lemma 2 (Message-Phase Privacy).** *Let $\mathcal{A}$ be any static adversary controlling $T$. Conditioned on its auxiliary input and the corrupted parties' inputs,* $\text{view}_{\mathcal{A}}^{\text{msg}}$ *can be perfectly simulated without* $\mathbf{x}$ *or* $\{\pi_i\}_{i\notin T}$*, even if corrupted parties send arbitrary messages. Equivalently, for any two choices* $(\mathbf{x}, \{\pi_i\}_{i\notin T})$ *and* $(\widetilde{\mathbf{x}}, \{\widetilde{\pi}_i\}_{i\notin T})$ *with identical public and corrupted-party inputs, the corresponding message-phase views are identically distributed.*

*Proof.* Fix $\mathbf{x}$, $\beta$, and all $\pi_i$, and define

$$\begin{aligned} &\mathbf{u}_1 = \mathbf{x} - \mathbf{r}_1, \quad \mathbf{u}_i = \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i, \\ &\mathbf{v}_1 = -\mathbf{r}'_1, \qquad \mathbf{v}_i = \pi_{i-1}(\mathbf{r}'_{i-1}) - \mathbf{r}'_i \quad (i = 2, \ldots, n). \end{aligned}$$

Both maps $(\mathbf{r}_i)_i \mapsto (\mathbf{u}_i)_i$ and $(\mathbf{r}'_i)_i \mapsto (\mathbf{v}_i)_i$ are bijective: their inverses follow recursively from

$$\mathbf{r}_1 = \mathbf{x} - \mathbf{u}_1, \quad \mathbf{r}_i = \pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{u}_i, \quad \mathbf{r}'_1 = -\mathbf{v}_1, \quad \mathbf{r}'_i = \pi_{i-1}(\mathbf{r}'_{i-1}) - \mathbf{v}_i.$$

A bijection maps the uniform distribution to the uniform distribution. Since the $\mathbf{r}_i$ and $\mathbf{r}'_i$ are independent and uniform, all $\mathbf{u}_i, \mathbf{v}_i$ are therefore mutually independent and uniform. Their relation to the online masks is

$$\begin{aligned} \mathbf{z}_1 &= (\mathbf{x} - \mathbf{r}_1, \beta\mathbf{x} - \beta\mathbf{r}_1 - \mathbf{r}'_1) = (\mathbf{u}_1, \beta\mathbf{u}_1 + \mathbf{v}_1), \\ \mathbf{z}_i &= (\pi_{i-1}(\mathbf{r}_{i-1}) - \mathbf{r}_i, \pi_{i-1}(\beta\mathbf{r}_{i-1} + \mathbf{r}'_{i-1}) - \beta\mathbf{r}_i - \mathbf{r}'_i) \\ &= (\mathbf{u}_i, \beta\mathbf{u}_i + \mathbf{v}_i), \qquad i = 2, \ldots, n. \end{aligned}$$

For fixed $\beta$, the last map is bijective, with inverse $(\mathbf{z}_i^1, \mathbf{z}_i^2) \mapsto (\mathbf{z}_i^1, \mathbf{z}_i^2 - \beta \mathbf{z}_i^1)$. Hence every mask tuple has probability $|\mathbb{F}|^{-2mn}$, independently of $\mathbf{x}$ and all permutations. Observing the masks of corrupted parties therefore does not change the distribution of these secrets, and the remaining honest masks stay independent and uniform.

We now proceed in protocol order. A corrupted party's next message is an arbitrary function of its current view. Before an honest $P_i$ acts, its incoming value $\mathbf{t}_i$ is fixed while its $\mathbf{z}_i$ remains fresh and uniform. Hence $\pi_i(\mathbf{t}_i + \mathbf{z}_i)$ is uniform and independent of $\mathbf{x}$ and $\pi_i$. Induction over all parties proves that the two message-phase views are identically distributed. The same argument gives a perfect simulator by using any dummy $\mathbf{x}$ and honest permutations.

The same induction also includes messages between honest parties: retain each incoming message in the induction, and sample the next honest outgoing message uniformly using its fresh mask. Thus the joint distribution of the view and the entire message sequence is independent of $\mathbf{x}$, the honest permutations, and $\beta$. Coupling this sequence and the independent challenges $c_i$ gives identical honest verification broadcasts in both executions; corrupted broadcasts are then generated from identical views. □

**Lemma 3 (Aggregate Verification).** *Define*

$$\mathsf{Valid} \iff \left( \bigwedge_{\substack{i=2,\dots,n \\ i \notin T}} \beta \mathbf{y}_{i-1}^1 = \mathbf{y}_{i-1}^2 + \pi_{i-1}(\mathbf{r}'_{i-1}) \right) \wedge \left( \beta \mathbf{y}_n^1 = \mathbf{y}_n^2 + \pi_n(\mathbf{r}'_n) \right).$$

*Protocol 3 is perfectly complete, in that if* Valid *holds and all parties follow the protocol,* Verify *accepts. Further, if* Valid *does not hold,* Verify *aborts with probability at least* $1-(m+n-1)/|\mathbb{F}|$. *Conditioned on abort,* $f$ *is uniform over* $\mathbb{F} \setminus \{0\}$ *and reveals neither an individual residual nor its index.*

*Proof.* For $i = 2, \dots, n$, define the vector defect

$$\boldsymbol{\delta}_i = \beta \mathbf{y}_{i-1}^1 - \mathbf{y}_{i-1}^2 - \pi_{i-1}(\mathbf{r}'_{i-1}), \tag{15}$$

and define

$$\boldsymbol{\delta}_{n+1} = \beta \mathbf{y}_n^1 - \mathbf{y}_n^2 - \pi_n(\mathbf{r}'_n).$$

An honest verifier computes $\eta_i = \sum_{j=1}^m c_i^{j-1} \boldsymbol{\delta}_i(j)$, while $\eta_{n+1} = \sum_{j=1}^m e^{j-1} \boldsymbol{\delta}_{n+1}(j)$. If every defect is zero and all parties follow the protocol, then every $\eta_i = 0$, so $f = 0$ and verification accepts.

Suppose some defect checked by an honest verifier, or $\boldsymbol{\delta}_{n+1}$, is nonzero. Its challenge is uniform and hidden until the corresponding message is fixed. The resulting nonzero polynomial has degree at most $m-1$, so its scalar residual vanishes with probability at most $(m-1)/|\mathbb{F}|$. Otherwise the vector $(\eta_2, \dots, \eta_{n+1})$ is nonzero. Hence

$$s(X) = \sum_{i=2}^{n+1} \eta_i X^{i-2}$$

is a nonzero polynomial of degree at most $n-1$. Since $\rho$ is sampled after all $\eta_i$ are fixed, $s(\rho) = 0$ with probability at most $(n-1)/|\mathbb{F}|$. If $s(\rho) \neq 0$, acceptance requires $\gamma = 0$, which occurs with probability $1/|\mathbb{F}|$. The claimed bound follows by addition.

Finally, conditioned on rejection, $s(\rho) \neq 0$ and $\gamma \neq 0$. Multiplication by the fixed nonzero $s(\rho)$ maps uniform nonzero $\gamma$ to uniform nonzero $f$. Thus the opened value identifies only that some checked relation failed, and not which residual failed. □

**Remark.** *Lemma 3 establishes only correlation validity. It does not yet rule out a forged message that differs from the prescribed one while still making* Valid *true. The following key-privacy and enforcement arguments show that such a forgery succeeds only with negligible probability.*

### C.2 Privacy of the Shuffle Authentication Key

Let $\text{view}_{\mathcal{A}}^{\text{fix}}$ extend $\text{view}_{\mathcal{A}}^{\text{msg}}$ with the values $c_i$ opened to corrupted verifiers, all broadcasts $(w_{i,1}, w_{i,2})$, and the public challenge $e$. This view is taken after these values are fixed and before the aggregation challenge $\rho$ is sampled.

