

FLOSS: Fast Linear Online Secret-Shared Shuffling

Ian Chang
University of Washington

Sela Navot
University of Washington

Alex Ozdemir
Max Planck Institute for Security and Privacy, Georgia Tech

Nirvan Tyagi
University of Washington

Abstract. Randomly permuting secret data vectors is a core building block in many privacy-preserving protocols, including those for analytics, advertising, and communication. Existing approaches either rely on computation-heavy public key cryptography and zero-knowledge proofs or scale poorly for large vectors due to use of a quasilinear-sized permutation network. This work presents a preprocessing approach to enable fast linear-time online shuffles in the malicious-secure two-party computation (2PC) setting. We propose FLOSS, a 2PC protocol for securely computing any *interactive arithmetic permutation circuit*, a notion we introduce to capture how higher level protocols are built on secret-shared field arithmetic and permutations. We show how secret-shared sorting (a subprotocol in data analytics) can be described as an arithmetic permutation circuit, and can thus be compiled to an efficient online 2PC protocol using FLOSS. Our implementation and evaluation confirm FLOSS performs online shuffles fast: shuffling 2^{20} elements in under 500ms, greater than $800\times$ faster than state-of-the-art alternatives.

Contents

1 Introduction **3**

2 Preliminaries **4**

3 Technical Overview **5**
- 3.1 Online/Offline Architecture . . . 5
- 3.2 Fast Online Shuffling via Preprocessing . . . 5
- 3.3 Authenticated Shuffle Tuple Preprocessing . . . 7
- 3.4 Building Applications on Shuffling . . . 8

4 Arithmetic Permutation Circuits **8**

5 Malicious-Secure 2PC for Arithmetic Permutation Circuits using Shuffle Tuples **9**
- 5.1 Offline Construction of Shuffle Tuples . . . 9
- 5.2 Online Application of Shuffle Tuples . . . 10

6 Evaluation **10**
- 6.1 Implementation . . . 11
- 6.2 Online Shuffling Costs . . . 12
- 6.3 Offline Shuffling Costs . . . 12
- 6.4 Sorting Costs . . . 13

7 Concurrent Work **14**

A Universal Composability Security Model **19**

B Building Block Protocols **20**
- B.1 Commitment Functionality and Protocol . . . 20
- B.2 SPDZ Protocol for Arithmetic Circuits . . . 20

C Security Proof for Online Protocol **21**

D Full Description of Preprocessing Protocol for Authenticated Shuffle Tuples **31**
- D.1 Switching and Permutation Networks . . . 34
- D.2 Malicious-Secure Permutation Network Construction: Forward Shuffle Tuples . . . 35
- D.3 Constructing Inverse Shuffle Tuples . . . 37
- D.4 Constructing κ-Authenticated Shuffle Tuples . . . 38
- D.5 Security . . . 38

E Application: Sorting **39**

F Alternate Circuit for Arbitrary Permutation Application **39**

1 Introduction

Randomly permuting private data is a core building block in constructing privacy-preserving protocols for analytics [NWI+15, MRR20, AHI+22, GKLX22, ACD+24, BBM+25], advertising [CJK+23], communication [AKTZ17, LYK+19, EB22], whistleblowing [KLDF16, Lin17], random-access memory [WHC+14, FJKW15, FOSZ23], contact discovery [KS08, MS13, HKRS20, GHJ+25], analysis of social graphs [AFO+21, KKPG24, BKKS25], and more. A common approach taken for permuting private data is to use multi-party computation (MPC) of a *secret-shared shuffle*. A data vector $\vec{x}$ is secret-shared across parties so each party holds only a share of the data $[\![\vec{x}]\!]$. Following a secret-shared shuffle, parties hold fresh shares $[\![\pi(\vec{x})]\!]$ for a random permutation π of the data vector.

Classical secret-shared shuffling techniques are based on *permutation networks*. That is, a permutation network, i.e., a circuit of switching gates [Wak68], is represented as an arithmetic circuit. Parties use an appropriate MPC protocol to evaluate the arithmetic circuit over the data shares [BGW88]. Since a permutation network for n elements has size $\Theta(n \log n)$, this approach has the disadvantage of incurring $\Theta(n \log n)$ MPC multiplication operations when shuffling n elements.

