

Truffle: Maliciously Secure Three-Party Shuffles with Applications to Parsing *

Nidhish Bhimrajka ¹, Yashvanth Kondi ¹, Daniel Noble ¹, and Bhavish Raj Gopal ²

¹Silence Laboratories,
nidhishbhimrajka1@gmail.com, yash@ykondi.net, danielnoble@pm.me
²Indian Institute of Science, bhavishraj@iisc.ac.in

October 6, 2026

Abstract

Deploying Secure Multiparty Computation (MPC) to operate on data from real world sources requires many connective elements that have not received much attention in the literature. One prominent example is that MPC protocols typically assume conveniently structured data, whereas data in the real world comes in formats that are meant to be handled by string parsing engines. In this work, we give a new constant round protocol for securely parsing secret shared strings into an MPC-friendly format, built upon *secure shuffle* as a key primitive.

Secure shuffle applies a permutation to a list while keeping both the permutation and the underlying data secret-shared at all times. While highly efficient protocols exist in the semi-honest setting, achieving malicious security has incurred significant overhead in both rounds and communication.

We present Truffle, a new three-party shuffle protocol that is secure against one active corruption to bridge this gap. Truffle achieves an amortised round complexity that for the first time matches the state-of-the-art semi-honest protocol, while incurring only sublinear communication overhead. The core technique that underlies Truffle is a novel distributed verifier zero-knowledge proof that checks the correctness of a putative shuffle. We implement Truffle and show via benchmarks that it is performant enough to handle real application data.

Beyond parsing secret shared data, secure shuffle finds applications in graph analytics, anonymous communication, Distributed Oblivious RAM, aggregate statistics, and secure sorting, each of which stand to benefit from the improvement that Truffle provides.

1 Introduction

Secure Multiparty Computation (MPC) enables a group of parties to collaboratively evaluate functions on their joint secret inputs. Since its initial theoretical conception, it has been deployed in various settings, in many cases to analyze data in controlled environments. However for true ubiquitous deployment, we must be able to connect MPC systems to data sources as they occur in the wild, and be compatible with ecosystems that were not designed with MPC as a first class concern. A lesson may be taken from the domain of threshold signatures: MPC-based decentralization of signature generation matured into an easily deployable, productized commodity only due to its

*This is the full version of the paper published at ACM CCS 2026 [BKNG26].

compatibility with widely used standards like ECDSA and EdDSA. In a similar vein, using MPC to process data in the real world requires mechanisms to connect to APIs and process standard file formats.

While there has been much work in the MPC literature on efficiently querying private datasets, secure inferencing on secret shared data, etc., it is standard to assume that the data is already structured in a convenient format. For instance, querying datasets is possible only once the data has been arranged into appropriate fields, and inferencing requires tokenization to have been performed beforehand. However, real data can come in formats such as JSON or XML, which are designed to be handled with string parsing libraries, but to our knowledge parsing such files inside of an MPC has not previously been considered.

Secure parsing is fundamentally challenging with such formats, as they are not *data-oblivious*, i.e., their memory access patterns are dependent on the underlying data values. As a result, even accessing a particular field may require knowing its position, which in turn may leak sensitive information.

For example, consider the following JSON records:

```
{owner: Alice, val: $1000.00}, {owner: Bob, val: $507.20}.
```

If Alice’s balance changes from \$1000.00 to \$999.99, the length of the first record decreases, shifting the position of all subsequent data. Consequently, the position of Bob’s record depends on Alice’s private value. Thus, even accessing Bob’s value can leak information about Alice’s balance.

To address this, the data must first be transformed into a data-oblivious representation before it can be efficiently processed within MPC. We can achieve this by parsing delimiter-separated strings and converting them into a fixed-length tabular form. Specifically, each segment is placed in a separate row and padded to a uniform length. For example, given the string

```
Alice-$1000.00-Bob-$507.20-
```

we transform it into

```
Alice_____
$1000.00____
Bob_____
$507.20_____
```

In this representation, all segments have fixed positions and lengths, ensuring that subsequent accesses are independent of the underlying data. Crucially, this transformation must be performed without revealing the positions of delimiters, segment lengths, or the amount of padding required for each segment.

In this work, we initiate the study of efficient secure parsing mechanisms for MPC systems, formalized as the *SplitAndPad* functionality. We focus on the three party, single active corruption setting as it is commonly used in the industry, as well as studied extensively in the academic literature. As latency is typically the biggest concern when executing MPC on substantially large datasets, we construct a new clean, constant round protocol for secure parsing. Our protocol is built upon two primitives: equality checking, and secure shuffle. While equality checking is a well studied, standard, and cheap primitive, known constructions for secure shuffle are relatively expensive, and form the clear bottleneck in secure parsing.

Secure shuffle refers to permuting an ordered list while ensuring that both the permutation and the underlying data remain private. It is a broadly useful primitive in its own right, having found applications in anonymous communication [KKP⁺23], secure graph analytics [AFO⁺21, KKPG24],

secure sorting [AHI⁺22], Distributed Oblivious RAM [LO13], secure aggregate statistics computation [JKK⁺22, AHI⁺22], and now secure parsing. Several works in the literature have studied and optimized secure shuffle protocols in the semi-honest setting [BPRS23, FJKW15, MRZ15, AFL⁺16], however active security has always come at substantial overhead.

Motivated by progressing towards an efficient realization of our parsing framework in the active corruption setting, in this work we focus on minimizing the overhead of actively secure shuffle. The most efficient semi-honest shuffle protocol in the three-party setting is due to Araki et al. [AFO⁺21], which achieves 2 rounds and $4n\ell$ bits of communication for shuffling n ℓ -bit elements. However, their actively secure construction incurs substantial overhead: a single shuffle instance requires around 5 extra rounds and $(2\ell + 12\kappa)n$ additional bits of bandwidth consumed, where κ is the statistical security parameter.

In this work, we show that the communication overhead for active security can be reduced to *sublinear* in the size of the dataset, so that its cost is subsumed by the baseline semi-honest protocol. When $\kappa = \ell$, this results in a 3x improvement in total communication cost. When a bit array is shuffled and $\kappa = 60$, this increases to a *90x improvement* in communication cost.

Our round complexity improves on prior work as well, reducing costs by 28%-33%. Moreover, we implement our protocol and demonstrate that we do not pay a penalty in terms of computation—for most parameters, our benchmarks show that we achieve comparable or better computation cost than prior work. Our key technical ingredient is a novel sublinear verification mechanism for a putative semi-honest shuffle, in the style of the Distributed Verifier Zero-Knowledge (DVZK) technique of Boyle et al. [BGIN19]. Our new Three-party shuffle protocol—Truffle—substantially improves secure parsing, and will be of benefit to the many applications of secure shuffle.

1.1 Our Contributions

We summarise our contributions below.

- **Maliciously secure shuffle.** We design a maliciously secure shuffle protocol in the three-party setting that achieves an amortised round complexity of 2 and communication complexity $4n\ell + 12\sqrt{nt} + O(\ell + \kappa + \lambda)$, where $t = \max(\ell, \kappa)$ and λ is the computational security parameter. Compared to the semi-honest protocol of [AFO⁺21], our protocol achieves the same amortised round complexity with only sublinear additional communication. In comparison, the malicious construction of [AFO⁺21] incurs at least linear overhead in communication cost and approximately a 50% overhead in round complexity. A detailed comparison is provided in Table 1. Note that these costs are based on instantiating the final step of the verification cost of [AFO⁺21] according to their footnote, as this is less expensive, as well as observing that dot-products have the same communication cost as a single multiplication, and that the protocol allows batched verification of multiple shuffles. Asharov et al. [AHI⁺22] also provide a maliciously secure shuffle based on authenticating secret-shares. This likewise uses a base semi-honest shuffle which is 50% more costly than ours, and incurs similar verification costs to [AFO⁺21] through the generation (1 round, $3n\kappa$ bits), shuffling ($6n\kappa$ bits) and checking (3 rounds, 12κ bits) of the authentication information. For completeness, we also include this as an analytical comparison in Table 1.

Our construction builds on the semi-honest shuffle protocol of Araki et al. [AFO⁺21], which achieves a 2 round protocol with a communication of $4n\ell$ for shuffling a list of size n with ℓ -bit elements. We upgrade this protocol to the malicious setting by enforcing the correctness of every message sent by the parties using verification mechanisms. In particular, we reduce the correctness of each of the four messages sent in the shuffle protocol to checking inner-product relations of the form $A \cdot B = c$. To verify these efficiently, we adapt Distributed Verifier Zero-Knowledge (DVZK) techniques of Boyle

Protocol	Semi-honest shuffle		Verification		Security
	Rounds	Comm. (bits)	Rounds	Comm. (bits)	
[AFO ⁺ 21] (Semi-Honest)	$2k$	$4n\ell k$	-	-	Semi-honest
[AFO ⁺ 21] (Malicious)	$3k$	$6n\ell k$	4	$6n\kappa k + 9\kappa$	Malicious (Abort)
[AHI ⁺ 22] (Malicious)	$3k$	$6n\ell k$	4	$6n\kappa k + 3n\kappa + 12\kappa$	Malicious (Abort)
Ours (Independent Shuffle)	$2k$	$4n\ell k$	3	$(12\sqrt{n} + 9)tk + 3\lambda + 3$	Malicious (Abort) [†]
Ours (Correlated Shuffle)				$(12\sqrt{n} + 9)t + 3\lambda + 3$	

- n denotes the number of elements to be shuffled, where each element is an ℓ -bit field element; κ denotes the statistical security parameter; λ denotes the computational security parameter; $t = \max(\ell, \kappa)$.

[†] Our protocol can be compiled to achieve full security using the standard approach of identifying a trusted third party (TTP) in the case of Abort. This requires a broadcast channel and incurs 2 rounds and communication of $O(\sqrt{n})$ field elements in the presence of a malicious party.

Table 1: Round complexity and communication of various shuffle protocols for k invocations.

et al. [BGIN19], which enable sublinear-cost verification of such relations. Since shuffle messages involve secret permutations, this reduction is non-trivial; we design a tailored transformation that maps permutation-based checks to inner-product relations where the prover knows all values, while the verifiers hold complementary shares. This allows us to verify the correctness of all messages with only sublinear overhead, while preserving the round complexity (amortised).

- **Correlated shuffle protocols.** In several applications, such as secure graph analytics [AFO⁺21, KKPG24], the same permutation is applied repeatedly across multiple input lists. We formalise this as the *correlated shuffle* setting and show that the verification cost of our protocol can be batched across multiple instances. This results in a communication cost for verification that is independent of the number of shuffle invocations. In comparison, prior protocols have communication cost for verification that scales linearly with the number of shuffle invocations as summarised in Table 1.
- **Secure string parsing.** We consider the problem of transforming a delimiter-separated string into a padded tabular representation, formalised as the *SplitAndPad* functionality. A naive approach would repeatedly scan the string to extract segments while inserting padding, resulting in a round complexity linear in the input length. Our protocol instead leverages secure shuffle to achieve constant round complexity, independent of the string length and the number of segments, while maintaining efficient communication.

At a high level, our protocol first identifies delimiter positions and securely computes, for each character, its target *row* (segment index) and *column* index based on its position in the input. It then determines the length of each segment and, using this information, generates candidate padding characters along with their target row and column indices. The real characters and padding candidates are combined, and a secure shuffle is used to route each element to its appropriate location in the output table. Our string parsing protocol is useful for many text processing MPC workflows, such as document parsing, natural language processing and large language model inference.

- **Implementation and Benchmarks.** We implement the malicious shuffle protocol in Rust and evaluate its performance against the state-of-the-art protocol of [AFO⁺21]. Our implementation is available at <https://github.com/anon1208/malicious-shuffle>.

2 Preliminaries

Notation. We use $\mathbb{F}$ to denote a finite field of size $|\mathbb{F}|$. For a positive integer n , we denote $[n] = \{1, 2, \dots, n\}$. We use uppercase variables to denote lists/vectors (e.g. A, X, T) and elements of list/vectors by standard indexing (e.g., $A[i]$). For vectors $A, B \in \mathbb{F}^n$, we use $A + B$ to denote element-wise addition and $A \cdot B$ to denote their inner product, i.e., $A \cdot B = \sum_{i=1}^n A[i]B[i]$.

Threat model. There are three parties $\mathcal{P} = \{P_1, P_2, P_3\}$ connected via pairwise private and authenticated channels. We assume a static, probabilistic polynomial-time adversary $\mathcal{A}$ that corrupts at most one party maliciously. Our protocols are proven secure in the standard real/ideal world simulation paradigm [Can07].

Secret sharing: (3,1)-RSS. We rely on replicated secret sharing. A value $x \in \mathbb{F}$ is shared as $x = x_1 + x_2 + x_3$, where $x_1, x_2, x_3 \in \mathbb{F}$ are uniformly random subject to the above constraint. P_i holds the pair (x_i, x_{i-1}) (indices modulo 3). We denote this sharing by $\langle x \rangle$.

Correlated randomness. We define the ideal functionality $\mathcal{F}_{\text{Rand}}$ that enables any pair of parties to generate shared randomness. On input a pair of party identifiers $i, j \in \{1, 2, 3\}$, the functionality outputs a common random value $r \in \mathbb{F}$ to parties P_i and P_j . $\mathcal{F}_{\text{Rand}}$ can be efficiently realized using pairwise keys for a pseudorandom function (PRF) established during a one-time setup phase. Thereafter, any pair of parties can non-interactively derive shared randomness by evaluating the PRF on a common input. With this, the parties can non-interactively generate $\langle \cdot \rangle$ -shares of random values. In particular, each pair of parties P_i and P_{i+1} for $i \in \{1, 2, 3\}$ can jointly sample their common share $r_i \in \mathbb{F}$ using their common key, thereby obtaining a valid sharing of a random value $r = r_1 + r_2 + r_3$. We also define an ideal functionality $\mathcal{F}_{\text{Coin}}$ that outputs a common random value $r \in \mathbb{F}$ to all parties. This can be realised by first using $\mathcal{F}_{\text{Rand}}$ to generate a shared random value in $\langle \cdot \rangle$ -form, followed by reconstructing it among all parties. When multiple random values are required, parties can instead generate and reconstruct a shared PRF key using $\mathcal{F}_{\text{Rand}}$, and subsequently expand it locally to obtain polynomially many common random values.