**Theorem 3 (Pre-Verification Key Privacy).** *Let $U$ be uniform over $\mathbb{F}$ and independent of $\text{view}_{\mathcal{A}}^{\text{fix}}$. Then*

$$(\beta, \text{view}_{\mathcal{A}}^{\text{fix}}) \equiv (U, \text{view}_{\mathcal{A}}^{\text{fix}}).$$

*In particular, the adversary learns no information about $\beta$ before $\rho$ is sampled.*

*Proof.* Lemma 2 shows that the joint message sequence and adversarial view are independent of $\beta$. The challenges $c_i$ are sampled independently offline and remain hidden until the message phase ends; $e$ is sampled independently afterward. An honest verifier's $(w_{i,1}, w_{i,2})$ is a function of its fixed incoming message and $c_i$; a corrupted verifier's broadcast is a function of its current view. Neither operation introduces dependence on $\beta$. Hence $\text{view}_{\mathcal{A}}^{\text{fix}}$ is independent of $\beta$, proving the claim. □

### C.3 Enforcement of Honest Behaviors in the Online Phase

An *effective deviation* is a non-prescribed message delivered to an honest party, a non-prescribed final broadcast, or a non-prescribed $(w_{i,1}, w_{i,2})$ used in verification. Deviations confined between corrupted parties are irrelevant until they affect an honest party or the final output. For example, a malicious $P_i$ could send an arbitrary message $\widehat{\mathbf{y}}_i$ to a malicious $P_{i+1}$. Since the message is passed only inside the corrupted parties, which is not observable to the honest parties, this is not an effective deviation. Here "prescribed" refers to the execution with the fixed offline permutations and no deviations. In particular, the reference verifier broadcast compresses that execution's message, not an altered internal

record supplied by corrupted parties. Internal changes that restore all prescribed observable messages therefore remain irrelevant.

We call the delivery of $\mathbf{y}_i$ to an honest $P_{i+1}$ an *honest boundary*; the final public broadcast is the output boundary. A *corrupted block*, or simply a *block*, is a maximal consecutive sequence $P_a, \ldots, P_b$ of corrupted parties in the shuffle chain. Its entry is the message received from $P_{a-1}$, or the opened $\mathbf{z}_1$ when $a = 1$; its outgoing message reaches an honest boundary when $b < n$, and the output boundary when $b = n$. These notions let us ignore arbitrary internal behavior while tracking its observable effects. Given a block's entry, its prescribed operations can be replayed using only its corrupted parties' permutations and masks. Comparing its outgoing message and verification broadcasts with this replay identifies the first effective deviation without knowing $\mathbf{x}$ or any honest permutation. This is why the enforcement and simulation arguments below reason about blocks and their boundaries.

**Theorem 4 (Defect Enforcement).** *Except with probability $O(nm/|\mathbb{F}|)$, if* Verify *accepts, then the following holds for $i = n$ and every $i < n$ for which $P_{i+1}$ is honest:*

$$(\mathbf{y}_i^1, \mathbf{y}_i^2) = \big(\overline{\pi}_i(\mathbf{x}) - \pi_i(\mathbf{r}_i), \overline{\pi}_i(\beta\mathbf{x}) - \pi_i(\beta\mathbf{r}_i + \mathbf{r}'_i)\big). \tag{16}$$

*Proof.* Consider a non-prescribed message $\mathbf{y}_{i-1}$ delivered to an honest $P_i$. Since $P_i$ retains this exact incoming message, Verify includes the scalar residual obtained from its defect $\boldsymbol{\delta}_i$ defined in (15). Once the verification broadcasts are fixed, this residual is a fixed coefficient of the final aggregate and cannot be changed by any subsequent message.

Consider the first such message, or the non-prescribed final broadcast. It leaves a maximal consecutive block of corrupted parties. Starting from that block's received message (or the opened $\mathbf{z}_1$ if it starts at $P_1$), replaying its prescribed operations determines the reference outgoing pair using only corrupted permutations and masks. Before the first boundary deviation this reference is the prescribed pair. Its difference from the actual pair is consequently a function of the adversary's view, which is independent of $\beta$. If it violates its correlation, Lemma 3 implies that verification aborts except with probability $O((m+n)/|\mathbb{F}|)$. Otherwise both the prescribed pair $(\mathbf{y}^1, \mathbf{y}^2)$ and the forged pair $(\widehat{\mathbf{y}}^1, \widehat{\mathbf{y}}^2)$ satisfy the same correlation. Their difference therefore satisfies

$$\beta(\widehat{\mathbf{y}}^1 - \mathbf{y}^1) = \widehat{\mathbf{y}}^2 - \mathbf{y}^2. \tag{17}$$

If $\widehat{\mathbf{y}}^1 - \mathbf{y}^1$ were zero, the equation would make the other difference zero as well. Hence some coordinate of the first difference is nonzero, and the equation determines $\beta$. By Theorem 3, this succeeds with probability at most $1/|\mathbb{F}|$.

The same argument covers a forged verifier broadcast: relative to the prescribed $(w_{i,1}, w_{i,2})$, a changed pair with zero residual must satisfy $\beta\Delta_1 = \Delta_2$; otherwise it contributes a nonzero scalar residual to the aggregate. A union bound over all parties, together with aggregate soundness, gives $O(nm/|\mathbb{F}|)$. □

**Remark.** *Different from Lemma 3, Theorem 4 establishes that unless* Verify *aborts, $\mathbf{y}_i$ received by honest $P_{i+1}$ is* precisely *the message in Equation* (16).

*This means that the message is not merely satisfying the hidden $\beta$-relation, but is exactly the one prescribed by the protocol.*

**Corollary 1.** *Outside an event of probability $O(nm/|\mathbb{F}|)$, an accepting online execution is equivalent to the prescribed execution at every honest boundary and at the final output. Any effective deviation instead causes the final verification to abort before output is released.*

*Proof.* This follows from Theorem 4 and the verifier-broadcast argument in its proof. □

### C.4 UC Simulator

We are now able to prove the UC security of the shuffle protocol. First, we formally define the two games in the real and ideal worlds, which form the basis for defining the security of the protocol.

**Definition 7 (Real and Ideal Worlds).** *In the real execution of the shuffle protocol, at the start of the game, the environment $\mathcal{E}$ sends $\mathbf{x}$ to $\mathcal{F}_{\text{MPC}}$, which stores it locally as $\langle \mathbf{x} \rangle$. $\mathcal{E}$ also sends $\pi_i$ to honest $P_i$, as $P_i$'s input for protocol* Shuffle*. Then the adversary $\mathcal{A}$ corrupting $T \subsetneq [n]$ begins executing the protocol* Shuffle *with honest parties, while receiving instructions from and sending information to $\mathcal{E}$. At the end of the protocol, if the parties do not abort, the protocol returns its shared output handle. For this comparison experiment, the output is subsequently opened to $\mathcal{E}$, yielding $\pi(\mathbf{x})$ in a prescribed execution. We denote this process by $\mathcal{E} \leftrightarrows \Pi_{\bar{T},\mathcal{A}}$.*