Recently, Chase et al. [CGP20] reduced the online cost of shuffling (in the honest-but-curious setting) by introducing the idea of a preprocessed *shuffle tuple* that encodes the permutation π. Parties consume a shuffle tuple to perform a secret-shared shuffle with efficient linear computation and communication. Creating the shuffle tuple still requires computing a secret-shared shuffle, e.g., via a permutation network, but this work can be pushed to an offline preprocessing phase. Intuitively, shuffle tuples are analogous to *multiplication triples* [Bea91], a popular preprocessing approach enabling fast online computation of multiplication in MPC. These strategies for pushing work into offline preprocessing phases and enabling fast online computation provide necessary flexibility for higher-level applications to meet latency and efficiency goals in practice.

In this work, we lift the shuffle tuple approach of Chase et al. [CGP20] from the honest-but-curious setting to the two-party (2PC) dishonest-majority malicious setting. The 2PC setting is the target of ongoing standards processes for privacy-preserving analytics [GPP+25] and advertising [PLS+25]. In this standards setting, a service provider wishing to compute telemetry or advertising attribution statistics over users pairs with a second infrastructure provider to jointly compute the desired statistics without either party learning sensitive user data; for example, Divvi Up [div] is a non-profit organization that acts as a second infrastructure provider to services such as Mozilla Firefox. While many basic statistics can be computed via relatively simple aggregation MPC protocols [CB17], there is growing interest in supporting more complex statistics like heavy hitters [AHI+22] and attribution [CJK+23]. Our work is directly applicable to enabling such statistics (that rely on shuffling for efficiency) in these standards.

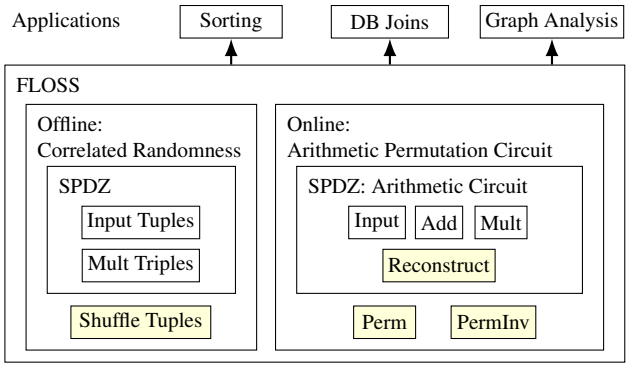


Figure 1: Overview of the FLOSS architecture for malicious-secure 2PC of arithmetic permutation circuits. Applications are built on top of the arithmetic permutation circuit abstraction. FLOSS uses an online/offline architecture in which correlated randomness is generated in an input-independent offline phase and consumed to compute gates in the online phase. FLOSS builds on the SPDZ [DPSZ12] protocol that provides functionality for arithmetic circuits, extending with new gates for permuting vectors that consume a new correlated randomness called a shuffle tuple. Reconstructing values also performs additional checks deferred during permutation. The new protocol contributions in this work are highlighted in yellow.

Prior work has attempted lifting shuffle tuples to the malicious setting, but achieving proper security has proved to be challenging. Initial proposed approaches to construct "authenticated" shuffle tuples [EB22, Lau21] are vulnerable to a selective failure attack in which a malicious party can learn information about the secret permutation, as noted by Song et al. [SYB+24]. Song et al. [SYB+24] propose a new approach using repeated shuffling and cut-and-choose techniques to defend against this attack. We construct a new online shuffling technique using shuffle tuples that avoids the use of expensive repeated shuffling and cut-and-choose entirely. In this way, FLOSS achieves the fastest online time for malicious 2PC secret-shared shuffling, shuffling a vector of 2^{20} 128-bit field elements in 0.5 seconds, improving $3600\times$ over Song et al. [SYB+24]. Offline preprocessing costs also compare favorably to prior work, scaling well and incurring less preprocessing overhead at large vector sizes. Table 1 summarizes the online and offline costs in comparison to other shuffling strategies.