Permutations. Let $[n] = \{1, 2, \dots, n\}$. A permutation is a bijection $\pi : [n] \rightarrow [n]$. The set of all permutations forms the symmetric group S_n under composition.

Sampling a common random permutation. Two parties P_i, P_j jointly sample a permutation $\pi \in S_n$ as follows: Using $\mathcal{F}_{\text{Rand}}$, they generate shared random values $r_1, r_2, \dots, r_n \in \{0, 1\}^\kappa$ and form a list $L = \{(r_k, k)\}_{k=1}^n$. They then sort L by the first coordinate, and define π as the second coordinate in the sorted list.

Properties of permutations. For a list/vector $A \in \mathbb{F}^n$, $(\pi(A))_i = A_{\pi(i)}$ and for the inverse permutation, $(\pi^{-1}(A))_i = A_{\pi^{-1}(i)}$. We use the following properties of permutations:

- **Linearity:** For all $A, B \in \mathbb{F}^n$, $\pi(A + B) = \pi(A) + \pi(B)$.
- **Compatibility with Inner Products:** For all $M, A \in \mathbb{F}^n$, $M \cdot \pi(A) = \pi^{-1}(M) \cdot A$.
- **Random Projection:** For any $A \neq B \in \mathbb{F}^n$, if $M \xleftarrow{\$} \mathbb{F}^n$, then $\Pr[M \cdot A = M \cdot B] \leq \frac{1}{|\mathbb{F}|}$. Equivalently, $\Pr[M \cdot (A - B) = 0] \leq \frac{1}{|\mathbb{F}|}$.

3 Secure String Parsing

We begin by discussing the application of string parsing to motivate the use of secure shuffle. String parsing transforms a delimiter-separated string into a structured representation. This is a fundamental primitive in many data processing tasks, including text analytics, database queries, and preprocessing for machine learning and LLM pipelines. In the MPC setting, performing such parsing in a data-oblivious manner is challenging, as the locations of delimiters and the lengths of substrings are sensitive and must remain hidden. In this regard, secure shuffle enables efficient reordering of data without revealing access patterns, thereby serving as a powerful building block for implementing parsing primitives.

Given a delimiter-separated string, the goal of *string parsing* is to separate each segment and pad it to a fixed length, while protecting sensitive structural information such as the positions of delimiters, the length of individual segments, and consequently the amount of padding required for each individual segment. String parsing arises naturally in scenarios where data is obtained from external sources. For instance, after establishing a secure connection with an endpoint, encrypted data can be downloaded and converted into secret shares between a set of outsourced servers. However, many APIs return data in formats such as JSON or XML, which are inherently variable-length and do not include padding in order to minimise storage. This poses a challenge for secure computation, where data access patterns must be independent of the underlying values. In particular, the location of relevant data items may reveal sensitive information such as length. Thus, this operation is crucial in processing formatted strings within secure computation.

In our setting, the parties hold bitwise Boolean shares $\langle T \rangle$ of a delimiter-separated string T . Additionally, the parties hold $\langle \cdot \rangle$ -shares of a delimiter character d , a padding character p , and a public parameter c denoting an upper bound on the length of any segment. The goal is to output $\langle \cdot \rangle$ -shares of a table $\langle S \rangle$ with c columns such that each segment of T , delimited by d , appears on a separate row, padded with p up to length c . We refer to this operation as *SplitAndPad*. The formal functionality is given below.

Functionality 3.1. $\mathcal{F}_{\text{SplitAndPad}}$

Without loss of generality, let $P_{\text{cor}} \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\text{SplitAndPad}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. It receives as input $\langle \cdot \rangle$ -shares of a list T , a delimiter character d , padding character p and a public parameter c . It sends the corrupted party's input view to $\mathcal{S}$, and then proceeds as follows:

- Reconstruct the input list T using the honest parties' shares.
- Split T into segments, as divided by the delimiter character d , pad each segment to length c , without the delimiter, using character p , and generate the table S by placing each segment on a new row and padding them with p .
- The functionality sends r , which corresponds to the number of segments in the text, to $\mathcal{S}$ and receives the corrupted party's output shares $(S_{\text{cor}}, S_{\text{cor}-1} \in \mathbb{F}^{r \times c})$ from $\mathcal{S}$.
- Set $S_{\text{cor}+1} = S - (S_{\text{cor}} + S_{\text{cor}-1})$.
- Receive commands $C_i \in \{\text{Abort}, \text{Partial}, \text{Accept}\}$ from $\mathcal{S}$ for each honest party P_i . If $C_i = \text{Abort}$, it sends Abort to P_i . If $C_i = \text{Partial}$, it sends (Abort, r) to P_i . Otherwise, it sends $(\text{Output}, S_i, S_{i-1})$ to P_i . (This captures that the adversary can cause the protocol to abort after the number of rows is revealed, but before the final shares are distributed.)

Public information and leakage. The size $n = |\mathbf{T}|$ of the input string and the maximum segment length c are public, while $\mathcal{F}_{\text{SplitAndPad}}$ additionally reveals the number of segments or delimiters r . Consequently, these values also determine aggregate information about the input. In particular, the total number of non-delimiter characters is $\sum_{i=1}^r \text{Row}^{\text{len}}[i] = n - r$, and hence the total number of padding characters in the output is $rc - (n - r)$. Thus, the protocol does not hide the aggregate length of all segments (or, equivalently, the total amount of padding). It does, however, hide the length of individual segments, the amount of padding associated with individual segments, and the locations of the delimiters in the original string.

We view SplitAndPad as the task of assigning each character in the input string to a target row and column while simultaneously inserting padding characters into the remaining empty locations. We explain how this can be realised through a worked example followed by the formal details in Protocol 3.2. Consider the hyphen-delimited string

$$\mathbf{T} = \text{MY-NAME-IS-BOB-}$$

with delimiter $d = -$ and padding character $p = _$. We take $c = 4$. Thus, the desired output is

$$\mathbf{S} = \begin{array}{c} \text{MY_} \\ \text{NAME} \\ \text{IS_} \\ \text{BOB_} \end{array}$$

Tagging and indexing $\mathbf{T}$. The protocol first identifies delimiter positions. Parties compute a tag array $\mathbf{Tag}$ such that $\langle \mathbf{Tag}[i] \rangle = \mathcal{F}_{\text{Equality}}(\langle \mathbf{T}[i] \rangle, \langle d \rangle)$, so that $\mathbf{Tag}[i] = 1$ iff $\mathbf{T}[i]$ is a delimiter. Using this, the row index of each position is computed as the prefix sum: $\langle \mathbf{T}_{\text{ind}}^{\text{row}}[i] \rangle = 1 + \sum_{j=1}^i \langle \mathbf{Tag}[j] \rangle$. Thus, every non-delimiter character is assigned the row corresponding to the segment it belongs to. Each position is also assigned a preliminary column index $\langle \mathbf{T}_{\text{ind}}^{\text{col}}[i] \rangle = i$, which will later be corrected.

$\mathbf{T}$	M	Y	$-$	N	A	M	E	$-$	I	S	$-$	B	O	B	$-$
$\mathbf{Tag}$	0	0	1	0	0	0	0	1	0	0	1	0	0	0	1
$\mathbf{T}_{\text{ind}}^{\text{row}}$	1	1	2	2	2	2	2	3	3	3	4	4	4	4	5
$\mathbf{T}_{\text{ind}}^{\text{col}}$	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Computing row boundaries and row lengths. To determine the length of each segment, the protocol needs to identify the delimiter positions. Revealing them directly would expose the segment boundaries in the original string. To avoid this, parties jointly shuffle the arrays $\mathbf{Tag}$, $\mathbf{T}_{\text{ind}}^{\text{row}}$, and $\mathbf{T}_{\text{ind}}^{\text{col}}$ using the same random permutation. In the protocol, we model this shuffle operation by an ideal functionality $\mathcal{F}_{\text{Shuffle}}$. *Informally*, $\mathcal{F}_{\text{Shuffle}}$ takes as input one or more secret-shared arrays and outputs their jointly permuted versions under a random permutation, such that the permutation remains hidden from all parties. Importantly, when multiple arrays are provided, the same permutation is applied to all of them, preserving the correspondence between entries while hiding their original positions. The formal functionality and its realisation are described later in §4.

Suppose the shuffle yields:

$\mathbf{T}'$	O	A	S	$-$	I	B	$-$	B	E	N	$-$	Y	M	M	$-$
$\mathbf{Tag}'$	0	0	0	1	0	0	1	0	0	0	1	0	0	0	1
$\mathbf{T}_{\text{ind}}^{\text{row}'}$	4	2	3	2	3	4	5	4	2	2	4	1	2	1	3
$\mathbf{Ind}$	13	5	10	3	9	12	15	14	7	4	11	2	6	1	8

Here, Ind is the shuffled version of $\text{T}_{\text{ind}}^{\text{col}}$, i.e., it contains the original position of each shuffled element. The protocol reconstructs Tag' , which also allows computing the number of delimiters/rows $r = \sum_{i=1}^n \text{Tag}'[i] = 4$. It then considers $\text{T}_{\text{ind}}^{\text{row}'}$ and Ind only at positions where $\text{Tag}' = 1$. These positions correspond precisely to the shuffled delimiters. In this example, the corresponding secret-shared pairs are

$$(\text{T}_{\text{ind}}^{\text{row}'}, \text{Ind}) = (2, 3), (5, 15), (4, 11), (3, 8).$$

The protocol then reconstructs $\text{T}_{\text{ind}}^{\text{row}'}$ in these pairs, while Ind remains secret-shared. This allows local sorting of the above pairs according to $\text{T}_{\text{ind}}^{\text{row}'}$, which consequently orders the delimiters according to the segments they terminate. Taking the corresponding Ind values gives the secret-shared delimiter positions

$$\text{Row}_{\text{ind}}^{\text{end}} = (3, 8, 11, 15).$$

The protocol then computes the starting position of each row as

$$\text{Row}_{\text{ind}}^{\text{start}}[1] = 1, \quad \text{Row}_{\text{ind}}^{\text{start}}[i] = \text{Row}_{\text{ind}}^{\text{end}}[i-1] + 1, \quad i = 2, \dots, r,$$

giving

$$\text{Row}_{\text{ind}}^{\text{start}} = (1, 4, 9, 12).$$

Since $\text{Row}_{\text{ind}}^{\text{end}}[i]$ denotes the position of the delimiter immediately following row i , the segment length is

$$\text{Row}^{\text{len}}[i] = \text{Row}_{\text{ind}}^{\text{end}}[i] - \text{Row}_{\text{ind}}^{\text{start}}[i],$$

and hence

$$\text{Row}^{\text{len}} = (2, 4, 2, 3).$$

These values are summarized below:

row i	1	2	3	4
$\text{Row}_{\text{ind}}^{\text{end}}[i]$	3	8	11	15
$\text{Row}_{\text{ind}}^{\text{start}}[i]$	1	4	9	12
$\text{Row}^{\text{len}}[i]$	2	4	2	3

Thus, the four segments are indeed

$$\text{MY}, \quad \text{NAME}, \quad \text{IS}, \quad \text{BOB}.$$

Unary representation of row lengths. The protocol next securely determines, for each row, which positions correspond to real characters and which should contain padding. For this, it computes a unary encoding of the row length. Concretely, for row i , the protocol constructs a vector U_i of length c such that the first $\text{Row}^{\text{len}}[i]$ positions are 1 and the rest are 0. This is achieved by initialising a unit vector $I = (0, 0, \dots, 1)$, cyclically shifting I by $\text{Row}^{\text{len}}[i]$, then computing a reverse prefix sum. For this example, they are:

$$\text{U}_1 = (1, 1, 0, 0), \quad \text{U}_2 = (1, 1, 1, 1), \quad \text{U}_3 = (1, 1, 0, 0), \quad \text{U}_4 = (1, 1, 1, 0).$$

The 1's mark positions already occupied by actual characters, while the 0's mark positions where padding must be inserted.

Observe that cyclic shift of arrays can also be realised using shuffle. For now, we model this operation using an ideal functionality $\mathcal{F}_{\text{Shift}}$. $\mathcal{F}_{\text{Shift}}$ takes as input a secret-shared vector and a secret-shared offset, and outputs a cyclic shift of the vector by the given offset, without revealing the offset or the data. The formal details and its realisation are described in the subsequent sections.

One subtlety is that, for efficiency of the $\mathcal{F}_{\text{Shift}}$ protocol, $\text{Row}^{\text{len}}[i]$ should be represented in a small arithmetic field, such as $\mathbb{Z}_q$ for a prime $c < q \leq 2c$. Variables $\text{Row}_{\text{ind}}^{\text{end}}$, $\text{Row}_{\text{ind}}^{\text{start}}$, Ind and (later) column indexes therefore must be in this field as well; reducing these modulo q does not cause any correctness errors as the only outputs from these are column IDs, which are always less than q .

Initialising padding characters. The protocol creates an array, pad , containing $r \times c = 16$ candidate padding characters, all equal to $\mathbf{p} = _$. For each row i , it assigns c such candidates to that row. Their row ids are

$$\mathbf{P}_{\text{ind}}^{\text{row}} = (1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4).$$

Each candidate padding character is also assigned a preliminary column index $\mathbf{P}_{\text{ind}}^{\text{col}}$. The protocol chooses these indices so that after subtracting the row start index, $\text{Row}_{\text{ind}}^{\text{start}}[i]$, the surviving padding characters will land exactly in the unused columns of row i . Intuitively, the unary vector is used to distinguish the first $\text{Row}^{\text{len}}[i]$ candidate positions, which correspond to real characters and hence should be discarded. The remaining $c - \text{Row}^{\text{len}}[i]$ positions are genuine padding and must be retained. For the example, the rows require $(4 - 2, 4 - 4, 4 - 2, 4 - 3) = (2, 0, 2, 1)$ padding characters, respectively.