*In the ideal world, $\mathcal{A}$ is replaced by simulator $\mathcal{S}$, which plays the role of all corrupted parties. $\mathcal{F}_{\text{MPC}}$ is equipped with a new command $\Pi_{\text{shuffle}}$, which requires parties to agree on $\langle \mathbf{x} \rangle$ to be shuffled, and takes as input $\pi_i$ from $P_i$. The modified $\mathcal{F}_{\text{MPC}}$ is hence denoted as $\mathcal{F}_{\text{shuffle}}$. At the start of the game, $\mathcal{E}$ sends $\mathbf{x}$ to the ideal functionality $\mathcal{F}_{\text{shuffle}}$. $\mathcal{E}$ also sends $\pi_i$ for honest party $P_i$ to $\mathcal{F}_{\text{shuffle}}$, as part of the inputs for $\Pi_{\text{shuffle}}$. Then $\mathcal{E}$ interacts with $\mathcal{S}$ as if with $\mathcal{A}$, controlling what are sent from corrupted parties and receiving what are received by corrupted parties. If the game does not abort, $\mathcal{E}$ receives the output vector through the same subsequent opening of the shared output handle. If the game aborts, $\mathcal{F}_{\text{shuffle}}$ reveals only the abort and does not reveal any honest party's $\pi_i$ or the composed $\pi$. Denote this process by $\mathcal{E} \leftrightarrows \mathcal{F}_{\bar{T},\mathcal{S}}$, where $\mathcal{F}$ refers to $\mathcal{F}_{\text{shuffle}}$.*

*At the end of each game, $\mathcal{E}$ outputs a single bit, representing its judgement of whether the execution corresponds to the real-world game.*

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

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

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

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

**Theorem 5.** *Assume the protocols work over a field with $|\mathbb{F}| = \Omega(nm2^{\kappa})$, where $\kappa$ is the statistical security parameter, and each party supplies a valid private $m$-permutation input.*

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

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

*Proof.* We construct a simulator for a dummy adversary $\mathcal{A}$, which forwards all messages to $\mathcal{E}$ and delegates all decisions to it. At a high level, the simulator only needs to sample the offline correlation independently of $\mathbf{x}$, leveraging the indistinguishability of $\mathbf{z}_i$, and deduce all corrupted party inputs for the ideal functionality. Below, we detail this simulation.

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

The simulator $\mathcal{S}$ works as follows. In the offline phase of the shuffle, $\mathcal{S}$ first draws $\beta$ uniformly from $\mathbb{F}$. It interacts with $\mathcal{E}$ and obtains the purported input $\pi'_i$ for corrupted party $P_i$. The simulator can obtain $\pi'_i$, because it is the input of corrupted party $P_i$ for $\Pi_{\text{perm}}$, which means that $\mathcal{E}$ needs to send $\pi'_i$ explicitly to $\mathcal{S}$ upon calling $\Pi_{\text{perm}}$. Then $\mathcal{S}$ simulates the output of the offline phase with $\beta$ and $\pi_i$, acts as if each honest $P_i$ has chosen identical permutation. It draws uniformly random vectors $\mathbf{r}_i$, $\mathbf{r}'_i$ and $c_i$, and computes $\mathbf{z}_i$ and $d_i$ as is specified by the protocol. To be specific, $\mathcal{S}$ hence generates and stores locally the entire shuffle correlation (in plaintext)

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

where

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

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

Then in the online phase, $\mathcal{S}$ first simulates the entire message phase, without performing any verification. If $P_1$ is honest, $\mathcal{S}$ sends, on behalf of $P_1$,

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

treating $\mathbf{x}$ as a zero vector. If $P_1$ is dishonest, $\mathcal{S}$ informs $\mathcal{E}$ that $P_1$ has received the above $\mathbf{z}_1$, delegating the decision of $\mathbf{y}_1$ to $\mathcal{E}$. For $i = 2, \ldots, n-1$, an honest $P_i$ stores $\mathbf{y}_{i-1}$ and sends $\mathbf{y}_i = \mathbf{y}_{i-1} + \mathbf{z}_i$ under the simulated identity permutation; a corrupted $P_i$ lets $\mathcal{E}$ choose the message. An honest $P_n$ similarly stores $\mathbf{y}_{n-1}$ and broadcasts $\mathbf{y}_n = \mathbf{y}_{n-1} + \mathbf{z}_n$; if $P_n$ is corrupted, $\mathcal{E}$ chooses this final broadcast.

After all messages are fixed, $\mathcal{S}$ simulates the single call to Verify. For every $i = 2, \ldots, n$, it gives $c_i$ to a corrupted $P_i$ and receives its chosen broadcast $(w_{i,1}, w_{i,2})$. For an honest $P_i$, it computes

$$w_{i,b} = \sum_{j=1}^{m} c_i^{j-1} \mathbf{y}_{i-1}^b(j), \qquad b \in \{1, 2\},$$

from the stored incoming message. Once all these broadcasts are fixed, it draws and opens uniform $e$ and computes

$$\begin{aligned} \eta_i &= \beta w_{i,1} - w_{i,2} - d_i, && i = 2, \ldots, n, \\ \eta_{n+1} &= \sum_{j=1}^{m} e^{j-1} \big(\beta \mathbf{y}_n^1(j) - \mathbf{y}_n^2(j) - \pi_n(\mathbf{r}'_n)(j)\big). \end{aligned}$$

It then draws and opens uniform $\rho$, computes $s = \sum_{i=2}^{n+1} \rho^{i-2} \eta_i$, samples uniform $\gamma$, opens $f = \gamma s$, and reports abort exactly when $f \neq 0$. If the above process does not result in an abort and $\mathcal{E}$ does not request $\mathcal{S}$ to abort, $\mathcal{S}$ instructs $\mathcal{F}_{\text{shuffle}}$ to proceed. The subsequent opening of the output from $\mathcal{F}_{\text{shuffle}}$ sends

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

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

It remains to compare the simulated and real views jointly with the verification outcome. Lemma 2 couples the entire message sequence and adversarial view under the real inputs and under zero input and identity honest permutations. Using the same challenges $c_i, e, \rho$ and adversarial random tape then couples all verification broadcasts and public challenges.

Whether an effective deviation exists is the same in this coupling. To see this, replay each maximal corrupted block from its received message, as in the proof of Theorem 4. Compare its outgoing message and verifier broadcasts with this replay. These comparisons use only the coupled view and corrupted inputs. If

every comparison agrees, induction over the blocks gives all prescribed observable messages and zero residuals. Otherwise the first boundary discrepancy, or a discrepant verifier broadcast when all boundaries agree, is an effective deviation in both executions. Internal changes invisible to honest parties do not affect this criterion.

In either execution, no effective deviation gives $f = 0$ exactly. Corollary 1 bounds the total probability of an effective deviation with $f = 0$ by $O(nm/|\mathbb{F}|)$. For every fixed transcript before opening $f$, its distribution conditional on $f \neq 0$ is uniform over $\mathbb{F} \setminus \{0\}$, by the independent blinding factor $\gamma$. Consequently, replacing $f$ by an independent uniform nonzero value whenever an effective deviation exists changes each execution by statistical distance at most $O(nm/|\mathbb{F}|)$. After this replacement, both the transcript and the accept-or-abort outcome have identical distributions in the two executions. Any adversarial abort is reproduced from the same visible prefix, including the value of $f$ before its delivery to honest parties.

Except for these bounded events, acceptance releases the prescribed $\pi(\mathbf{x})$ in the real execution and the same value from $\mathcal{F}_{\text{shuffle}}$ in the ideal execution; abort releases no output. This proves the required joint simulation. Since $|\mathbb{F}| = \Omega(nm2^{\kappa})$, the distinguishing advantage is $O(nm/|\mathbb{F}|) = O(2^{-\kappa})$.

□

## D Existing Shuffle Correlation

We note that before our work, the concept of shuffle correlation has already been proposed by Eskandarian and Boneh [20] and Chen et al. [12]. As their constructions are similar in many aspects, e.g. both are based on the share translation protocol of Chase et al. [10], both are specialized for additive secret sharing, both are semi-honest only, etc., below we focus on reviewing the construction of Eskandarian and Boneh.

As Song et al. [42] point out, the construction of [20] is vulnerable to selective abort attack. In this section, we briefly review the construction of [20] and the attack of [42], and compare them with our construction.