Equipped with our new shuffling protocol, we turn to higher-level applications that use shuffling. Prior applications are often built on top of an *arithmetic circuit* abstraction for MPC that allows parties to securely compute addition and multiplication over secret data. We propose a new abstraction for MPC that we call an *arithmetic permutation circuit* that extends the standard arithmetic circuit abstraction to allow parties to additionally securely compute permutations over

Protocol	Offline costs		Online costs	
MPC-Enc+Shuf [MSS14]	$O(\lambda n)\,\mathbb{E}$	–	$O(\lambda n)\,\mathbb{E}$	–
Linear-Only LHE [DNO25]	$O(n)$	–	$O(\lambda n)\,\mathbb{L}$	–
SimpleNet [Wak68]	$O(n \log n)$	532s	$O(n \log n)$	491s
PermNet [MSS14]	$O(n \log n)$	2436s	$O(n \log n)$	407s
OPM [SYB+24]	$O(nT \log \frac{n}{T})$	6762s	$O(n \log \frac{n}{T})$	1835s
FLOSS (this work)	$O(n \log n)$	1270s	$O(n)$	0.5s

Table 1: Summary of costs for malicious-secure 2PC shuffling protocols. Asymptotic costs for computation are provided and, for practically instantiable constructions, a concrete running time for shuffling a vector of length 2^{20} is included. Symbols n, λ, and T denote the vector size, security parameter, and the size of the oblivious punctured matrix, respectively. Symbols $\mathbb{E}$ and $\mathbb{L}$ denote elliptic curve operation and linearly-homomorphic encryption addition, respectively. Unspecified costs correspond to arithmetic over λ-bit fields. Specified costs are for shuffling 128-bit field elements on two instances with 25 Gbps bandwidth and single-digit-millisecond latency; OPM uses $T = 2^4$.

vectors of secret data. We believe this extended abstraction is sufficient to describe most (if not all) existing applications that use secret-shared shuffling. We prove that our protocol FLOSS securely implements this abstraction in the malicious 2PC setting.

To demonstrate the power of the arithmetic permutation circuit abstraction, we consider *secret-shared sorting* as a case study, a protocol built on shuffling [HKI+12, FO21, AHI+22] that is itself a building block for many analytics applications [AHI+22, CJK+23, ACD+24, BBM+25]. We adapt the popular radix-sort-and-shuffle protocol [BTW12, BLT14, CHI+19, AHI+22] to be written as an arithmetic permutation circuit. All together, the benefits of the arithmetic permutation circuit abstraction include (1) simpler implementation (≈ 140 LoC for secret-shared sorting), (2) simpler proofs of security that deal only with the high-level functionality, and (3) automatic compilation to a fast online protocol via FLOSS (2^{13} 32-bit inputs sorted in 3 seconds).

Contributions. In summary, our contributions include:

- A new arithmetic permutation circuit abstraction and formalism on which to build higher-level MPC protocols (Section 4).
- A protocol FLOSS to instantiate the arithmetic permutation circuit abstraction in the malicious 2PC setting with efficient online computation (Section 5). Our online protocol makes use of correlated randomness in the form of authenticated shuffle tuples (Section 5.2). We provide an offline protocol for constructing this correlated randomness (Section 5.1).
- An implementation and evaluation of FLOSS, including a case-study of applying FLOSS in building an efficient online sorting protocol (Section 6).

2 Preliminaries

Notation. Let λ denote the security parameter and $\mathbb{F}$ be a finite field with $|\mathbb{F}| \geq 2^\lambda$. We use $\leftarrow$ for assignment and $\leftarrow\!\!\$$ for randomized assignment. In particular, if S is a finite set, $x \leftarrow\!\!\$\, S$ indicates sampling a random element from S and storing it in x. If Eval is a randomized algorithm $x \leftarrow\!\!\$\, \mathsf{Eval}(y)$ indicates storing the output of Eval, when run with input y and uniformly random coins, in x. Similarly, $\Pi.\mathsf{P}_0 \leftrightarrow \Pi.\mathsf{P}_1$ denotes an interactive protocol Π between two parties, P_0 and P_1. We use $\perp$ to indicate an invalid computation along with pseudocode keyword "Require" to return $\perp$ if a property does not hold.