Merging, filtering and routing. The protocol now forms the combined arrays

$$\mathbf{A} = \mathbf{T} \parallel \mathbf{P}, \quad \mathbf{A}_{\text{ind}}^{\text{row}} = \mathbf{T}_{\text{ind}}^{\text{row}} \parallel \mathbf{P}_{\text{ind}}^{\text{row}}, \quad \mathbf{A}_{\text{ind}}^{\text{col}} = \mathbf{T}_{\text{ind}}^{\text{col}} \parallel \mathbf{P}_{\text{ind}}^{\text{col}},$$

and also combines the delimiter tags and unary indicators into a single array

$$\mathbf{U} = \text{Tag} \parallel \mathbf{U}_1 \parallel \dots \parallel \mathbf{U}_r.$$

Here, the entries coming from $\mathbf{T}$ correspond to actual input characters, the entries coming from $\mathbf{P}$ correspond to candidate padding characters. The tag $\mathbf{U}$ indicates which entries to discard (delimiter characters from $\mathbf{T}$ have $\mathbf{U} = 1$ and must be removed, and excess padding characters also have indicator 1 and must be removed).

Thus, after combining, the protocol has a single list containing all actual characters, delimiters, and candidate padding characters, together with their row and column identifiers. The protocol then applies a fresh common random shuffle to $\mathbf{A}$, $\mathbf{A}_{\text{ind}}^{\text{row}}$, $\mathbf{A}_{\text{ind}}^{\text{col}}$, and $\mathbf{U}$. This hides the origin of every entry before any further information is reconstructed. $\mathbf{U}$ is then reconstructed. This is secure since $\mathbf{U}$ consists of exactly n 1s (for the delimiters and excess padding) and rc 0s (for the characters and padding to be output), shuffled according to a fresh random permutation. Every entry with mask value 1 is then removed. In our example, this removes: (i) the four delimiters from the original string, and (ii) the excess padding candidates, namely the first 2, 4, 2, 3 candidates from rows 1, 2, 3, 4, respectively.

What remains are exactly the 11 real characters $\mathbf{M}, \mathbf{Y}, \mathbf{N}, \mathbf{A}, \mathbf{M}, \mathbf{E}, \mathbf{I}, \mathbf{S}, \mathbf{B}, \mathbf{O}, \mathbf{B}$, and the 5 padding characters needed to fill the rows to length 4. At this point, each surviving entry has a row index and a preliminary column index. The protocol reconstructs $\mathbf{A}_{\text{ind}}^{\text{row}}$, and corrects the column indices by subtracting the row start of the corresponding row:

$$\mathbf{A}_{\text{ind}}^{\text{col}}[j] \leftarrow \mathbf{A}_{\text{ind}}^{\text{col}}[j] - (\text{Row}_{\text{ind}}^{\text{start}}[\mathbf{A}_{\text{ind}}^{\text{row}}[j]] - 1).$$

This gives the local column positions within each row. For example, $\mathbf{N}, \mathbf{A}, \mathbf{M}, \mathbf{E}$ from positions 4, 5, 6, 7 belong to row 2, so after subtracting $\text{Row}_{\text{ind}}^{\text{start}}[2] - 1 = 3$ they land in columns 1, 2, 3, 4.

Finally, the protocol reconstructs $A_{\text{ind}}^{\text{col}}$ and places each surviving entry of A into the output table according to its row and column indices. The resulting table, as desired, is:

$$S = \begin{array}{l} \text{MY_} \\ \text{NAME} \\ \text{IS_} \\ \text{BOB_} \end{array}$$

The formal details of the protocol appear below. Characters are represented as Boolean sharings. All other variables are Arithmetic sharings (with a prime modulus), which allows for local addition and subtractions (including calculating cumulative sums).

Observe that random shuffles are critical at multiples stages of the above protocol. Shuffles ensure that the positions of delimiters, segment boundaries, and intermediate indices remain hidden during computation. Operations such as reconstructing delimiter tags or row identifiers would otherwise reveal sensitive structural information about the input, such as the lengths and locations of segments. By first randomly permuting the data and then selectively reconstructing only the required components, the protocol guarantees that no information about the original ordering is leaked.

Moreover, the shuffle plays a crucial role in securely filtering elements (such as delimiters and excess padding candidates) and in routing characters to their final row and column positions without exposing access patterns. Without this randomisation step, the protocol would inherently leak positional information through memory access patterns or reconstruction steps. Thus, shuffle serves as the backbone that enables data-oblivious manipulation of variable-length structured data, making it indispensable for achieving privacy in string parsing within secure computation.

Protocol 3.2. $\Pi_{\text{SplitAndPad}}$

Input: Parties hold $\langle \cdot \rangle$ -shares of a list T (of size n), a delimiter character d , a padding character p and a public parameter c that denotes the maximum length of a string.

Output: Parties output $\langle \cdot \rangle$ -shares of a table S of dimension $r \times c$, where r denotes the number of segments in T . Further, the i^{th} row in S contains the i^{th} segment of T padded with p for upto c characters.

Tagging and indexing T .

1. Compute $\langle \text{Tag}[i] \rangle = \mathcal{F}_{\text{Equality}}(\langle T[i] \rangle, \langle d \rangle)$ for $i = 1$ to n .
2. Compute $\langle T_{\text{ind}}^{\text{row}}[i] \rangle = 1 + \sum_{j=1}^i \langle \text{Tag}[j] \rangle$ for $i = 1$ to n .
3. Initialise array $\langle T_{\text{ind}}^{\text{col}}[j] \rangle = \langle j \rangle$ for $j = 1$ to n .

Computing row boundaries and row lengths.

4. Party $P_i, P_{i+1} \in \mathcal{P}$ invoke $\mathcal{F}_{\text{Rand}}$ to sample random permutation $\pi_i \in S_n$. Parties compute

$$\begin{aligned} \langle \text{Tag}' \rangle &= \mathcal{F}_{\text{Shuffle}}(\langle \text{Tag} \rangle, P_i : (\pi_i, \pi_{i-1})) \\ \langle T_{\text{ind}}^{\text{row}'} \rangle &= \mathcal{F}_{\text{Shuffle}}(\langle T_{\text{ind}}^{\text{row}} \rangle, P_i : (\pi_i, \pi_{i-1})) \\ \langle \text{Ind} \rangle &= \mathcal{F}_{\text{Shuffle}}\left(\left\langle T_{\text{ind}}^{\text{col}} \right\rangle, P_i : (\pi_i, \pi_{i-1})\right) \end{aligned}$$

5. Reconstruct Tag' followed by reconstructing $T_{\text{ind}}^{\text{row}'}$ of entries with $\text{Tag}' = 1$.
6. Compute number of rows $r = \sum_{i=1}^n \text{Tag}'[i]$.

7. Initialise array $\langle \text{Row}_{\text{ind}}^{\text{end}} \rangle$ to be the sub-array of $\langle \text{Ind} \rangle$, for which Tag' is 1, sorted according to $\text{T}_{\text{ind}}^{\text{row}'}$.
8. Initialise $\text{Row}_{\text{ind}}^{\text{start}}$ such that $\langle \text{Row}_{\text{ind}}^{\text{start}}[1] \rangle = \langle 1 \rangle$ and $\langle \text{Row}_{\text{ind}}^{\text{start}}[i] \rangle = \langle \text{Row}_{\text{ind}}^{\text{end}}[i-1] \rangle + 1$ for $i = 2$ to r .
9. Compute $\langle \text{Row}_{\text{ind}}^{\text{len}}[i] \rangle = \langle \text{Row}_{\text{ind}}^{\text{end}}[i] \rangle - \langle \text{Row}_{\text{ind}}^{\text{start}}[i] \rangle$.

Unary representation of segment.

10. Initialise array $\langle I \rangle$ of length $q > c$ such that $I[j] = 0$ for $1 \leq j \leq q-1$ and $I[q] = 1$. Parties compute:

$$\langle W_i \rangle = \mathcal{F}_{\text{Shift}} \left(\langle I \rangle, \langle \text{Row}_{\text{ind}}^{\text{len}}[i] \rangle \right) \quad \text{for } i = 1 \text{ to } r$$

11. For $i = 1$ to r , $j = 1$ to c , set $\langle \text{U}_i[j] \rangle = \sum_{k=j}^c \langle W_i[k] \rangle$.

Initialising padding characters.

12. Initialise array $\langle \text{P} \rangle$ consisting of $r \cdot c$ copies of $\langle p \rangle$.
13. Initialise arrays $\langle \text{P}_{\text{ind}}^{\text{row}} \rangle, \langle \text{P}_{\text{ind}}^{\text{col}} \rangle$ of length $r \times c$.
14. For $i = 1$ to r

- (a) For $j = 1$ to c
 - i. $k = (i-1) * c + j$
 - ii. $\langle \text{P}_{\text{ind}}^{\text{row}}[k] \rangle = \langle i \rangle$
 - iii. $\langle \text{P}_{\text{ind}}^{\text{col}}[k] \rangle = \langle \text{Row}_{\text{ind}}^{\text{start}}[i] \rangle + j - 1$.

Merging, filtering and routing.

15. Set $\langle \text{A} \rangle = \langle \text{T} \rangle \parallel \langle \text{P} \rangle, \langle \text{A}_{\text{ind}}^{\text{row}} \rangle = \langle \text{T}_{\text{ind}}^{\text{row}} \rangle \parallel \langle \text{P}_{\text{ind}}^{\text{row}} \rangle, \langle \text{A}_{\text{ind}}^{\text{col}} \rangle = \langle \text{T}_{\text{ind}}^{\text{col}} \rangle \parallel \langle \text{P}_{\text{ind}}^{\text{col}} \rangle$ and $\langle \text{U} \rangle = \langle \text{Tag} \rangle \parallel \langle \text{U}_1 \rangle \parallel \dots \parallel \langle \text{U}_r \rangle$.
16. Party $P_i, P_{i+1} \in \mathcal{P}$ invoke $\mathcal{F}_{\text{Rand}}$ to sample random permutation $\rho_i \in S_{n+c \cdot r}$. Parties compute

$$\begin{aligned} \langle \text{A} \rangle &= \mathcal{F}_{\text{Shuffle}}(\langle \text{A} \rangle, P_i : (\rho_i, \rho_{i-1})) \\ \langle \text{A}_{\text{ind}}^{\text{row}} \rangle &= \mathcal{F}_{\text{Shuffle}}(\langle \text{A}_{\text{ind}}^{\text{row}} \rangle, P_i : (\rho_i, \rho_{i-1})) \\ \langle \text{A}_{\text{ind}}^{\text{col}} \rangle &= \mathcal{F}_{\text{Shuffle}}(\langle \text{A}_{\text{ind}}^{\text{col}} \rangle, P_i : (\rho_i, \rho_{i-1})) \\ \langle \text{U} \rangle &= \mathcal{F}_{\text{Shuffle}}(\langle \text{U} \rangle, P_i : (\rho_i, \rho_{i-1})) \end{aligned}$$

17. Reconstruct U and remove entries in $\langle \text{A} \rangle, \langle \text{A}_{\text{ind}}^{\text{row}} \rangle$ and $\langle \text{A}_{\text{ind}}^{\text{col}} \rangle$ that have 1 in U .
18. Reconstruct $\text{A}_{\text{ind}}^{\text{row}}$ and compute $\langle \text{A}_{\text{ind}}^{\text{col}}[j] \rangle = \langle \text{A}_{\text{ind}}^{\text{col}}[j] \rangle - (\langle \text{Row}_{\text{ind}}^{\text{start}}[\text{A}_{\text{ind}}^{\text{row}}[j]] \rangle - 1)$ for $j = 1$ to $r \cdot c$.
19. Reconstruct $\text{A}_{\text{ind}}^{\text{col}}$ and compute $\langle \text{S} \rangle$ based on $\text{A}_{\text{ind}}^{\text{row}}$ and $\text{A}_{\text{ind}}^{\text{col}}$, i.e., for $i = 1$ to $r \times c$
 - (a) $j = \text{A}_{\text{ind}}^{\text{row}}[i], k = \text{A}_{\text{ind}}^{\text{col}}[i]$.
 - (b) $\langle \text{S}[j][k] \rangle = \langle \text{A}[i] \rangle$.

Claim 3.3. *The protocol $\Pi_{\text{SplitAndPad}}$ requires only a constant round complexity and a communication complexity of $\mathcal{O}(r \cdot c \cdot w + r \cdot c \cdot \log(n) + n \cdot w \cdot \log(n))$, where n denotes the size of T , w denotes the bit-length of each character, r denotes the number of segments, and c denotes the maximum*

length of any segment.

Proof. The costs follow from the cost of $\mathcal{F}_{\text{Equality}}$, $\mathcal{F}_{\text{Shift}}$, $\mathcal{F}_{\text{Shuffle}}$, and reconstruction. For $\mathcal{F}_{\text{Equality}}$, we evaluate a simple Boolean OR-tree circuit ($\log(w)$ rounds) and convert the result to an arithmetic sharing by evaluating the XOR arithmetic circuit on the three shares (2 rounds). Arithmetic uses a prime modulus with at most $v = \lceil \log(n) + 1 \rceil$ bits. Both the Boolean and Arithmetic circuits can be evaluated efficiently using DVZK [BGIN19]. This means that the communication cost equals that of the semi-honest protocol (3 field elements per multiplication [AFL⁺16]) plus a verification cost that is sublinear in the number of gates. Verification requires 3 rounds but these can occur in parallel with the subsequent shuffle, so do not contribute to the round complexity. Therefore, $\mathcal{F}_{\text{Equality}}$ requires $2 + \log(w)$ rounds and communicates at most $3nw + 6nv + o(n)$ bits.