### D.1 Semi-honest Shuffle Correlation of [20]

The basic idea of [20] is similar to our idea discussed in Section 4. In the online phase, after opening $\mathbf{x} + \mathbf{r}$ to $P_1$, each party $P_i$ receives a masked vector $\mathbf{y}_{i-1}$ from $P_{i-1}$, applies its own permutation $\pi_i$ on it, adds a random mask $\mathbf{r}_i$, and then sends to $P_{i+1}$

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

Following this idea, the main goal of the offline phase is to generate

$$[\![\mathbf{\Delta}]\!] = \pi_n(\pi_{n-1}(\cdots \pi_1([\![\mathbf{r}]\!]) + [\![\mathbf{r}_1]\!] \cdots) + [\![\mathbf{r}_{n-1}]\!]) + [\![\mathbf{r}_n]\!].$$

It is shown by Eskandarian and Boneh [20] that the above shuffle correlation can be generated in $O(1)$ rounds, with the help of a two-party share translation

protocol [10]. The share translation protocol is a two-party protocol that outputs $(\pi_i, \mathbf{\Delta}_{i,j})$ to party $P_i$ and $(\mathbf{a}_{i,j}, \mathbf{b}_{i,j})$ to party $P_j$, such that $\pi_i$ is the input of $P_i$ and $\mathbf{\Delta}_{i,j} = \pi_i(\mathbf{a}_{i,j}) + \mathbf{b}_{i,j}$. Given such a primitive, [20] defines their shuffle correlation as follows.

**Definition 8 (Shuffle Correlation, from [20]).** *The $n$-party shuffle correlation is a situation where for $i = 1, \ldots, n$, $P_i$ holds:*

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

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

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

Note that this $\mathbf{\Delta}_n$ has a form very close to the one we mentioned above, and indeed, they are equivalent up to some simple reformations. To generate $\mathbf{\Delta}_n$ in $O(1)$ rounds, [20] further specifies

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

Wherein each $(\mathbf{a}_{i,j}, \mathbf{b}_{i,j}, \mathbf{\Delta}_{i,j})$ is generated by invoking the share translation protocol of [10] for $P_i$ and $P_j$. To compute $\mathbf{a}'_i$, each party $P_i$ shares each $[\![\mathbf{a}_{j,i}]\!]$, $[\![\mathbf{b}_{j,i}]\!]$, $[\![\mathbf{\Delta}_{i,j}]\!]$, which requires in addition $O(n^3 m)$ communication for sharing $O(n^2)$ random values.

As seen above, it is clear that the definition of shuffle correlation in [20] is specialized for additive secret sharing, and that its generation relies on the two-party share translation protocol of [10]. Their construction requires an expensive translation from two-party randomness to $n$-party randomness, which results in additional $O(n^3 m)$ communication and computation overhead, making the overall complexity $O(n^2 m \log m + n^3 m)$. This complexity is suboptimal, as executing the share translation protocol of [10] for $n^2$ times requires only $O(n^2 m \log m)$ complexities.

In contrast to this approach, our definition (c.f. Definition 3) for semi-honest shuffle correlation is designed for and compatible with a general MPC framework. Specifically, we define[11]

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

which we then implement as

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

[11] Note that the definition of $\mathbf{s}$ is slightly different from the $\mathbf{\Delta}$ above. The two definitions are equivalent up to some reformations.

Clearly, this definition and implementation is based on $n$-party randomness, which avoids the usage of two-party randomness and thus avoids the $O(n^3m)$ complexity term. Furthermore, it also offers a clear approach for generating the correlation. As now $\mathbf{s} = \pi_n(\mathbf{r}_n)$, we can easily parallelize the invocations to $\Pi_{\text{perm}}$ and accelerate the correlation generation.

Thus, besides its simplicity and compatibility, our definition also achieves a better offline communication and computation complexity compared to [20] in semi-honest additive secret sharing setting — the secret sharing scheme for which [20] has been specialized.

### D.2 Malicious Security and Selective Abort Attack

To achieve malicious security, [20] suggests that parties generate and share in addition a secret random vector $\mathbf{g}$ and compute and share (via MPC multiplication) $\mathbf{h}$, such that $\mathbf{h}(i) = \mathbf{x}(i) \cdot \mathbf{g}(i)$ (i.e. Hadamard product). By shuffling $\mathbf{g}, \mathbf{h}$ along with $\mathbf{x}$, and checking in the end if $\mathbf{h}'(i) = \mathbf{x}'(i) \cdot \mathbf{g}'(i)$ for all entry $i$, it is suggested that the protocol is secure against malicious adversary.

However, as Song et al. [42] point out, this construction is vulnerable to selective abort attack. For example, suppose $P_1$ and $P_3$ are corrupted. The adversary may choose random indices $i$ and $j$, and let $P_1$ send $\mathbf{y}_1 + \boldsymbol{\delta}_i$ instead of $\mathbf{y}_1$, where $\boldsymbol{\delta}_i$ is a one-hot vector with its $i$-th entry being 1. After receiving $\mathbf{y}_2$ from $P_2$, $P_3$ first adjusts it by computing $\mathbf{y}_2 - \boldsymbol{\delta}_j$, and acts honestly hereafter. If $P_2$ happens to choose a permutation $\pi_2$ with $\pi_2(j) = i$ (with non-negligible probability $1/m$), the behavior of $P_3$ corrects the error introduced by $P_1$. Hence, if parties do not abort in the final check, the adversary learns $\pi_2(j) = i$ without being detected.

At first glance, it seems quite easy to defend against such an attack: simply check if the Hadamard product relation remains inside each message $\mathbf{y}_i$ sent by $P_i$, and everything seems to work out. However, such a check is costly. To protect the underlying data, the message $P_{i+1}$ receives from $P_i$ is masked with uniform random mask, which prevents $P_{i+1}$ from locally checking its correctness. If $P_{i+1}$ wants to check if the message is correct, i.e. if the Hadamard product relation is maintained, $P_{i+1}$ needs to share the entire message back to all parties, and then all parties, after jointly removing the random masks, carry out $O(m)$ MPC multiplications over all $O(m)$ entries. — This is already $O(n^2m)$ online communication and computation, as an MPC multiplication operation typically costs $O(n)$ overheads, which is repeated for $n$ parties over $m$ entries.

Fortunately, we observe that this overhead can be avoided by carefully designing the correlation embedded in the messages. Suppose the message sent from $P_i$ to $P_{i+1}$ is $\mathbf{y}_i$. We now require $P_i$ to send in addition a message $\hat{\mathbf{y}}_i$ that acts as a MAC, which satisfies

$$\beta \mathbf{y}_i = \hat{\mathbf{y}}_i + \mathbf{s}_i,$$

with $\langle \beta \rangle$ and $\langle \mathbf{s}_i \rangle$ stored at $\mathcal{F}_{\text{MPC}}$. Due to lacking $\beta$ and vector $\mathbf{s}_i$, $P_i$ will not be able to forge wrong message satisfying such relation. After $P_n$ broadcasts $\mathbf{y}_n$ and $\hat{\mathbf{y}}_n$, all parties jointly check all previously sent $\mathbf{y}_i$ and $\hat{\mathbf{y}}_i$, and abort if any check fails.

The protocol up here still requires $O(n^2m)$ communication and computation, due to $P_i$ broadcasting the $\mathbf{y}_{i-1}$ it receives, and all parties jointly computing $\langle\beta\rangle\mathbf{y}_i - \hat{\mathbf{y}}_i - \mathbf{s}_i$.

These checks are compressed and aggregated only after all online messages are fixed. Each receiver obtains a hidden preprocessed challenge, broadcasts two scalar compressions of its stored message, and thereby defines one secret-shared residual. A fresh public challenge then combines all residuals into one polynomial evaluation, which is blinded before opening. The total communication and computation are $O(C_{\mathrm{BC}} + nm)$.