Looking ahead, $\mathcal{F}_{\text{Shift}}$ and $\mathcal{F}_{\text{Shuffle}}$ when instantiated with Protocol 4.4 require 5 rounds. Note, however, that no reconstruction occurs between $\mathcal{F}_{\text{Shift}}$ and the $\mathcal{F}_{\text{Shuffle}}$ of step 16, so their verification can be combined, leading to $\mathcal{F}_{\text{Shift}}$ and $\mathcal{F}_{\text{Shuffle}}$ contributing a total of 12 rounds to $\Pi_{\text{SplitAndPad}}$. The shuffle/shift of step 4 (resp. 10, 16) shuffles 1 (resp. r , 1) list(s) containing n (resp. $2 \cdot c$, $r \cdot c + n$) items, each of size $3v$ (resp. v , $3v + w$) bits, so results in communication cost of $12nv + o(n)$ (resp. $8rcv + o(rc)$, $12(n + rc)v + 4(n + rc)w + o(rc)$). Together the shuffles/shift communicate $24nv + 20rcv + 4nw + 4rcw + o(rc)$ bits.

Additionally, there are 5 sequential reconstructions, of up to $n + rc$ values, each of size at most v . In reconstruction, each share need only be sent by one party, the other can send a hash. Therefore reconstruction adds up to $15(n + rc)v + \mathcal{O}(\kappa)$ bits to the communication cost. Therefore $\Pi_{\text{SplitAndPad}}$ requires $19 + \log(w)$ rounds and communicates $45nv + 7nw + 35rcv + 4rcw + o(rc) = \mathcal{O}(rcw + rc \log(n))$ bits.

Generally $\log(w)$ can be viewed as a constant, since w is the bit-length of a character. Even if w cannot be viewed as a constant, the total round complexity can be made constant by instead instantiating $\mathcal{F}_{\text{Equality}}$ with the Symmetric Functions protocol of [DFK⁺06]. This would also increase the communication cost by $\mathcal{O}(n \cdot w \cdot \log(n))$, leading to the total communication cost given in the claim. $\square$

Concrete cost. Consider processing an input file containing $n = 1000$ single-byte characters (i.e. $w = 8$) which consists of $r = 50$ segments of width at most $c = 40$. Ignoring terms that are sublinear in n or $r \cdot c$, the total communication cost then would be about $45 \cdot 1000 \cdot 11 + 7 \cdot 1000 \cdot 8 + 35 \cdot 50 \cdot 40 \cdot 11 + 4 \cdot 50 \cdot 40 \cdot 8$ bits ≤ 174 KB, or 174 bytes per input character. The round complexity would be 22 and independent of the input length or number of segments.

3.1 Security of $\Pi_{\text{SplitAndPad}}$

Theorem 3.4. *Protocol 3.2 securely computes Functionality 3.1 with abort in the presence of one malicious party.*

Proof. Let $\mathcal{A}$ be a real-world adversary corrupting at most one party, and let P_{cor} denote the corrupted party. We construct an ideal-world simulator $\mathcal{S}$ such that the view of $\mathcal{A}$ in the real execution of $\Pi_{\text{SplitAndPad}}$ is identically distributed to its view in the ideal execution with $\mathcal{F}_{\text{SplitAndPad}}$. The simulator works as follows:

$\mathcal{S}$ receives the corrupted party's input shares of $\mathbf{T}$ from $\mathcal{F}_{\text{SplitAndPad}}$ and forwards them to $\mathcal{F}_{\text{SplitAndPad}}$. $\mathcal{S}$ simulates the execution of $\Pi_{\text{SplitAndPad}}$ as follows. The simulator emulates all invocations of $\mathcal{F}_{\text{Rand}}$, $\mathcal{F}_{\text{Shuffle}}$ and $\mathcal{F}_{\text{Shift}}$ using random inputs for honest parties. Apart from this, the only instances where messages are exchanged are during reconstruction of Tag' , $\mathbf{T}_{\text{ind}}^{\text{row}}$ (step 5), $\mathbf{U}$ (step 17), $\mathbf{A}_{\text{ind}}^{\text{row}}$ (step 18) and $\mathbf{A}_{\text{ind}}^{\text{col}}$ (step 19).

- *Reconstruction of Tag' (Step 5).* From $\mathcal{F}_{\text{SplitAndPad}}$, $\mathcal{S}$ learns the number of segments r . It samples a random binary vector Tag' of length n with exactly r ones and adjusts the honest parties' shares so that reconstruction yields this Tag' .
- *Reconstruction of $\mathbf{T}_{\text{ind}}^{\text{row}}$ (Step 5).* At positions where $\text{Tag}' = 1$, $\mathcal{S}$ assigns distinct row indices in $\{2, \dots, r+1\}$ in random order.
- *Reconstruction of $\mathbf{U}$ (Step 17).* $\mathcal{S}$ samples a vector $\mathbf{U}$ such that exactly $r \cdot c$ entries are 0 (corresponding to surviving entries) and the rest are 1 (to be discarded). $\mathcal{S}$ adjusts the honest parties' shares so that reconstruction yields this $\mathbf{U}$.
- *Reconstruction of $\mathbf{A}_{\text{ind}}^{\text{row}}$ and $\mathbf{A}_{\text{ind}}^{\text{col}}$ (Steps 18–19).* $\mathcal{S}$ assigns each surviving entry (i.e., those with $\mathbf{U} = 0$) a random valid row and column index such that each row has exactly c entries, and all row index, column index pairs are unique. Accordingly, $\mathcal{S}$ adjusts the shares of honest parties during reconstruction.

Using the above assignments, $\mathcal{S}$ locally computes the corresponding output shares for the corrupted party and sends them to $\mathcal{F}_{\text{SplitAndPad}}$. Observe that in the real execution, all reconstructed values/indices (row/column assignments) are independent of the original positions due to prior shuffling. Thus, the simulator also generates identically distributed values randomly, subject only to the constraints imposed by the public parameters and the revealed value r ; in particular, the reconstructed intermediate values are not linked to the individual segment lengths. Hence, the view of $\mathcal{A}$ in the simulated execution is identical to its view in the real execution. $\square$

4 Secret-shared shuffle

We begin by defining the ideal functionality for shuffle, denoted by $\mathcal{F}_{\text{Shuffle}}$.

Our protocol for secure shuffle builds on the semi-honest 3-party shuffle protocol of [AFO⁺21], which achieves a 2-round, 4-message protocol in the considered setting. While this protocol ensures privacy and correctness in the semi-honest setting, it does not protect against malicious deviations.

Functionality 4.1. $\mathcal{F}_{\text{Shuffle}}$

Without loss of generality, let $P_{\text{cor}} \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\text{Shuffle}}$ interacts with the parties in $\mathcal{P}$ and the adversary $\mathcal{S}$. It receives as input the shares $(\mathbf{T}_i, \mathbf{T}_{i-1})$ and a permutation π_i, π_{i-1} from each honest party $P_i \in \mathcal{P}$, where these constitute a $\langle \cdot \rangle$ -sharing of the input list size n . It sends the corrupted party's input view $(\mathbf{T}_c, \mathbf{T}_{c-1}), (\pi_c, \pi_{c-1})$ to $\mathcal{S}$.

Let $\mathbf{T}_O$ denote the shuffled output list. The functionality receives from $\mathcal{S}$ the corrupted party's output shares $(\mathbf{T}_{O_c}, \mathbf{T}_{O_{c-1}})$, and then proceeds as follows:

- Reconstruct the input list $\mathbf{T}$ using the honest parties' shares.
- Set $\pi = \pi_3 \circ \pi_2 \circ \pi_1$ and compute $\mathbf{T}_O = \pi(\mathbf{T})$.
- Set $\mathbf{T}_{O_{c+1}} = \mathbf{T}_O - (\mathbf{T}_{O_c} + \mathbf{T}_{O_{c-1}})$.
- Receive a command in $\{\text{Abort}, \text{Accept}\}$ from $\mathcal{S}$. If it receives **Abort**, it sends **Abort** to all honest parties. Otherwise, it sends $(\text{Output}, \mathbf{T}_{O_i}, \mathbf{T}_{O_{i-1}})$ to each honest party P_i .

To obtain malicious security, we augment this protocol with verification mechanisms that ensure the correctness of every message sent during execution. Concretely, for each message computed by a party, we require the sender to additionally provide a proof that the computation was performed

correctly. These proofs are jointly verified by the other parties, thereby ensuring that any deviation from the prescribed computation is detected.

A naive approach to such verification would incur linear overhead in the size of the input list. To achieve better efficiency, we rely on the distributed-verifier zero-knowledge (DVZK) techniques of [BGIN19], which enable sublinear-cost verification of degree-2 relations. In particular, these techniques allow efficient verification of multiplications or inner-product relations of the form $A \cdot B = c$, where the values are secret-shared among the parties.

However, the messages in the shuffle protocol involve applying secret permutations, and verifying the correctness of such operations does not directly reduce to standard inner-product checks. To address this, we first reduce the correctness of each message in the shuffle protocol to verifying inner product relations (i.e $A \cdot B = c$), where all values are known to the prover (i.e the party who sends the message), A and B are known to different verifiers, and c is additively shared between them. We then design a tailored DVZK protocol, adapted from [BGIN19], to verify these relations in sublinear cost. This allows us to enforce the correctness of all messages while preserving the efficiency of the underlying semi-honest protocol.

In what follows, we first recall the semi-honest shuffle protocol of [AFO⁺21]. We then show how verifying the correctness of each message can be reduced to verifying inner-product relations, and describe our DVZK protocol for efficiently verifying these checks.

4.1 Semi-honest shuffle protocol of [AFO⁺21]

We now recall the semi-honest 3-party shuffle protocol $\Pi_{\text{SH-Shuffle}}$ of [AFO⁺21]. The protocol takes as input $\langle \cdot \rangle$ -shares of a list T of size n , where $T = T_1 + T_2 + T_3$, and a permutation π_i from parties $P_i, P_{i+1} \in \mathcal{P}$. It outputs $\langle \cdot \rangle$ -shares of $T_O = \pi(T)$ such that $\pi = \pi_3 \circ \pi_2 \circ \pi_1$.

At a high level, the protocol implements a random permutation of the secret-shared list T by applying a sequence of three permutations π_1, π_2, π_3 , where each permutation is known only to a distinct pair of parties. The key idea is to ensure that no single party learns the overall permutation $\pi = \pi_3 \circ \pi_2 \circ \pi_1$, while still allowing the parties to correctly transform the shared input into a shuffled output. Two of the output shares are fixed in advance by sampling them uniformly at random: parties P_1, P_2 jointly sample T_{O1} , and parties P_2, P_3 jointly sample T_{O2} . Thus, the task reduces to securely computing the third share $T_{O3} = \pi(T) - T_{O1} - T_{O2}$, which will be held by P_1 and P_3 . Further, to preserve privacy, all intermediate values exchanged during the computation of T_{O3} are carefully masked using the randomness, such that the masks cancel out only at the end of the computation.

In the first round, each pair of parties P_i, P_{i+1} jointly sample a random list R_i . Additionally, P_1, P_2 jointly sample T_{O1} and P_2, P_3 jointly sample T_{O2} which corresponds to the two random output shares. Party P_2 combines its input shares (T_1, T_2) with randomness (R_1) and applies π_1 to compute $X_1 = \pi_1(T_1 + T_2 + R_1)$. Following this, P_2 computes $X_2 = \pi_2(X_1 + R_2)$, which it sends to P_1 . In parallel, P_1 applies π_1 on its share T_3 followed by removing the mask with R_1 (added by P_2 in his computation) to compute Y_1 , which it sends to P_3 . In the second round, P_1 applies π_3 to X_2 (after adding the mask R_3) to obtain X_3 , and then masks it using T_{O1} to compute Z_1 , which is sent to P_3 . Meanwhile, P_3 continues the permutation chain on Y_1 by applying π_2 and π_3 , while appropriately removing the masks R_2 and R_3 to obtain Y_3 . It then computes Z_2 by subtracting T_{O2} and sends it to P_1 . Finally, the parties P_1 and P_3 compute their common share of the output $T_{O3} = Z_1 + Z_2$. The formal details of the protocol appear below.

Protocol 4.2. $\Pi_{\text{SH-Shuffle}}$

Input: Parties hold $\langle \cdot \rangle$ -shares of a list $\mathbf{T}$ of size n . Additionally, party P_i, P_{i+1} hold a permutation π_i .

Output: Parties output $\langle \cdot \rangle$ -shares of a list $\mathbf{T}_O = \pi(\mathbf{T})$ where $\pi = \pi_3 \circ \pi_2 \circ \pi_1$.

1. **Round 1**

- (a) Parties P_i, P_{i+1} for $i \in \{1, 2, 3\}$ invoke $\mathcal{F}_{\text{Rand}}$ to jointly sample a random list $\mathbf{R}_i$.
- (b) Parties P_1, P_2 invoke $\mathcal{F}_{\text{Rand}}$ to jointly sample a random list $\mathbf{T}_{O1}$. Parties P_2, P_3 invoke $\mathcal{F}_{\text{Rand}}$ to jointly sample a random list $\mathbf{T}_{O2}$.
- (c) Party P_2 computes $\mathbf{X}_1 = \pi_1(\mathbf{T}_1 + \mathbf{T}_2 + \mathbf{R}_1)$ and $\mathbf{X}_2 = \pi_2(\mathbf{X}_1 + \mathbf{R}_2)$. P_2 sends $\mathbf{X}_2$ to P_1 .
- (d) Party P_1 computes $\mathbf{Y}_1 = \pi_1(\mathbf{T}_3 - \mathbf{R}_1)$ and sends $\mathbf{Y}_1$ to P_3 .

2. **Round 2**

- (a) Party P_1 computes $\mathbf{X}_3 = \pi_3(\mathbf{X}_2 + \mathbf{R}_3)$ and $\mathbf{Z}_1 = \mathbf{X}_3 - \mathbf{T}_{O1}$ and sends $\mathbf{Z}_1$ to P_3 .
- (b) Party P_3 computes $\mathbf{Y}_2 = \pi_2(\mathbf{Y}_1 - \mathbf{R}_2)$ and $\mathbf{Y}_3 = \pi_3(\mathbf{Y}_2 - \mathbf{R}_3)$.
- (c) Party P_3 computes $\mathbf{Z}_2 = \mathbf{Y}_3 - \mathbf{T}_{O2}$ and sends $\mathbf{Z}_2$ to P_1 .
- (d) Parties P_1 and P_3 compute $\mathbf{T}_{O3} = \mathbf{Z}_1 + \mathbf{Z}_2$.
- (e) Party P_i outputs $(\mathbf{T}_{O_i}, \mathbf{T}_{O_{i-1}})$.

At the end of the protocol, each party P_i locally holds $(\mathbf{T}_{O_i}, \mathbf{T}_{O_{i-1}})$, which together form valid $\langle \cdot \rangle$ -shares of the shuffled table $\mathbf{T}_O = \pi(\mathbf{T})$, where $\pi = \pi_3 \circ \pi_2 \circ \pi_1$. Specifically, observe that $\mathbf{T}_O = \mathbf{T}_{O1} + \mathbf{T}_{O2} + \mathbf{T}_{O3} \implies \mathbf{T}_O = \mathbf{T}_{O1} + \mathbf{T}_{O2} + \mathbf{Z}_1 + \mathbf{Z}_2$. Substituting $\mathbf{Z}_1 = \mathbf{X}_3 - \mathbf{T}_{O1}$ and $\mathbf{Z}_2 = \mathbf{Y}_3 - \mathbf{T}_{O2}$, we get

$$\mathbf{T}_O = \mathbf{T}_{O1} + \mathbf{T}_{O2} + (\mathbf{X}_3 - \mathbf{T}_{O1}) + (\mathbf{Y}_3 - \mathbf{T}_{O2}) = \mathbf{X}_3 + \mathbf{Y}_3.$$

Substituting for $\mathbf{X}_3$ and $\mathbf{Y}_3$ and using linearity of permutations, we get

$$\begin{aligned} \mathbf{X}_3 + \mathbf{Y}_3 &= \pi_3 \left(\pi_2 (\pi_1 (\mathbf{T}_1 + \mathbf{T}_2 + \mathbf{R}_1) + \mathbf{R}_2) + \mathbf{R}_3 \right) \\ &\quad + \pi_3 \left(\pi_2 (\pi_1 (\mathbf{T}_3 - \mathbf{R}_1) - \mathbf{R}_2) - \mathbf{R}_3 \right) \\ &= \pi_3 \left(\pi_2 (\pi_1 (\mathbf{T}_1 + \mathbf{T}_2 + \mathbf{T}_3)) \right) \\ &= \pi_3 \circ \pi_2 \circ \pi_1 (\mathbf{T}). \end{aligned}$$

We note that our presentation is equivalent to the protocol in [AFO⁺21, Fig. 5] up to a relabelling of parties and intermediate randomness. Specifically, the parties (S_1, S_2, S_3) in [AFO⁺21] correspond to (P_2, P_1, P_3) in our notation, respectively. Under this mapping, the party that is active only in the first round is P_2 . The random lists $\mathbf{R}_1, \mathbf{R}_2, \mathbf{R}_3$ correspond to the three internal masks used in [AFO⁺21], while $\mathbf{T}_{O1}$ and $\mathbf{T}_{O2}$ correspond to the two independently sampled output masks. The remaining output share $\mathbf{T}_{O3}$ is computed from the exchanged messages in both descriptions.

4.2 Achieving malicious security

In this section, we describe the protocol for malicious security in the $\mathcal{F}_{\text{VerifyIP}}$ -hybrid. This functionality allows the parties to verify the correctness of an inner-product relation in a distributed

manner. It takes as input a designated prover P_i that holds vectors (A, B) and a scalar c and verifiers P_{i-1} that holds A, c_{i-1} and P_{i+1} that holds B, c_{i+1} such that $c = c_{i-1} + c_{i+1}$. The functionality checks whether $A \cdot B = c_{i-1} + c_{i+1}$. If the check fails, or if the adversary signals **Abort**, the functionality outputs **Abort** to all honest parties; otherwise, it outputs **Accept**. Thus, $\mathcal{F}_{\text{VerifyIP}}$ guarantees that the claimed inner-product relation holds, while revealing no additional information. The formal details appear below.

Functionality 4.3. $\mathcal{F}_{\text{VerifyIP}}$

Without loss of generality, let $P_{\text{cor}} \in \mathcal{P}$ denote the party corrupted by adversary $\mathcal{S}$. $\mathcal{F}_{\text{VerifyIP}}$ receives from honest parties the prover's identity $j \in \{1, 2, 3\}$. Then:

- If P_j is honest, then $\mathcal{F}_{\text{VerifyIP}}$ receives A, B, c from P_j and sends $\mathcal{S}$ the inputs of P_{cor} .
- If P_j is corrupt then $\mathcal{F}_{\text{VerifyIP}}$ receives A, c_{j-1} from P_{j-1} and B, c_{j+1} from P_{j+1} . It hands over A, B, c to $\mathcal{S}$.

$\mathcal{F}_{\text{VerifyIP}}$ proceeds as follows.

- If $A \cdot B \neq c$ $\mathcal{F}_{\text{VerifyIP}}$ sends **Abort** to all parties. Else it receives a command in $\{\text{Abort}, \text{Accept}\}$ from $\mathcal{S}$. If the message is **Abort**, it sends **Abort** to the honest parties. Else it sends $(\text{Output}, \text{Accept})$ to all honest parties.

At a high level, our maliciously secure protocol involves running the semi-honest protocol of [AFO⁺21], verifying the correctness of each message sent in the protocol by reducing it to an inner-product check, and then using $\mathcal{F}_{\text{VerifyIP}}$ to verify each inner-product check.

We next show how each message in the semi-honest protocol of [AFO⁺21] can be reduced to verifying an inner-product relation. Recall that the protocol involves four messages: P_2 sends X_2 , P_1 sends Y_1 and Z_1 , and P_3 sends Z_2 .

Verifying the message X_2 sent by P_2 . Let X_2 represent the message that P_2 should send if it follows the protocol and let $\hat{X}_2$ represent the message that P_2 actually sends. Recall that $X_2 = \pi_2(X_1 + R_2)$, where $X_1 = \pi_1(T_1 + T_2 + R_1)$. Expanding X_2 , we get

$$X_2 = \pi_2(\pi_1(T_1 + T_2 + R_1) + R_2)$$

Applying π_2^{-1} on both sides,

$$\pi_2^{-1}(X_2) = \pi_1(T_1 + T_2 + R_1) + R_2$$

Rearranging, we obtain

$$\pi_2^{-1}(X_2) - R_2 = \pi_1(T_1 + T_2 + R_1).$$

If the statement above also holds for $\hat{X}_2$, then $\hat{X}_2 = X_2$. To verify this efficiently, we use a random linear projection. Let $M^1 \xleftarrow{\$} \mathbb{F}^n$ be a random vector. Taking inner products with M^1 , we obtain

$$M^1 \cdot (\pi_2^{-1}(X_2) - R_2) = M^1 \cdot \pi_1(T_1 + T_2 + R_1).$$

Expanding,

$$M^1 \cdot \pi_2^{-1}(X_2) - M^1 \cdot R_2 = M^1 \cdot \pi_1(T_1 + R_1) + M^1 \cdot \pi_1(T_2).$$

Using the permutation property of inner products,

$$M^1 \cdot \pi_2^{-1}(X_2) = \pi_2(M^1) \cdot X_2.$$

$$\mathbf{M}^1 \cdot \pi_1(\mathbf{T}_2) = \pi_1^{-1}(\mathbf{M}^1) \cdot \mathbf{T}_2$$

Thus,

$$\pi_2(\mathbf{M}^1) \cdot \mathbf{X}_2 - \mathbf{M}^1 \cdot \mathbf{R}_2 = \mathbf{M}^1 \cdot \pi_1(\mathbf{T}_1 + \mathbf{R}_1) + \pi_1^{-1}(\mathbf{M}^1) \cdot \mathbf{T}_2.$$

Rearranging this, we get

$$\underbrace{\mathbf{X}_2}_{P_1} \cdot \underbrace{\pi_2(\mathbf{M}^1)}_{P_3} - \underbrace{\pi_1^{-1}(\mathbf{M}^1)}_{P_1} \cdot \underbrace{\mathbf{T}_2}_{P_3} = \underbrace{\mathbf{M}^1 \cdot \pi_1(\mathbf{T}_1 + \mathbf{R}_1)}_{P_1} + \underbrace{\mathbf{M}^1 \cdot \mathbf{R}_2}_{P_3}.$$

Observe that the above relation is of the form $\mathbf{A}_1 \cdot \mathbf{B}_1 + \mathbf{A}_2 \cdot \mathbf{B}_2 = \mathbf{c}$, where the prover P_2 knows all the terms, the verifier P_1 holds $\mathbf{A}_1 = \mathbf{X}_2$, $\mathbf{A}_2 = -\pi_1^{-1}(\mathbf{M}^1)$ and verifier P_3 holds $\mathbf{B}_1 = \pi_2(\mathbf{M}^1)$, $\mathbf{B}_2 = \mathbf{T}_2$. Further, both P_1 and P_3 jointly hold $\mathbf{c} = \mathbf{M}^1 \cdot \pi_1(\mathbf{T}_1 + \mathbf{R}_1) + \mathbf{M}^1 \cdot \mathbf{R}_2$. Note that this can be reduced to a single inner product check by considering $\mathbf{A} = \mathbf{A}_1 || \mathbf{A}_2$ and $\mathbf{B} = \mathbf{B}_1 || \mathbf{B}_2$, where $||$ denotes the concatenation operation. Hence, correctness of $\mathbf{X}_2$ reduces to verifying an inner-product relation using $\mathcal{F}_{\text{VerifyIP}}$.

If the adversary sends an incorrect message (i.e. $\hat{\mathbf{X}}_2 \neq \mathbf{X}_2$), the inner-product relation will only hold if

$$\hat{\mathbf{X}}_2 \cdot \pi_2(\mathbf{M}^1) = \mathbf{X}_2 \cdot \pi_2(\mathbf{M}^1)$$

By the random projection property, the probability that this holds for a random $\mathbf{M}^1 \in \mathbb{F}^n$ is $\frac{1}{|\mathbb{F}|}$.

Verifying the messages sent by P_1 . P_1 sends two messages: $\mathbf{Y}_1$ and $\mathbf{Z}_1$.

(1) *Verifying $\mathbf{Y}_1$.* Recall that

$$\mathbf{Y}_1 = \pi_1(\mathbf{T}_3 - \mathbf{R}_1).$$

Taking inner product with a random vector $\mathbf{M}^2 \xleftarrow{\$} \mathbb{F}^n$,

$$\mathbf{M}^2 \cdot \mathbf{Y}_1 = \mathbf{M}^2 \cdot \pi_1(\mathbf{T}_3) - \mathbf{M}^2 \cdot \pi_1(\mathbf{R}_1).$$

Using the permutation property of inner products,

$$\mathbf{M}^2 \cdot \pi_1(\mathbf{T}_3) = \pi_1^{-1}(\mathbf{M}^2) \cdot \mathbf{T}_3.$$

$$\mathbf{M}^2 \cdot \mathbf{Y}_1 = \pi_1^{-1}(\mathbf{M}^2) \cdot \mathbf{T}_3 - \mathbf{M}^2 \cdot \pi_1(\mathbf{R}_1),$$

which can be rearranged as

$$\underbrace{\mathbf{T}_3}_{P_3} \cdot \underbrace{\pi_1^{-1}(\mathbf{M}^2)}_{P_2} = \underbrace{\mathbf{M}^2 \cdot \mathbf{Y}_1}_{P_3} + \underbrace{\mathbf{M}^2 \cdot \pi_1(\mathbf{R}_1)}_{P_2}.$$

Observe that this is an inner-product relation of the form $\mathbf{A} \cdot \mathbf{B} = \mathbf{c}$, where the prover P_1 knows all the terms, the verifier P_3 holds $\mathbf{A} = \mathbf{T}_3$ and verifier P_2 holds $\mathbf{B} = \pi_1^{-1}(\mathbf{M}^2)$. Further, both P_3 and P_2 jointly hold $\mathbf{c} = \mathbf{M}^2 \cdot \mathbf{Y}_1 + \mathbf{M}^2 \cdot \pi_1(\mathbf{R}_1)$. Hence, verifying the correctness of $\mathbf{Y}_1$ reduces to verifying an inner-product relation, which can be done using $\mathcal{F}_{\text{VerifyIP}}$.

(2) *Verifying $\mathbf{Z}_1$.* Recall that

$$\mathbf{Z}_1 = \pi_3(\mathbf{X}_2 + \mathbf{R}_3) - \mathbf{T}_{O1}.$$

Taking inner product with a random $\mathbf{M}^3$,

$$\mathbf{M}^3 \cdot \mathbf{Z}_1 = \mathbf{M}^3 \cdot \pi_3(\mathbf{X}_2) + \mathbf{M}^3 \cdot \pi_3(\mathbf{R}_3) - \mathbf{M}^3 \cdot \mathbf{T}_{O1}.$$

Using the permutation property, we get

$$\mathbf{M}^3 \cdot \mathbf{Z}_1 = \pi_3^{-1}(\mathbf{M}^3) \cdot \mathbf{X}_2 + \mathbf{M}^3 \cdot \pi_3(\mathbf{R}_3) - \mathbf{M}^3 \cdot \mathbf{T}_{O1}.$$

Thus, correctness reduces to checking

$$\underbrace{\pi_3^{-1}(\mathbf{M}^3)}_{P_3} \cdot \underbrace{\mathbf{X}_2}_{P_2} = \underbrace{\mathbf{M}^3 \cdot (\mathbf{Z}_1 - \pi_3(\mathbf{R}_3))}_{P_3} + \underbrace{\mathbf{M}^3 \cdot \mathbf{T}_{O1}}_{P_2}$$

In this way, the correctness of $\mathbf{Z}_1$ also reduces to verifying an inner-product relation via $\mathcal{F}_{\text{VerifyIP}}$.