Before this final verification, the masked message-phase view is independent of the input and honest permutations. An invalid relation is rejected except with polynomial soundness error, while a different relation-valid message requires guessing $\beta$. The single opened aggregate reveals no failing index. Thus the protocol either aborts without leaking the input or an honest permutation, or completes with the prescribed $\langle\pi(\mathbf{x})\rangle$.

## E Additional Experiments with Shamir Secret Sharing

In this section, we provide additional experiments for shuffle protocols instantiated with Shamir secret sharing. We compare the shuffle protocol of Keller and Scholl [33] with our shuffle protocol instantiated from the same underlying permutation protocol. We report the performance under both semi-honest and malicious security. We run the experiment under the same setting as in Section 7. Communication is reported per party and in MB and running time is reported in seconds. Our code is available at `https://github.com/GJCPP/KellerShamirShuffle`.

### E.1 Sensitivity to Network Conditions

We additionally evaluate how bandwidth and RTT affect the running time of the Shamir instantiations. We fix $n = 3$ and use $m = 2^{16}$ for semi-honest security and $m = 2^{12}$ for malicious security. Figures 5 and 6 report the network sensitivity of the online and total times.

Our online phase remains faster across all bandwidth and RTT settings. In the semi-honest setting, the total times are nearly identical at low RTT, and our protocol becomes faster as RTT increases. This is because our protocol invokes $\Pi_{\mathrm{perm}}$ in parallel, which reduces the round complexity. In the malicious setting, its increased offline cost dominates the total time despite the smaller online time.

### E.2 Experiments with Number of Parties

We first evaluate the impact of the number of parties. For the semi-honest setting, we fix the number of items to $m = 2^{16}$ and vary the number of parties as $n = 3, 6, 9, 12, 15$. The results are listed in Table 4.

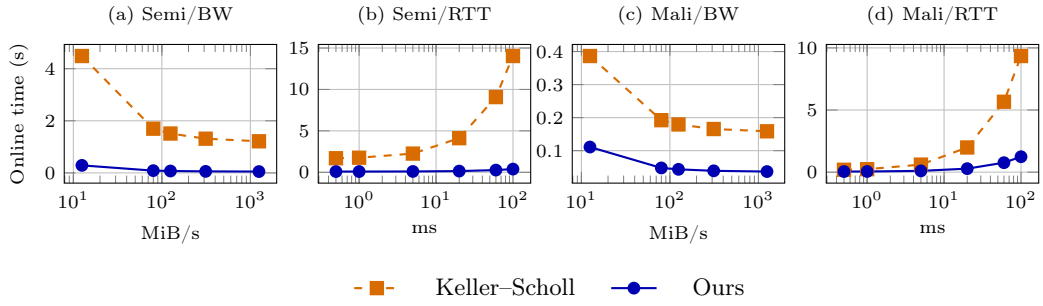


**Fig. 5.** Online time under varying network conditions for the semi-honest and malicious Shamir shuffles. Bandwidth panels fix the RTT to 0.5 ms, and RTT panels fix the bandwidth to 80 MiB/s. Each point is the arithmetic mean of three raw executions.

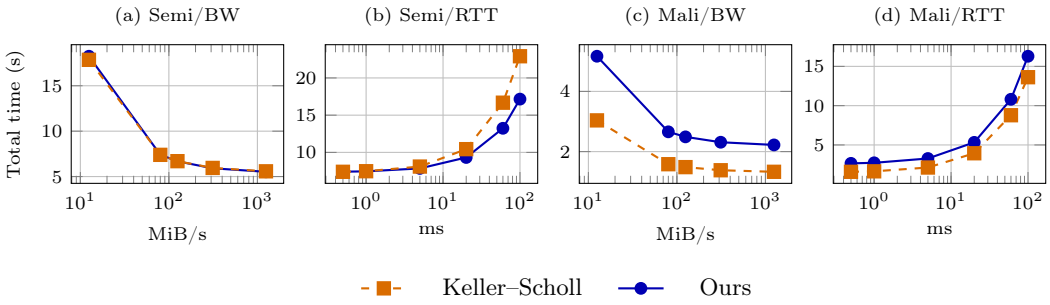


**Fig. 6.** Total time under varying network conditions for the semi-honest and malicious Shamir shuffles. Bandwidth panels fix the RTT to 0.5 ms, and RTT panels fix the bandwidth to 80 MiB/s. Each point is the arithmetic mean of three raw executions.

From Table 4, we can observe that our online communication per party remains almost unchanged as the number of parties increases. When $n$ grows from 3 to 15, the online communication of [33] increases from 43.3 MB to 243 MB, while that of our protocol only increases from 1.40 MB to 1.96 MB. This is consistent with the theoretical result that the online phase of our protocol avoids the dominant $n^2m$ term of the underlying shuffle protocol. The online running time shows a similar trend: our protocol is about 34× faster than [33] at $n = 3$, and about 37× faster at $n = 15$.

An important phenomenon in Table 4 is that the overall running time of our protocol is even smaller than that of [33], despite the fact that our offline phase is more expensive. This is because we can parallelize[12] the online phase of the permutation protocol of [33] in the offline phase, which reduces the round complexity and hence the overall running time. This also implies that our transformation is really "free" in the semi-honest setting, as it does not increase the

[12] By "parallelize", we mean that the parties send the messages in independent sessions in one batch, instead of waiting until the end of one session to start the next session. Every party is single-threaded in our experiment.

**Table 4.** Semi-honest Shamir: Offline/Online Communication and Running Time with Varying Number of Parties

| | | Communication (MB) | | | | | Running Time (s) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | 3 | 6 | 9 | 12 | 15 | 3 | 6 | 9 | 12 | 15 |
| Offline | [33] | 108 | 379 | 795 | 1356 | 2063 | 7.75 | 22.2 | 56.7 | 109 | 208 |
| | ours | 155 | 488 | 964 | 1598 | 2372 | 12.9 | 29.6 | 66.6 | 126 | 225 |
| Online | [33] | 43.3 | 97.5 | 144 | 199 | 243 | 9.08 | 18.3 | 23.6 | 32.9 | 38.3 |
| | ours | 1.40 | 1.75 | 1.86 | 1.92 | 1.96 | 0.265 | 0.472 | 0.641 | 0.854 | 1.03 |

$n$ is the number of parties. The number of items is $m = 2^{16}$.
Communication is reported per party.
Each entry is the median of three repetitions; times are at 80 MiB/s bandwidth and 60 ms RTT.

overall running time of the shuffle protocol while dramatically reducing the online communication and running time.

We also evaluate the malicious setting with the same range of parties, fixing the number of items to $m = 2^{12}$. The results are listed in Table 5.

**Table 5.** Malicious Shamir: Offline/Online Communication and Running Time with Varying Number of Parties

| | | Communication (MB) | | | | | Running Time (s) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | 3 | 6 | 9 | 12 | 15 | 3 | 6 | 9 | 12 | 15 |
| Offline | [33] | 19.1 | 108 | 202 | 411 | 575 | 3.11 | 10.8 | 31.1 | 77.5 | 159 |
| | ours | 37.5 | 217 | 411 | 837 | 1179 | 9.98 | 24.2 | 60.8 | 147 | 297 |
| Online | [33] | 3.02 | 15.1 | 30.2 | 58.1 | 84.5 | 5.66 | 11.6 | 14.8 | 21.5 | 24.9 |
| | ours | 0.808 | 2.41 | 4.07 | 6.98 | 9.60 | 0.762 | 0.950 | 1.09 | 1.36 | 1.53 |

$n$ is the number of parties. The number of items is $m = 2^{12}$.
Communication is reported per party.
Each entry is the median of three repetitions; times are at 80 MiB/s bandwidth and 60 ms RTT.

Table 5 shows that the online communication of our protocol is consistently smaller than that of [33]; the improvement grows from about 3.7× at $n = 3$ to about 8.8× at $n = 15$. The online running time improvement is also substantial, ranging from about 7.4× to 16.3×.

The malicious setting also demonstrates the expected trade-off between the offline and online phases. Our protocol roughly doubles the offline communication, which is consistent with the additional correlation material needed for malicious security. This leads to a maximum 1.63× increase in the overall running time, when $n = 15$. Nevertheless, the online phase becomes much cheaper and scales more slowly with the number of parties, matching the purpose of moving the expensive work into the input-independent phase.

### E.3 Experiments with Number of Items

We next evaluate the impact of the number of items. For the semi-honest setting, we fix $n = 3$ and vary the number of items as $m = 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}$. The results are listed in Table 6.

**Table 6.** Semi-honest Shamir: Offline/Online Communication and Running Time with Varying Number of Items

| | | Communication (MB) | | | | | Running Time (s) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\log_2 m$ | 14 | 16 | 18 | 20 | 22 | 14 | 16 | 18 | 20 | 22 |
| Offline | [33] | 23.6 | 108 | 489 | 2181 | 9619 | 2.90 | 7.75 | 26.8 | 111 | 489 |
| | ours | 33.8 | 155 | 698 | 3104 | 13668 | 6.66 | 12.9 | 38.9 | 151 | 652 |
| Online | [33] | 9.44 | 43.3 | 196 | 872 | 3848 | 6.84 | 9.08 | 15.9 | 42.6 | 161 |
| | ours | 0.350 | 1.40 | 5.59 | 22.4 | 89.5 | 0.201 | 0.265 | 0.524 | 1.56 | 5.76 |

$m$ is the number of items. The number of parties is $n = 3$.
Communication is reported per party.
Each entry is the median of three repetitions; times are at 80 MiB/s bandwidth and 60 ms RTT.

From Table 6, the communication of both protocols grows with the number of items, as expected. The online communication of our protocol grows linearly in $m$ and remains much smaller than that of [33]. At $m = 2^{14}$, our online communication is about 27× smaller; at $m = 2^{22}$, the gap increases to about 43×. The online running time is also reduced by roughly one to two orders of magnitude, with a 27.9× improvement at $m = 2^{22}$.

An important phenomenon in Table 6 is that the overall running time of our protocol does not incur a material overhead despite the more expensive offline phase: it is smaller through $m = 2^{20}$ and only 1.3% larger at $m = 2^{22}$. Note that different from 4, for a fixed number of $n = 3$ parties, the parallelization of the online phase of [33] does not significantly reduce the overall running time, since the round complexity is already small with $n = 3$ parties. Nevertheless, this implies that our transformation is really "free" in the semi-honest setting, since it adds no material overall-time overhead while making the online phase dramatically faster.

Finally, we evaluate the malicious setting with $n = 3$ and the number of items ranging from $2^{10}$ to $2^{18}$. The results are listed in Table 7.

Table 7 confirms that the malicious Shamir implementation also benefits from the same online/offline separation. As the number of items increases, both protocols scale roughly linearly in the online communication, but our protocol has a much smaller constant. The online communication improvement grows from about 3.1× at $m = 2^{10}$ to about 5.7× at $m = 2^{18}$. This growth is consistent with theoretical analysis, since [33] has $m \log m$ online communication, while our protocol has $m$ online communication. The online running time of our protocol is also consistently smaller, with an improvement of about 6.7× at $m = 2^{18}$.

**Table 7.** Malicious Shamir: Offline/Online Communication and Running Time with Varying Number of Items

| | | Communication (MB) | | | | | Running Time (s) | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\log_2 m$ | 10 | 12 | 14 | 16 | 18 | 10 | 12 | 14 | 16 | 18 |
| Offline | [33] | 3.94 | 19.1 | 89.7 | 412 | 1859 | 1.84 | 3.11 | 8.07 | 28.4 | 119 |
| | ours | 7.95 | 37.5 | 173 | 781 | 3488 | 6.85 | 9.98 | 19.4 | 56.5 | 217 |
| Online | [33] | 0.625 | 3.02 | 14.2 | 65.0 | 294 | 4.59 | 5.66 | 7.11 | 10.3 | 21.6 |
| | ours | 0.202 | 0.808 | 3.23 | 12.9 | 51.7 | 0.730 | 0.762 | 0.874 | 1.34 | 3.21 |

$m$ is the number of items. The number of parties is $n = 3$.
Communication is reported per party.
Each entry is the median of three repetitions; times are at 80 MiB/s bandwidth and 60 ms RTT.

As can be observed from the table, our transformation introduces larger overhead to the overall running time in the malicious setting, compared to the semi-honest setting, with a maximum observed slowdown of about 1.57× at $m = 2^{18}$. Nevertheless, the improvement in the online phase is still significant, with a 6.3×–8.1× improvement across all $m$.

## F Supporting Power-of-Two Rings

*Overview.* We extend the construction to semantic values in $R = \mathbb{Z}_{2^k}$ using the lifted authentication technique of SPDZ$_{2^k}$ [14]. Let $\widehat{R} = \mathbb{Z}_{2^{k+s}}$. A value $x \in R$ is represented by a lift $\widehat{x} \in \widehat{R}$ satisfying $\widehat{x} \bmod 2^k = x$, and the shuffle authentication key $\beta$ is sampled uniformly from $\mathbb{Z}_{2^s}$ and embedded in $\widehat{R}$.

The offline and online shuffle and mask cancellation retain their existing form. We replace polynomial verification by independent random linear combinations whose coefficients remain private. Each verifier secretly inputs its two compressed values, and the parties open only their aggregate residual. A harmful deviation changes a value modulo $2^k$; the lifted authentication argument bounds acceptance of such a deviation by $(s + 1)2^{-s}$. The online communication and computation remain $O(C_{\mathrm{BC}} + nm)$.

### F.1 Protocol Modifications

We retain the existing notation for lifted protocol variables. Unless stated otherwise, all sampling, arithmetic, and equalities below are over $\widehat{R}$. The functionality $\mathcal{F}_{\mathrm{MPC}}$ must support exact arithmetic, private input, opening, and permutation over $\widehat{R}$. The shuffle key $\beta$ remains distinct from any backend authentication key $\alpha$.

The input is supplied as a shared lift. The ideal shuffle interprets this input modulo $2^k$ and returns a fresh uniformly lifted sharing of the shuffled semantic vector.

*Offline phase.* The parties generate $\beta$, the masks, and the shuffle correlations as in Algorithm 4, using the domains specified above. Replace the scalar challenges and their powers by independent uniform vectors

$$\langle \mathbf{c}_i \rangle \leftarrow \Pi_{\mathrm{rand}}(), \qquad i = 2, \ldots, n+1,$$

where the command is applied componentwise, and precompute

$$\langle d_i \rangle = \sum_{j=1}^{m} \langle \mathbf{c}_i(j) \rangle \langle \pi_{i-1}(\mathbf{r}'_{i-1})(j) \rangle.$$

For $i = 2, \ldots, n$, privately open $\mathbf{c}_i$ to $P_i$ during preprocessing. Keep $\mathbf{c}_{n+1}$ secret-shared. The values $d_i$ remain secret-shared throughout.

Additionally, parties preprocess an independent uniform vector $\langle \mathbf{q} \rangle \in \widehat{R}^m$, used only to randomize the upper bits of the output. All correlations are used once.

*Verification.* Execute the message phase of Algorithm 5 over $\widehat{R}$. After all messages are fixed, replace Algorithm 3 by Algorithm 6. Each verifier privately inputs two local compressions of its retained message; the final public message is compressed locally using the secret-shared coefficients $\langle \mathbf{c}_{n+1} \rangle$.