Since the verifications of $\mathbf{Y}_1$ and $\mathbf{Z}_1$ both use random projections, it suffices to perform a single inner product check, similar to that of $\mathbf{X}_2$, by concatenating the input arrays. Letting $\hat{\mathbf{Y}}_1$ and $\hat{\mathbf{Z}}_1$ be the actual messages sent by a potentially malicious P_1 , the inner product will hold exactly when

$$(\hat{\mathbf{Y}}_1 || \hat{\mathbf{Z}}_1) \cdot (\mathbf{M}^2 || \mathbf{M}^3) = (\mathbf{Y}_1 || \mathbf{Z}_1) \cdot (\mathbf{M}^2 || \mathbf{M}^3)$$

Since $\mathbf{M}^2$ and $\mathbf{M}^3$ are chosen at random, if either $\hat{\mathbf{Y}}_1 \neq \mathbf{Y}_1$ or $\hat{\mathbf{Z}}_1 \neq \mathbf{Z}_1$, this verification will fail except with probability $\frac{1}{|\mathbb{F}|}$.

Verifying the message sent by P_3 . Recall that P_3 sends the message $\mathbf{Z}_2$, where

$$\mathbf{Z}_2 = \mathbf{Y}_3 - \mathbf{T}_{O2} = \pi_3(\mathbf{Y}_2 - \mathbf{R}_3) - \mathbf{T}_{O2}.$$

Applying π_3^{-1} on both sides, we obtain

$$\pi_3^{-1}(\mathbf{Z}_2) = \mathbf{Y}_2 - \mathbf{R}_3 - \pi_3^{-1}(\mathbf{T}_{O2}).$$

Substituting $\mathbf{Y}_2 = \pi_2(\mathbf{Y}_1 - \mathbf{R}_2)$, we get

$$\pi_3^{-1}(\mathbf{Z}_2) = \pi_2(\mathbf{Y}_1 - \mathbf{R}_2) - \mathbf{R}_3 - \pi_3^{-1}(\mathbf{T}_{O2}) \tag{1}$$

$$\pi_3^{-1}(\mathbf{Z}_2) = \pi_2(\mathbf{Y}_1) - \pi_2(\mathbf{R}_2) - \mathbf{R}_3 - \pi_3^{-1}(\mathbf{T}_{O2}) \tag{2}$$

$$\pi_2(\mathbf{Y}_1) - \pi_3^{-1}(\mathbf{T}_{O2}) = \pi_2(\mathbf{R}_2) + \pi_3^{-1}(\mathbf{Z}_2) + \mathbf{R}_3. \tag{3}$$

To verify this relation efficiently, we take inner products with a random vector $\mathbf{M}^4 \xleftarrow{\$} \mathbb{F}^n$:

$$\mathbf{M}^4 \cdot \pi_2(\mathbf{Y}_1) - \mathbf{M}^4 \cdot \pi_3^{-1}(\mathbf{T}_{O2}) = \mathbf{M}^4 \cdot \pi_2(\mathbf{R}_2) + \mathbf{M}^4 \cdot \pi_3^{-1}(\mathbf{Z}_2) + \mathbf{M}^4 \cdot \mathbf{R}_3.$$

Using the permutation property of inner products, we obtain,

$$\underbrace{\pi_2^{-1}(\mathbf{M}^4)}_{P_2} \cdot \underbrace{\mathbf{Y}_1}_{P_1} - \underbrace{\mathbf{T}_{O2}}_{P_2} \cdot \underbrace{\pi_3(\mathbf{M}^4)}_{P_1} = \underbrace{\mathbf{M}^4 \cdot \pi_2(\mathbf{R}_2)}_{P_2} + \underbrace{\mathbf{M}^4 \cdot (\pi_3^{-1}(\mathbf{Z}_2) + \mathbf{R}_3)}_{P_1}.$$

Similar to the verification of the message $\mathbf{X}_2$, the correctness of $\mathbf{Z}_2$ reduces to verifying an inner-product relation using $\mathcal{F}_{\text{VerifyIP}}$. Due to the random projection, an erroneous message will result in a false inner product claim, except with probability $\frac{1}{|\mathbb{F}|}$.

The formal protocol for malicious secure shuffle appears below.

Protocol 4.4. Π_{Shuffle}

Input: Parties hold $\langle \cdot \rangle$ -shares of a list T of size n . Additionally, party P_i, P_{i+1} hold a permutation π_i .

Output: Parties output $\langle \cdot \rangle$ -shares of a list $\mathsf{T}_O = \pi(\mathsf{T})$ where $\pi = \pi_3 \circ \pi_2 \circ \pi_1$.

1. Semi-honest shuffle

(a) Parties run the semi-honest shuffle protocol described in Protocol 4.2

2. Verification

- (a) Parties invoke $\mathcal{F}_{\text{Coin}}$ to sample $\mathsf{M}^1, \mathsf{M}^2, \mathsf{M}^3, \mathsf{M}^4 \in \mathbb{F}^n$, where $\mathbb{F}$ is either the same field as the items of T or a field extension thereof such that $|\mathbb{F}| \geq 2^\kappa$.
- (b) Parties P_2, P_1 compute $\mathsf{A}_1 = \mathsf{X}_2 || (-\pi_1^{-1}(\mathsf{M}^1))$ and $\mathsf{c}_1^1 = \mathsf{M}^1 \cdot \pi_1(\mathsf{T}_1 + \mathsf{R}_1)$. Parties P_2, P_3 compute $\mathsf{B}_1 = \pi_2(\mathsf{M}^1) || \mathsf{T}_2$ and $\mathsf{c}_3^1 = \mathsf{M}^1 \cdot \mathsf{R}_2$. Parties invoke $\mathcal{F}_{\text{VerifyIP}}$ with P_2 as the prover to verify $\mathsf{A}^1 \cdot \mathsf{B}^1 = \mathsf{c}_1^1 + \mathsf{c}_3^1$.
- (c) Parties P_1, P_3 compute $\mathsf{A}^2 = \mathsf{T}_3 || \pi_3^{-1}(\mathsf{M}^3)$ and $\mathsf{c}_3^2 = \mathsf{M}^2 \cdot \mathsf{Y}_1 + \mathsf{M}^3 \cdot (\mathsf{Z}_1 - \pi_3(\mathsf{R}_3))$. Parties P_1, P_2 compute $\mathsf{B}^2 = \pi_1^{-1}(\mathsf{M}^2) || \mathsf{X}_2$ and $\mathsf{c}_2^2 = \mathsf{M}^2 \cdot \pi_1(\mathsf{R}_1) + \mathsf{M}^3 \cdot \mathsf{T}_{O1}$. Parties invoke $\mathcal{F}_{\text{VerifyIP}}$ with P_1 as the prover to verify $\mathsf{A}^2 \cdot \mathsf{B}^2 = \mathsf{c}_3^2 + \mathsf{c}_2^2$.
- (d) Parties P_3, P_2 compute $\mathsf{A}^3 = \pi_2^{-1}(\mathsf{M}^4) || (-\mathsf{T}_{O2})$ and $\mathsf{c}_2^3 = \mathsf{M}^4 \cdot \pi_2(\mathsf{R}_2)$. Parties P_3, P_1 compute $\mathsf{B}^3 = \mathsf{Y}_1 || \pi_3(\mathsf{M}^4)$ and $\mathsf{c}_1^3 = \mathsf{M}^4 \cdot (\pi_3^{-1}(\mathsf{Z}_2) + \mathsf{R}_3)$. Parties invoke $\mathcal{F}_{\text{VerifyIP}}$ with P_3 as the prover to verify $\mathsf{A}^3 \cdot \mathsf{B}^3 = \mathsf{c}_2^3 + \mathsf{c}_1^3$.
- (e) If any instance of the $\mathcal{F}_{\text{VerifyIP}}$ returns **Abort**, parties output **Abort**. Else, party P_i outputs $(\mathsf{T}_{O_i}, \mathsf{T}_{O_{i-1}})$.

Claim 4.5. *The protocol Π_{Shuffle} incurs an amortised round complexity of 2 rounds and a communication complexity of $4n\ell + (12\sqrt{n} + 9)t + 3\lambda + 3$ bits, where n denotes the size of the input list T , where each element is from an ℓ -bit field and where $t = \max(\ell, \kappa)$, for statistical security parameter κ .*

Proof. The semi-honest shuffle phase, i.e., Protocol 4.2, consists of 2 rounds. Its communication comprises the four list $\mathsf{X}_2, \mathsf{Y}_1, \mathsf{Z}_1$, and Z_2 , each of size n field elements. Therefore, this step incurs 2 rounds and $4n\ell$ bits of communication. The verification phase consists of parties sampling four random lists using $\mathcal{F}_{\text{Coin}}$ followed by three parallel invocations of $\mathcal{F}_{\text{VerifyIP}}$. $\mathcal{F}_{\text{Coin}}$ requires 1 round and communication of 3λ bits, where λ is the computational security parameter. Looking ahead, instantiating $\mathcal{F}_{\text{VerifyIP}}$ using Protocol 5.2 incurs 2 rounds and communication of $(2m + l + 3)t + 1$ bits for verifying an inner product of dimension ml on a field of size t . If $\ell \geq \kappa$, the field size of table elements is large enough for the security to hold. Otherwise, the parties *locally* use field extensions to convert the message elements to be from a sufficiently large field, and execute the verification in this field. In Protocol Π_{Shuffle} , we require three such invocations, each of dimension $2n$. Setting $m = \sqrt{n}$ and $l = 2\sqrt{n}$, the total communication in the verification phase becomes

$$3\lambda + 3(2\sqrt{n} + 2\sqrt{n} + 3)t + 3 = 3\lambda + (12\sqrt{n} + 9)t + 3$$

Combining both phases, a single invocation of Π_{Shuffle} incurs 5 rounds (2 from the shuffle phase and 3 from the verification phase) and a total communication of $4n\ell + 3\lambda + (12\sqrt{n} + 9)t + 3$. When multiple shuffle instances are executed, their verification phases can be deferred and performed

in parallel. Since all $\mathcal{F}_{\text{VerifyIP}}$ invocations can be batched across instances, the verification phase contributes only a constant number of rounds overall. Consequently, Π_{Shuffle} achieves an amortised round complexity of 2, matching that of the semi-honest shuffle. $\square$

Shuffle in the preprocessing paradigm. We note that our protocol is designed in the all-online model. In contrast, works such as [KKP⁺23] adopt the function-dependent preprocessing paradigm, wherein input-independent computation is offloaded to a preprocessing phase. This enables a highly efficient online phase with amortised cost of 2 rounds and $3n$ field elements of communication, along with a verification cost that, when batched across multiple shuffles, requires 2 rounds and only constant communication. However, their preprocessing phase relies on invoking the shuffle protocol of [AFO⁺21] to generate correlated randomness. Thus, in the preprocessing phase, it incurs a round complexity of 7 rounds and a communication complexity of $6n\ell + 12n\kappa + 3\kappa$, where κ denotes the statistical security parameter. We observe that our protocol can be used as a drop-in replacement for this component, thereby improving the preprocessing efficiency of such constructions. This yields a protocol that retains the fast online phase of the preprocessing paradigm while achieving improved overall complexity. Specifically, it improves the preprocessing cost to 5 rounds and $4n\ell + (12\sqrt{n} + 9)t + 3\lambda + 3$ communicated bits.

Circular shifts using shuffle. Circular shifts can be realised using the shuffle protocol. Let parties hold $\langle \cdot \rangle$ -shares of a list $\mathbf{T}$ of size n , and additionally $\langle \cdot \rangle$ -shares of a value $x \in \{0, \dots, n-1\}$. The goal is to output $\langle \cdot \rangle$ -shares of $\mathbf{T}_O = (\mathbf{T} \gg x)$, where $\gg$ denotes a cyclic right shift. We abstract this as the functionality $\mathcal{F}_{\text{Shift}}$. At a high level, a cyclic shift can be viewed as a permutation over $[n]$ that maps index i to $(i+x) \bmod n$. To realise this securely, we decompose x into shares across the parties and let each pair of parties locally derive a rotation permutation corresponding to their share. These pairwise permutations compose to the global rotation permutation corresponding to x . The parties then invoke the shuffle protocol with these permutations as inputs, thereby applying the desired circular shift to $\mathbf{T}$ while preserving the privacy of x . By leveraging the composability of permutations, this yields a correct and secure realisation of $\mathcal{F}_{\text{Shift}}$.

4.3 Security of Π_{Shuffle}

The proof of security of Π_{Shuffle} is presented in Appendix B.

4.4 Correlated Shuffle

In many applications, e.g. secure graph analysis [AFO⁺21, KKPG24], the same random permutation is applied to multiple input vectors (both sequentially and in parallel). We refer to this setting as the *correlated shuffle* setting, where multiple shuffle instances share a common underlying permutation. In such scenarios, verifying each shuffle independently incurs a linear overhead in the number of instances. We show how to batch-verify multiple correlated shuffle instances, thereby significantly reducing the verification cost.

Suppose a party (say P_j) sends messages $\mathbf{Y}^{(1)}, \mathbf{Y}^{(2)}, \dots, \mathbf{Y}^{(k)}$, each corresponding to a shuffle instance where the same permutation π is applied. For concreteness, consider the first message ($\mathbf{Y}_1$ in Protocol 4.2) sent by P_1 in the j^{th} instance of shuffle, which is of the form

$$\mathbf{Y}_1^j = \pi_1(\mathbf{T}_3^j - \mathbf{R}_1^j) \quad j \in [k],$$

where $\mathbf{T}_3^j$ denotes the share of the input table held by P_3, P_1 in the j^{th} instance, and $\mathbf{R}_1^j$ is the corresponding randomness sampled jointly by the relevant parties. Following our previous approach,

each equation can be verified independently by sampling a random vector $M^{(j)}$ and checking

$$M^j \cdot Y_1^j = \pi_1^{-1}(M^j) \cdot T_3^j - M^j \cdot R_1^j,$$

resulting in k invocations of $\mathcal{F}_{\text{VerifyIP}}$.

Batching via random linear combination. To reduce this cost, we aggregate all k relations into a single check using a random linear combination. Let $\alpha_1, \dots, \alpha_k \xleftarrow{\$} \mathbb{F}$ be random scalars. We define the aggregated vectors:

$$Y = \sum_{j=1}^k \alpha_j Y^j, \quad T_3 = \sum_{j=1}^k \alpha_j T_3^j, \quad R_1 = \sum_{j=1}^k \alpha_j R_1^j.$$

Thus, the correctness of all k relations reduces to checking a single relation $Y_1 = \pi_1(T_3 - R_1)$, which can be reduced to checking a single inner-product relation that can be verified using a single invocation of $\mathcal{F}_{\text{VerifyIP}}$. Observe that, if any of the k relations is incorrect, then the aggregated relation fails with probability at least $1 - \frac{1}{|\mathbb{F}|}$ over the choice of $\{\alpha_\ell\}$, by a standard Schwartz-Zippel argument.

5 Instantiating $\mathcal{F}_{\text{VerifyIP}}$

We now describe a distributed verifier zero-knowledge (DVZK) protocol for verifying inner-product relations. Our construction closely follows the polynomial-based DVZK constructions of [BGIN19], adapted to our setting. Let P_i denote the prover, and P_{i-1}, P_{i+1} denote the two verifiers. The prover holds vectors $A, B \in \mathbb{F}^n$ and a value $c \in \mathbb{F}$ such that $c = A \cdot B$. The verifier P_{i-1} knows A and a share c_1 , while P_{i+1} knows B and a share c_2 , where $c = c_1 + c_2$.

Our goal is to verify that $\sum_{t=1}^n a_t b_t = c$, while ensuring soundness against a malicious prover. For this, we partition the vectors A and B into m blocks of size l and view them as $m \times l$ arrays. More concretely, for each $k \in [m]$, the k -th row consists of the entries $a_{(k-1)l+1}, \dots, a_{kl}$ and $b_{(k-1)l+1}, \dots, b_{kl}$. Thus, $A \cdot B = \sum_{k=1}^m \sum_{j=1}^l a_{(k-1)l+j} b_{(k-1)l+j}$.

We encode the columns of the $m \times l$ arrays using l polynomials for A and l polynomials for B . For each $j \in [l]$, we define polynomials f_j, g_j of degree at most m as follows:

- for every $k \in [m]$,

$$f_j(k) = a_{(k-1)l+j}, \quad g_j(k) = b_{(k-1)l+j},$$

- the point $f_j(0)$ is jointly sampled by P_i and P_{i-1} ,
- the point $g_j(0)$ is jointly sampled by P_i and P_{i+1} .

Given these points, the prover P_i uniquely interpolates each f_j and g_j and, defines $h(x) = \sum_{j=1}^l f_j(x)g_j(x)$. Observe that h has degree at most $2m$, and for every $k \in [m]$,

$$h(k) = \sum_{j=1}^l a_{(k-1)l+j} b_{(k-1)l+j}.$$

Therefore,

$$A \cdot B = \sum_{k=1}^m h(k).$$

Thus, verifying $A \cdot B = c$ reduces to checking that $\sum_{k=1}^m h(k) = c$, where h is correctly formed from polynomial encodings of A and B . The prover computes the polynomial h and secret-shares

its coefficients among the verifiers. The verifiers then check the consistency of h with the encoded polynomials by evaluating at a random point.

Claim 5.1. *The protocol Π_{VerifyIP} incurs a round complexity of 2 rounds and a communication complexity of $(2m + l + 3)\ell + 1$ bits, where $ml = n$ denotes the size of the input lists $\mathbf{A}, \mathbf{B}$ where each element is from an ℓ -bit field.*

Proof. After generating the proof, P_i sends to P_{i+1} the coefficient vector $(h_0^{i+1}, \dots, h_{2m}^{i+1})$, that is $2m + 1$ field elements. P_i does not need to send anything to P_{i-1} because P_{i-1} 's share of the proof is generated using $\mathcal{F}_{\text{Rand}}$, which generates pairwise random values without interaction given a one-time setup (see §2). Additionally, P_{i-1} sends P_{i+1} the computed values $\{f_j(\mathbf{r})\}_{j \in [l]}$, $\mathbf{x}_{i-1}$ and $\mathbf{c}_{i-1}$, that is $l + 2$ field elements. Note that since P_{i-1} computes these based only on locally generated correlated randomness, P_{i-1} can send this message *in the same round* that P_i sends its message. Finally, P_{i+1} sends P_{i-1} a single bit indicating whether to abort the protocol. Thus, the protocol requires 2 rounds of interaction, with communication of $((2m + 1) + (l + 2))\ell + 1$ bits. Given $|\mathbf{A}| = |\mathbf{B}| = n = ml$, by setting parameters $m \approx l \approx \sqrt{n}$, the communication reduces to $\mathcal{O}(\sqrt{n}\ell)$. In this way, the communication cost is sublinear in the input size. $\square$

Protocol 5.2. Π_{VerifyIP}

Input: Prover P_i holds $\mathbf{A}, \mathbf{B}, \mathbf{c}$. Verifier P_{i-1} holds $\mathbf{A}, \mathbf{c}_{i-1}$ and verifier P_{i+1} holds $\mathbf{B}, \mathbf{c}_{i+1}$ such that $\mathbf{c} = \mathbf{c}_{i-1} + \mathbf{c}_{i+1}$.

Output: Output Accept if $\mathbf{A} \cdot \mathbf{B} = \mathbf{c}$; else output Abort.

1. **Proof Generation:** P_i generates the proof.

- (a) Parse $\mathbf{A} = (\mathbf{a}_1, \dots, \mathbf{a}_{ml})$ and $\mathbf{B} = (\mathbf{b}_1, \dots, \mathbf{b}_{ml})$.
- (b) For $j \in [l]$, P_i, P_{i-1} invoke $\mathcal{F}_{\text{Rand}}$ to jointly sample $\mathbf{w}_j \in \mathbb{F}$ and set $f_j(0) = \mathbf{w}_j$. For all $k \in [m]$, set $f_j(k) = \mathbf{a}_{(k-1)l+j}$ and interpolate f_j .
- (c) For $j \in [l]$, P_i, P_{i+1} invoke $\mathcal{F}_{\text{Rand}}$ jointly sample $\mathbf{w}'_j \in \mathbb{F}$ and set $g_j(0) = \mathbf{w}'_j$. For all $k \in [m]$, set $g_j(k) = \mathbf{b}_{(k-1)l+j}$ and interpolate g_j .
- (d) P_i computes

$$h(\mathbf{x}) = \sum_{j=1}^l f_j(\mathbf{x}) \cdot g_j(\mathbf{x}),$$

and obtains coefficients $(h_0, \dots, h_{2m})$.

- (e) P_i, P_{i-1} jointly sample $(h_0^{i-1}, \dots, h_{2m}^{i-1}) \in \mathbb{F}^{2m+1}$.
- (f) P_i sets

$$h_j^{i+1} = h_j - h_j^{i-1} \quad \forall j \in \{0, \dots, 2m\},$$

and sends $(h_0^{i+1}, \dots, h_{2m}^{i+1})$ to P_{i+1} .

2. **Verification Preparation:**

- (a) P_{i-1}, P_{i+1} invoke $\mathcal{F}_{\text{Rand}}$ to sample a random $\mathbf{r} \in \mathbb{F} \setminus \{0, 1, \dots, m\}$.
- (b) P_{i-1} computes:

$$f_j(\mathbf{r}) \quad \forall j \in [l], \quad \mathbf{x}_{i-1} = h^{i-1}(\mathbf{r}), \quad \mathbf{c}'_{i-1} = \mathbf{c}_{i-1} - \sum_{k=1}^m h^{i-1}(k).$$

(c) P_{i+1} computes:

$$g_j(r) \forall j \in [l], \quad x_{i+1} = h^{i+1}(r), \quad c'_{i+1} = c_{i+1} - \sum_{k=1}^m h^{i+1}(k).$$

(d) P_{i-1} sends $\{f_j(r)\}_{j \in [l]}$, x_{i-1} , and c'_{i-1} to P_{i+1} .

3. Final Verification:

(a) P_{i+1} checks:

$$x_{i-1} + x_{i+1} \stackrel{?}{=} \sum_{j=1}^l f_j(r) \cdot g_j(r), \tag{4}$$

$$c'_{i-1} + c'_{i+1} \stackrel{?}{=} 0. \tag{5}$$

(b) If any check fails, P_{i+1} sends **Abort** to P_{i-1} and aborts, otherwise it sends **Accept** to P_{i-1} and continues.

(c) If P_{i-1} receives **Abort** it aborts.

Security of Π_{VerifyIP} . We give the formal proof of security for Π_{VerifyIP} in Appendix B.

6 Implementation and Evaluation

We implement the malicious shuffle protocol in Rust. Our implementation operates over the 64-bit Goldilocks prime field $\mathbb{F}_p$, where $p = 2^{64} - 2^{32} + 1$, instantiated using the **ark-ff** crate. Polynomial operations, including evaluation, interpolation, and inner-product computation are performed using FFT-based methods provided by the **ark-poly** crate. This choice of prime field enables highly efficient NTT-based arithmetic, as its multiplicative group contains large power-of-two subgroups. To further optimize performance, we exploit multi-core parallelism via the **rayon** crate for computationally intensive tasks such as permutation, polynomial evaluation, and interpolation.

Experimental Setup. Our experimental environment closely matches, and is in some aspects slightly less powerful than, that of [AFO⁺21]. Specifically, we deploy three parties on AWS **c5.9xlarge** instances, equipped with Intel Xeon Platinum 8000 series processors (up to 3 GHz) and 68 GB of RAM. This setup mirrors the high-bandwidth, low-latency conditions used in [AFO⁺21], where machines featured up to 72 GB of RAM and clock speeds of up to 3.5 GHz. All reported runtimes are averaged over five independent executions, following the same methodology as [AFO⁺21].

Our protocol achieves strictly better round and communication complexity than [AFO⁺21] (see Table 1). To account for computational costs, we conduct benchmarks in a comparable (or slightly less favorable) environment. As the implementation of [AFO⁺21] is not publicly available, we rely on their reported results for comparison. Table 2 contains our benchmarks of runtime from these experiments compared with the reported benchmarks of [AFO⁺21], which confirms that even taking computation into consideration, our protocol is better for larger values of n by a factor of about 30%. While [AFO⁺21] did not report communication costs, we calculate these from their protocol under the parameter ranges of the experiment ($\ell = 64, \kappa = 48, k = 1$) and similarly calculate our

n	Runtime (ms)		Communication (MiB)	
	[AFO ⁺ 21]	This work	[AFO ⁺ 21]	This work
10,000	10.41	13.17	1.1	0.3
100,000	85.53	60.12	11.4	3.1
1,000,000	835.88	554.65	114.4	30.6

Table 2: Performance comparison with [AFO⁺21]

communication costs under the same parameter choices. We do not compare with [AHI⁺22], since their benchmarks do not include the cost of generating the authentication information.

Results. Table 2 summarizes the runtime performance of the malicious shuffle protocol. While our implementation incurs a modest overhead for small input sizes, it outperforms [AFO⁺21] for larger instances. In particular, we achieve a $1.42\times$ speedup at $N = 100,000$ and a $1.51\times$ speedup at $N = 1,000,000$, demonstrating improved scalability.

We emphasize that both implementations are evaluated under similar high-bandwidth, low-latency AWS conditions. In more typical deployment scenarios, such as bandwidth-constrained or geographically distributed settings, our protocol is expected to yield greater improvements. When $\ell \gg \kappa$, in an environment that is latency constrained, our protocol would outperform [AFO⁺21] by 28-33%, depending on the number of shuffle invocations. Similarly if $\ell \gg \kappa$ in a bandwidth-constrained environment, our protocol would outperform [AFO⁺21] by about 50%. If, instead $\kappa \gg \ell$, our protocol would provide much greater benefits. For the case of permuting bit strings ($\ell = 1$) and $\kappa = 60$, in a bandwidth-constrained environment, the performance improvement over [AFO⁺21] would be $90\times$.

7 Acknowledgements

This paper was edited for grammar using Grammarly and ChatGPT.

References

- [AFL⁺16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. pages 805–817, 2016.
- [AFO⁺21] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. pages 610–629, 2021.
- [AHI⁺22] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. Efficient secure three-party sorting with applications to data analysis and heavy hitters. pages 125–138, 2022.
- [BGIN19] Elette Boyle, Niv Gilboa, Yuval Ishai, and Ariel Nof. Practical fully secure three-party computation via sublinear distributed zero-knowledge proofs. pages 869–886, 2019.
- [BKNG26] Nidhish Bhimrajka, Yashvanth Kondi, Daniel Noble, and Bhavish Raj Gopal. Truffle: Maliciously secure three-party shuffles with applications to parsing. 2026.

- [BPRS23] Lennart Braun, Mahak Pancholi, Rahul Rachuri, and Mark Simkin. Ramen: Souper fast three-party computation for RAM programs. pages 3284–3297, 2023.
- [Can07] Ran Canetti. Obtaining universally composable security: Towards the bare bones of trust (invited talk). pages 88–112, 2007.
- [DFK⁺06] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. Unconditionally secure constant-rounds multi-party computation for equality, comparison, bits and exponentiation. pages 285–304, 2006.
- [FJKW15] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. pages 360–385, 2015.
- [JKK⁺22] Pranav Jangir, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Poster: Vogue: Faster computation of private heavy hitters. pages 3371–3373, 2022.
- [KKP⁺23] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, Somya Sangal, et al. Ruffle: Rapid 3-party shuffle protocols. *Proceedings on Privacy Enhancing Technologies*, 2023.
- [KKPG24] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Graphiti: Secure graph computation made more scalable. pages 4017–4031, 2024.
- [LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. pages 377–396, 2013.
- [MRZ15] Payman Mohassel, Mike Rosulek, and Ye Zhang. Fast and secure three-party computation: The garbled circuit approach. pages 591–602, 2015.

A Open science

In compliance with the open science policy, we have made the source code of our work publicly available at <https://github.com/anon1208/malicious-shuffle>.