**Algorithm 6** Verify$((P_i : \mathbf{y}_{i-1})_{i=2}^{n}, \mathbf{y}_n, \mathrm{cor})$ over $\widehat{R}$

**Require:** Fresh cor from the modified offline phase, and all $\mathbf{y}_1, \ldots, \mathbf{y}_n$ are fixed.
**Ensure:** Accept if all parties are honest; reject harmful deviations w.h.p.
  **for** $i = 2$ to $n$ **do parallel**
    $P_i$ computes $w_{i,b} \coloneqq \sum_{j=1}^{m} \mathbf{c}_i(j) \mathbf{y}_{i-1}^{b}(j)$, $b \in \{1, 2\}$.
    $\langle w_{i,b} \rangle \leftarrow \Pi_{\mathrm{input}}(P_i : w_{i,b})$, $b \in \{1, 2\}$.
  **end for**
  Wait until all $(w_{i,1}, w_{i,2})$ are fixed.
  **for** $b = 1$ to $2$ **do**
    $\langle w_{n+1,b} \rangle \leftarrow \sum_{j=1}^{m} \mathbf{y}_n^b(j) \langle \mathbf{c}_{n+1}(j) \rangle$.
  **end for**
  **for** $i = 2$ to $n+1$ **do parallel**
    $\langle \eta_i \rangle \leftarrow \langle \beta \rangle \cdot \langle w_{i,1} \rangle - \langle w_{i,2} \rangle - \langle d_i \rangle$.
  **end for**
  $\langle f \rangle \leftarrow \sum_{i=2}^{n+1} \langle \eta_i \rangle$.
  Open $f$. Abort if $f \neq 0$.

*Output and authenticated openings.* Upon acceptance, return

$$\langle \mathbf{x}' \rangle = \langle \mathbf{y}_n^1 \rangle + \langle \pi_n(\mathbf{r}_n) \rangle + 2^k \langle \mathbf{q} \rangle.$$

The last term preserves the semantic output and makes its upper bits independently uniform.

Internal openings, including $f$, are exact openings in $\widehat{R}$. A concrete backend must follow the authentication order in Section 8.1: authenticate the computation before revealing $f$, and authenticate its opening before branching or releasing output. In particular, an $\mathrm{SPDZ}_{2^k}$-style backend must realize arithmetic with semantic width $k+s$ for these internal values.

### F.2 A Note on Forgery, Intuition of Security, and $\beta$

**Forgery** Different from the case of a field, the adversary now enjoys much more freedom in tampering with the messages without getting caught.

To illustrate, suppose we have a secret $\beta \leftarrow \mathbb{Z}_{2^s}$ and $r \in \widehat{R}$, where an honest receiver expects $(a, b)$ with $a = \beta b + r$. A malicious sender could send another $(a', b')$. If $a' = \beta b' + r$, this means

$$a - a' = \beta(b - b').$$

In a field, this immediately suggests that $\beta$ is guessed correctly, with negligible probability. In $\widehat{R}$, however, this is not true. For example, suppose $a - a' = 0$ and $b - b' = 2^{k+s-1}$, the relation holds as long as $\beta$ is even, with probability $1/2$. Testing this relation reveals the parity of $\beta$. Unlike a field forgery requiring a correct guess of the entire key, this ring forgery succeeds with constant probability.

The intuition behind the security is hence no longer that $\mathcal{A}$ cannot change any message undetected, but rather: it cannot tamper with the semantic part of the message, namely the lower $k$ bits of the message. If it only tampered higher bits, then the check could pass with non-negligible probability, but the semantic output is unchanged.

In the next subsection, we formalize this by constructing a simulator that simulates the view of an adversary from its input, randomness, and $\beta$. This essentially demonstrates:

1. The probability of accepting an incorrect semantic output is at most $(s+1)2^{-s}$;
2. Abort or not, the permutations of honest parties are protected.

The intuition behind such a simulator is discussed below.

**Intuition of Security** The simulator identifies the possible values of $f$ as a subset $I' \subseteq \widehat{R}$, which is determined by its sampled $\beta$ and the corrupted parties' behavior, without the honest permutations. In an honest execution, $I' = \{0\}$. Otherwise, $f$ is uniform over $I' = \tau + I$ for an ideal $I$ of the ring $\widehat{R}$; i.e. uniform over a coset of an ideal.

To see this, let $H$ index the honest verifiers and the final check, and let

$$\boldsymbol{\delta}_i = \beta \widehat{\mathbf{y}}^1_{i-1} - \widehat{\mathbf{y}}^2_{i-1} - \pi_{i-1}(\mathbf{r}'_{i-1})$$

denote the vector defects checked at indices $i \in H$. Honest message-phase operations only permute an incoming defect. Their aggregate is

$$S = \sum_{i \in H} \sum_{j=1}^{m} \mathbf{c}_i(j) \boldsymbol{\delta}_i(j).$$

And precisely

$$I = \sum_{i \in H} \sum_{j=1}^{m} \boldsymbol{\delta}_i(j) \widehat{R},$$

the ideal generated by the checked defect coordinates. The coefficients are independent and uniform, and remain hidden from the adversary. The map from these coefficients to $S$ is an additive homomorphism with image $I$ and equally sized fibers, hence the independent uniformity of the coefficients over $\widehat{R}$ implies the uniformity of $S$ over $I$. The additive bias $\tau$ can be introduced by corrupted parties by contributing incorrect $w_{i,1}$ and $w_{i,2}$, making $f$ uniform on $I' = \tau + I$.

To bound acceptance of an incorrect shuffle output, consider tampering with the lower $k$ bits. This deviation provides a coordinate where $\Delta_1 \not\equiv 0 \pmod{2^k}$ and a corresponding tag deviation $\Delta_2$, with $\beta\Delta_1 - \Delta_2 \in I$. A small ideal can give a high conditional acceptance probability, but the adversary must fix these deviations without knowing $\beta$. Averaging over the hidden key bounds acceptance of a harmful deviation by $(s+1)2^{-s}$.

*Leaking $\beta$.* The opened verification value can reveal information about $\beta$ on both acceptance and abort. Therefore, $\beta$ cannot be reused across sessions. To reuse permutations (Section 8), parties sample a fresh independent $\beta$ for each online shuffle, in addition to fresh masks and verification coefficients.

### F.3 Security

Let $T \subsetneq [n]$ be the set of corrupted parties. Write $\widehat{\mathbf{y}}_i$ and $\widehat{w}_{i,b}$ for the actual, possibly deviated messages and verifier inputs denoted by $\mathbf{y}_i$ and $w_{i,b}$ in Algorithm 6.

The offline simulation and message-phase masking arguments of Lemmas 1 and 2 apply over $\widehat{R}$: they use only independent uniform additive masks and bijections. The additional coefficients revealed to corrupted verifiers are independent randomness, while their $d_i$ remain secret. Thus the argument of Theorem 3 also applies: $\beta$ remains uniform given the adversary's view before $f$ is opened.

The following lemma replaces the field-specific aggregate-verification argument.

**Lemma 4 (Private Aggregate).** *Let* $\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}$ *be the adversary's view immediately before $f$ is opened, including its private verifier inputs. Conditioned on* $(\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}, \beta)$*, the value $f$ is uniform on a coset $\tau + I$ of an ideal $I \subseteq \widehat{R}$. Both $\tau$ and $I$ can be computed from* $(\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}, \beta)$ *without the honest permutations.*

*Proof.* For each maximal corrupted block $B = P_a, \ldots, P_b$, let $\widetilde{\mathbf{y}}_b$ be the outgoing pair obtained by replaying its prescribed operations from its actual entry message. The replayed pair need not equal the globally prescribed pair $\mathbf{y}_b$ after an earlier deviation. Define the block's deviation by

$$(\boldsymbol{\Delta}_B^1, \boldsymbol{\Delta}_B^2) \coloneqq (\widehat{\mathbf{y}}_b^1 - \widetilde{\mathbf{y}}_b^1, \widehat{\mathbf{y}}_b^2 - \widetilde{\mathbf{y}}_b^2),$$

and set

$$\boldsymbol{\omega}_B = \beta \boldsymbol{\Delta}_B^1 - \boldsymbol{\Delta}_B^2.$$

These differences are computable from $\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}$.