B Deferred Proofs

Here we present the proofs of security for the shuffle and inner-product verification protocols.

Theorem B.1. *Protocol 4.4 securely computes Functionality 4.1 with abort in the presence of one malicious party.*

Proof. Let $\mathcal{A}$ be a real-world adversary corrupting at most one party, and let P_{cor} denote the corrupted party. We construct an ideal-world simulator $\mathcal{S}$ such that the view of $\mathcal{A}$ in the real execution of Π_{SHuffle} is identically distributed to its view in the ideal execution with $\mathcal{F}_{\text{SHuffle}}$. The simulator works as follows:

Simulating $\Pi_{\text{SH}}\text{--SHuffle}$: $\mathcal{S}$ receives the corrupted party’s input shares of T from $\mathcal{F}_{\text{SHuffle}}$. The simulator emulates all invocations of $\mathcal{F}_{\text{Rand}}$ and $\mathcal{F}_{\text{Coin}}$. $\mathcal{S}$ simulates the execution of $\Pi_{\text{SH}}\text{--SHuffle}$ (Protocol 4.2) as follows. Note that $\mathcal{A}$ ’s behavior is deterministically defined by its initial shares, the messages it receives from other parties and the invocations of $\mathcal{F}_{\text{Rand}}$ and $\mathcal{F}_{\text{Coin}}$. Therefore $\mathcal{S}$

can verify whether $\mathcal{A}$ sends an incorrect message at any point, in which case it instructs $\mathcal{F}_{\text{Shuffle}}$ to abort the protocol. While the simulator always aborts when $\mathcal{A}$ sends an incorrect message, the protocol might fail to abort if $\mathcal{A}$ sends incorrect message due to a badly chosen random projection, but this only occurs with probability at most $\frac{1}{|\mathbb{F}|}$, as was shown in section 4.2.

We distinguish the simulations of messages sent to the corrupt party based on its identity.

- *Case 1: $P_{\text{cor}} = P_1$.* Party P_1 receives the two protocol messages X_2 and Z_2 . The simulator samples both messages uniformly at random from $\mathbb{F}^n$ and sends them to $\mathcal{A}$. This is identically distributed to the real execution, where $X_2 = \pi_2(\pi_1(T_1 + T_2 + R_1) + R_2)$ is perfectly masked from the point of view of P_1 , since R_2 and π_2 are unknown to P_1 and are sampled uniformly at random. Similarly, $Z_2 = Y_3 - T_{O2}$ is perfectly masked by the fresh random list T_{O2} , which is unknown to P_1 . Hence, both incoming messages are uniformly random lists. The simulator receives from $\mathcal{A}$ the messages Y_1 and Z_1 on behalf of the honest parties. $\mathcal{S}$ computes $T_{O3} = Z_1 + Z_2$ and sends T_{O1}, T_{O3} to $\mathcal{F}_{\text{Shuffle}}$.

- *Case 2: $P_{\text{cor}} = P_2$.* Party P_2 sends the message X_2 but receives no protocol message during the semi-honest shuffle phase. Hence, there is nothing to simulate. $\mathcal{S}$ receives the message X_2 sent by $\mathcal{A}$ on behalf of the honest parties. $\mathcal{S}$ sends T_{O2}, T_{O1} generated from $\mathcal{F}_{\text{Rand}}$ to $\mathcal{F}_{\text{Shuffle}}$.

- *Case 3: $P_{\text{cor}} = P_3$.* Party P_3 receives the two protocol messages Y_1 and Z_1 . The simulator samples both messages uniformly at random from $\mathbb{F}^n$ and sends them to $\mathcal{A}$. Again, this is identically distributed to the real execution, where, $Y_1 = \pi_1(T_3 - R_1)$ is masked from the point of view of P_3 , since R_1 and π_1 are unknown to P_3 . Similarly, $Z_1 = X_3 - T_{O1}$ is perfectly masked by the random list T_{O1} , which is unknown to P_3 . Thus, both incoming messages are uniformly random lists. The simulator receives from $\mathcal{A}$ the message Z_2 on behalf of the P_1 . $\mathcal{S}$ computes $T_{O3} = Z_1 + Z_2$ and sends T_{O2}, T_{O3} to $\mathcal{F}_{\text{Shuffle}}$.

Simulating verification: For each verification instance in which the prover is honest, the simulator simply emulates $\mathcal{F}_{\text{VerifyIP}}$ on behalf of the honest parties. For each verification instance in which the prover is corrupted, the simulator computes the honest verifiers' inputs to the corresponding inner-product check from: (i) the corrupted prover's message received during the semi-honest phase, (ii) the honest parties' random values obtained from $\mathcal{F}_{\text{Rand}}$, and $\mathcal{F}_{\text{Coin}}$. It then emulates $\mathcal{F}_{\text{VerifyIP}}$ on behalf of the honest parties. By construction of the four reductions to inner-product verification, each instance of $\mathcal{F}_{\text{VerifyIP}}$ accepts if and only if the corresponding shuffle message was computed correctly. Hence, if the corrupted party deviates in any of the shuffle messages, the corresponding verification call aborts. If all relations are satisfied, then $\mathcal{F}_{\text{VerifyIP}}$ accepts unless the adversary chooses to abort. In case $\mathcal{F}_{\text{VerifyIP}}$ outputs abort, $\mathcal{S}$ sends **Abort** to $\mathcal{F}_{\text{Shuffle}}$ to simulate the honest parties aborting in the real-world execution, outputs whatever P_{cor} outputs, and halts. Else, it sends **Accept** to $\mathcal{F}_{\text{Shuffle}}$ and returns the output of $\mathcal{F}_{\text{Shuffle}}$ to $\mathcal{A}$. □

Theorem B.2. *Assume $|\mathbb{F}| > m + 1$. Then Protocol 5.2 securely realises $\mathcal{F}_{\text{VerifyIP}}$ with abort in the $(\mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Coin}})$ -hybrid model against one malicious corruption. Moreover, if the prover is corrupted, the soundness error is at most $\frac{2m}{|\mathbb{F}| - m - 1}$.*

Proof. Let $\mathcal{A}$ be a real-world adversary corrupting at most one party, and let P_{cor} denote the corrupted party. We construct a simulator $\mathcal{S}$ in the ideal world and show that the view of $\mathcal{A}$ in the real and ideal executions is indistinguishable. We consider separately the cases where the corrupt party is the prover, the deciding verifier, or the non-deciding verifier.

Case 1: the corrupt party is the prover P_i . In this case, $\mathcal{S}$ receives the inputs of P_{cor} from $\mathcal{F}_{\text{VerifyIP}}$. So $\mathcal{S}$ knows the inputs to the honest verifiers and can therefore emulate them exactly. Specifically, for each $j \in [l]$, the simulator emulates $\mathcal{F}_{\text{Rand}}$ on behalf of honest parties to obtain

the jointly sampled values $w_j = f_j(0)$ and $w'_j = g_j(0)$. It then interpolates the unique polynomials f_j, g_j satisfying $f_j(0) = w_j, f_j(k) = a_{(k-1)l+j}$ and $g_j(0) = w'_j, g_j(k) = b_{(k-1)l+j} \forall k \in [m]$. It then receives the proof sent by P_{cor} on behalf of the honest verifiers and simulates the verifier's protocol. If the checks pass, $\mathcal{S}$ sends **Accept** to $\mathcal{F}_{\text{VerifyIP}}$. $\mathcal{S}$ sends **Accept** to the $\mathcal{A}$ on behalf of the honest verifiers. If the verification check fails (indicating that the prover cheated during execution), then $\mathcal{S}$ sends **Abort** to $\mathcal{F}_{\text{VerifyIP}}$ and sends **Abort** to $\mathcal{A}$ on behalf of the honest parties. However, if the honest verifiers simulated by $\mathcal{S}$ output **Accept**, while $\mathcal{F}_{\text{VerifyIP}}$ outputs **Abort**, then the simulation fails and $\mathcal{S}$ outputs **Fail** and halts.

Case 2: the corrupted party is the non-deciding verifier P_{i-1} . In this case, $\mathcal{S}$ receives the inputs of A, c_{i-1} from $\mathcal{F}_{\text{VerifyIP}}$. For each $j \in [l]$, the simulator emulates $\mathcal{F}_{\text{Rand}}$ to generate jointly sampled values $w_j = f_j(0)$ and interpolates the polynomial f_j from the known values $f_j(0) = w_j$ and $f_j(k) = a_{(k-1)l+j} \forall k \in [m]$. It also emulates $\mathcal{F}_{\text{Rand}}$ to generate the corrupt party's share of proof $(h_0^{i-1}, \dots, h_{2m}^{i-1})$ and the evaluation point $r \in \mathbb{F} \setminus \{0, 1, \dots, m\}$. It receives the message from the corrupt party on behalf of P_{i+1} . Since $\mathcal{S}$ knows the input of corrupt party, it can compute the correct message that P_{i-1} should send to P_{i+1} , so it compares the received message with the correct message to decide whether to **Accept** or **Abort**. $\mathcal{S}$ accordingly sends **Accept** or **Abort** to $\mathcal{F}_{\text{VerifyIP}}$ and returns the output to the corrupt party.

Case 3: the corrupted party is the deciding verifier P_{i+1} . In this case, $\mathcal{S}$ receives the inputs of B, c_{i+1} from $\mathcal{F}_{\text{VerifyIP}}$. In this case, for each $j \in [l]$, the simulator emulates $\mathcal{F}_{\text{Rand}}$ to generate jointly sampled values $w'_j = g_j(0)$ and interpolates the polynomial g_j from $g_j(0) = w'_j$ and $g_j(k) = b_{(k-1)l+j} \forall k \in [m]$. It also samples uniformly random field elements $\tilde{h}_0^{i+1}, \dots, \tilde{h}_{2m}^{i+1} \xleftarrow{\$} \mathbb{F}$. Since, in the real execution $(h_0^{i-1}, \dots, h_{2m}^{i-1})$ is chosen uniformly at random and the prover sends the complementary share $(h_0^{i+1}, \dots, h_{2m}^{i+1})$. The message received by P_{i+1} is uniformly distributed over $\mathbb{F}^{2m+1}$. Thus, $\mathcal{S}$ can sample this vector uniformly at random. The simulator then emulates $\mathcal{F}_{\text{Rand}}$ to obtain $r \xleftarrow{\$} \mathbb{F} \setminus \{0, 1, \dots, m\}$ and computes

$$g_j(r) \forall j \in [l], \quad x_{i+1} = h^{i+1}(r), \quad c'_{i+1} = c_{i+1} - \sum_{k=1}^m h^{i+1}(k),$$

and then simulates the message on behalf of the honest verifier P_{i-1} as $\{f_j(r)\}_{j \in [l]}, x_{i-1}, c'_{i-1}$. Because $f_j(0) = w_j$ is uniformly random and jointly sampled between the prover and P_{i-1} , each $f_j(r)$ is uniformly distributed over $\mathbb{F}$. Similarly, the coefficient shares $(h_0^{i-1}, \dots, h_{2m}^{i-1})$ are uniform random shares, and therefore $x_{i-1} = h^{i-1}(r)$ and $c'_{i-1} = c_{i-1} - \sum_{k=1}^m h^{i-1}(k)$ are uniformly distributed subject only to the consistency conditions $x_{i-1} + x_{i+1} = h(r)$ and $c'_{i-1} + c'_{i+1} = c - \sum_{k=1}^m h(k)$. Thus, the simulator can generate the honest verifier's message with the correct distribution. Finally, the simulator receives the corrupt party's message to **Accept** or **Abort** the protocol on behalf of the honest parties and forwards it to $\mathcal{F}_{\text{VerifyIP}}$.

From the above three cases, observe that the simulator fails to perfectly simulate the messages of the corrupt party only when a corrupt prover convinces the deciding verifiers to accept, even though $A \cdot B \neq c$. We show that this happens with probability at most $\frac{2m}{|\mathbb{F}| - m - 1}$. If the second check passes, then necessarily $c'_{i-1} + c'_{i+1} = 0$, which implies $\sum_{k=1}^m h(k) = c$. Hence, acceptance of a false statement can only occur if the polynomial sent by the prover satisfies $\sum_{k=1}^m h(k) = c$ but is not equal to the correctly formed polynomial $\sum_{j=1}^l f_j(x)g_j(x)$. In that case, $q(x) = h(x) - \sum_{j=1}^l f_j(x)g_j(x)$ is a non-zero polynomial of degree at most $2m$. Since r is chosen uniformly from $\mathbb{F} \setminus \{0, 1, \dots, m\}$, the probability that the first check still passes is at most the probability that r is a root of q . Since, $\Pr[q(r) = 0] \leq \frac{2m}{|\mathbb{F}| - m - 1}$ by the Schwartz-Zippel lemma, the soundness error is at most $\frac{2m}{|\mathbb{F}| - m - 1}$.

Combining the three cases, we conclude that Protocol 5.2 securely realises $\mathcal{F}_{\text{VerifyIP}}$ with abort in the $(\mathcal{F}_{\text{Rand}}, \mathcal{F}_{\text{Coin}})$ -hybrid model against one malicious corruption. □

Achieving Full Security. Our protocol can be upgraded to achieve full security using the standard approach of ensuring guaranteed output delivery (GOD) via identification of a trusted third party (TTP) in case of abort. This requires the availability of a broadcast channel. Concretely, if the deciding verifier outputs **Abort** during any invocation of Π_{VerifyIP} , the parties enter a complaint phase. Both verifiers then broadcast all messages received during the protocol execution. Using this public transcript, each party locally recomputes the verification checks and identifies any inconsistency, thereby determining a conflicting pair of parties, at least one of whom must be corrupt. Since at most one party is corrupted, the remaining party is guaranteed to be honest and is designated as the TTP. Once the TTP is identified, all parties reconstruct their inputs to the TTP, who performs the shuffle in the clear and redistributes $\langle \cdot \rangle$ -shares of the output to all parties, ensuring correct output delivery. The cost of identifying the TTP is 2 rounds and $O(\sqrt{n})$ field elements.