Define the set of indices of messages observable to honest parties by

$$H = \{i \in \{2, \ldots, n\} : i \notin T\} \cup \{n+1\}.$$

For $i \in H$, define the checked vector defect

$$\boldsymbol{\delta}_i = \beta \widehat{\mathbf{y}}_{i-1}^1 - \widehat{\mathbf{y}}_{i-1}^2 - \pi_{i-1}(\mathbf{r}'_{i-1}).$$

Defects are permuted between corrupted blocks and acquire the additional defect $\boldsymbol{\omega}_B$ at each block's exit. As permutations preserve the ideal generated by vector coordinates, induction over the blocks gives

$$I := \sum_{i \in H} \sum_{j=1}^{m} \boldsymbol{\delta}_i(j) \widehat{R} = \sum_{B} \sum_{j=1}^{m} \boldsymbol{\omega}_B(j) \widehat{R}.$$

Hence $I$ is computable from $(\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}, \beta)$.

For every $i \in H$, the coefficient vector $\mathbf{c}_i$ remains independent and uniform, and $\eta_i = \sum_j \mathbf{c}_i(j) \boldsymbol{\delta}_i(j)$. Such a linear combination with uniform coefficients is uniform over the ideal $\sum_j \boldsymbol{\delta}_i(j) \widehat{R}$. Consequently, $\sum_{i \in H} \eta_i$ is uniform on $I$.

For a corrupted verifier $P_i$ in block $B$, let $\widetilde{\mathbf{y}}_{i-1}$ be the incoming pair that $P_i$ would receive if all preceding parties in $B$ performed their prescribed operations, starting from the block's actual entry message. Define its reference defect by

$$\widetilde{\boldsymbol{\delta}}_i := \beta \widetilde{\mathbf{y}}_{i-1}^1 - \widetilde{\mathbf{y}}_{i-1}^2 - \pi_{i-1}(\mathbf{r}'_{i-1}).$$

The block's entry defect is either zero or checked by an honest verifier. Replay only permutes this defect, so every coordinate of $\widetilde{\boldsymbol{\delta}}_i$ lies in $I$. Define

$$\Delta_{i,b} = \widehat{w}_{i,b} - \sum_{j=1}^{m} \mathbf{c}_i(j) \widetilde{\mathbf{y}}_{i-1}^b(j),$$

i.e. the deviation from its prescribed output of Algorithm 6, given $\widetilde{\mathbf{y}}_{i-1}$ as inputs. Substitution into the definition of $\eta_i$ gives

$$\eta_i = \underbrace{\sum_{j=1}^{m} \mathbf{c}_i(j) \widetilde{\boldsymbol{\delta}}_i(j)}_{u_i \in I} + \beta \Delta_{i,1} - \Delta_{i,2}.$$

Thus $f = S + u + \tau$, where $S = \sum_{i \in H} \eta_i$, $u = \sum_{i \in \{2,\dots,n\} \cap T} u_i \in I$, and

$$\tau = \sum_{\substack{i=2,\dots,n \\ i \in T}} \big(\beta \Delta_{i,1} - \Delta_{i,2}\big).$$

For any fixed message-phase execution and $\beta$, $u$ is independent of the hidden coefficients used in $S$. Since $S$ is uniform on $I$, $f$ is uniform on $\tau + I$. The reference pairs, corrupted verifiers' coefficients, and submitted inputs are computable from $\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}$. Hence the simulator can compute $\tau$ given $\beta$, without computing $u$. As $I$ and $\tau$ are determined by $(\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}, \beta)$, the same distribution holds conditioned only on these values. □

**Theorem 6.** *Against a static adversary corrupting any proper subset of the parties, the modified protocol UC-securely realizes the semantic shuffle over $R$ in the $\mathcal{F}_{\mathrm{MPC}}$-hybrid model, with distinguishing advantage at most*

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

*Proof.* Correctness follows from the same mask cancellation as in Algorithm 5, followed by reduction modulo $2^k$. For security, reuse the simulator of Theorem 5, with the modified preprocessing and the distribution of $f$ supplied by Lemma 4. This simulates the verification transcript and any abort without the honest inputs or permutations.

It remains to exclude an accepted harmful deviation. If every $\boldsymbol{\Delta}_B^1$ is zero modulo $2^k$, the semantic output is correct. Otherwise, conditioned on $\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}$, select the first harmful block and one coordinate

$$\Delta_1 = \boldsymbol{\Delta}_B^1(j) \not\equiv 0 \pmod{2^k}, \qquad \Delta_2 = \boldsymbol{\Delta}_B^2(j).$$

Both are fixed independently of $\beta$. As in the existing non-forgery argument, a valid forgery requires $\beta\Delta_1 = \Delta_2$. Writing $\Delta_1 = 2^t u$ with $t < k$ and $u$ odd shows that this equation admits at most one $\beta \in \mathbb{Z}_{2^s}$.

To include acceptance through random compression, observe that $\beta\Delta_1 - \Delta_2 \in I$. Therefore,

$$\Pr[f = 0 \mid \mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}, \beta] \le \frac{1}{|I|} \le \frac{1}{|(\beta\Delta_1 - \Delta_2)\widehat{R}|}.$$

Write $\mathsf{lsb}(x)$ for the number of trailing zero bits in the canonical representative of $x \in \widehat{R}$, with $\mathsf{lsb}(0) = k + s$. For $\widehat{R} = \mathbb{Z}_{2^{k+s}}$,

$$\log_2 |(\beta\Delta_1 - \Delta_2)\widehat{R}| = k + s - \mathsf{lsb}(\beta\Delta_1 - \Delta_2).$$

Now we apply the same argument as $\mathrm{SPDZ}_{2^k}$ [14]. If $2^t \nmid \Delta_2$, this bound is already at most $2^{-s}$. Otherwise, write $\Delta_2 = 2^t \Delta_2'$. For $0 \le j < s$,

$$\Pr_{\beta}[\mathsf{lsb}(\beta u - \Delta_2') = j] = 2^{-j-1},$$

and the remaining event has probability $2^{-s}$. Averaging the preceding acceptance bound gives

$$\Pr[f = 0 \mid \mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}] \leq 2^{-s} + s\,2^{t-(k+s)-1} \leq (s+1)2^{-s}.$$

The witness was selected from $\mathrm{view}_{\mathcal{A}}^{\mathrm{fix}}$, so no union bound over blocks or coordinates is needed.

Outside this event, acceptance yields the correct semantic output. Since $\mathbf{q}$, which is used to mask the higher bits at the end of the online phase, is independent and used nowhere else, the returned value is a fresh uniform lift of that output, also jointly with the verification transcript. This completes the existing simulation argument. □

Choosing $s - \log_2(s+1) \geq \kappa$ gives statistical error at most $2^{-\kappa}$.

*Online complexity.* Count sending one $\widehat{R}$ element and one ring operation as constant cost. Assume scalar private input and multiplication each have $O(n)$ total online cost. The message phase, local compression, final shared compression, and output randomization cost $O(C_{\mathrm{BC}} + nm)$. Verification adds $O(n)$ scalar inputs and multiplications, for total online cost

$$O(C_{\mathrm{BC}} + nm + n^2).$$

Under total point-to-point communication and computation costs, an all-party scalar broadcast has $C_{\mathrm{BC}} = \Omega(n^2)$. Thus the online communication and computation are $O(C_{\mathrm{BC}} + nm)$. All coefficient generation, private coefficient delivery, and products defining $d_i$ occur offline.