We use $[x_i]_{i=1}^m$ to denote a vector of elements $[x_1, \ldots, x_m]$, $[n]$ to denote $[1, \ldots, n]$, and $[\cdot]$ to denote an empty vector. We use $\vec{1}_n \in \mathbb{F}^n$ to denote the all-ones vector $(1, \ldots, 1)$ of length n, and $\vec{0}_n \in \mathbb{F}^n$ to denote the all-zeros vector $(0, \ldots, 0)$ of length n. Define $\mathbb{S}_n$ to be the set of all permutation functions. We abuse notation to interpret $\pi \in \mathbb{S}_n$ both as a function that can be applied to a vector $\vec{x} \in \mathbb{F}^n$ to permute its elements $\pi(\vec{x})$ and as a function that can be applied to a specific index position in $[n]$ that outputs the index of the permuted vector that position is mapped to. Lastly, we encode π as a vector of permuted indices $\vec{\pi} = \pi([n])$.

For secret sharing, we use $[\![x]\!]_b$ to denote additive secret shares of x held by P_b, and $\langle x \rangle_b^{(\gamma)} = ([\![x]\!]_b, [\![\gamma x]\!]_b)$ to denote the pair of additive data share and γ-authenticated share of x; see Appendix B.2 for more details on the secret-sharing scheme employed by SPDZ [DPSZ12]. When it is clear from context, we omit the superscript (γ) and party subscript b. We denote operations over secret shares such as $\langle x \rangle \cdot \langle y \rangle$ to implicitly indicate interactive evaluation of a secret-shared multiplication.

Security model. We prove the security of our 2PC protocols in the Universal Composability (UC) framework [Can01]. At a high level, security of an MPC protocol means that an adversary cannot learn more about the honest party's input than the output of the function being evaluated, and an adversary cannot skew the output beyond choosing their input.

We consider a strong threat model in which the adversary controls the circuit being evaluated, potentially adaptively in response to intermediate outputs, and may deviate from the protocol arbitrarily. Specifically, we target the MPC-with-

abort setting [GL05], where correctness and privacy are guaranteed but the adversary may cause the protocol to abort at any point—potentially after learning its own output but before the honest party receives theirs. In the two-party setting (where honest-majority cannot be assumed), stronger guarantees such as guaranteed output delivery or fairness are known to be impossible [Cle86]. We refer the reader to Appendix A for a more detailed description of our security model.

3 Technical Overview

We overview the main technical contributions of our work putting it in context with relevant related work.

3.1 Online/Offline Architecture

The goal of a 2PC secret-shared shuffle is to shuffle additive secret shares of a data vector $\vec{x} \in \mathbb{F}^n$ where party P_0 holds shares $[\![\vec{x}]\!]_0$ and party P_1 holds shares $[\![\vec{x}]\!]_1$ (s.t., $[\![\vec{x}]\!]_0 + [\![\vec{x}]\!]_1 = \vec{x}$) with a random permutation $\pi \in \mathsf{Perm}(n)$ unknown to both parties. We will focus on a simpler primitive, referred to as a *one-sided shuffle*, in which shares are shuffled according to a permutation known by one party (w.l.o.g. P_0). That is, P_0 holds π_0 and $[\![\vec{x}]\!]_0$ and P_1 holds $[\![\vec{x}]\!]_1$. The parties learn fresh random shares $[\![\pi_0(\vec{x})]\!]_0$ and $[\![\pi_0(\vec{x})]\!]_1$, respectively, and P_1 learns nothing about π_0. Two one-sided shuffles composed in sequence realize a full secret-shared shuffle: P_0 first shuffles with random π_0, then P_1 shuffles with random π_1 to output $[\![\pi(\vec{x})]\!]$ for $\pi = \pi_1 \circ \pi_0$.

Higher-level MPC protocols often use an underlying *arithmetic circuit* functionality abstraction (sometimes referred to as an arithmetic black-box [DN03]). This abstraction exposes arithmetic operations such as addition and multiplication through an interactive functionality interface. The functionality can be instantiated by different underlying low-level protocols. For example, SPDZ [DPSZ12] is a popular protocol that instantiates the arithmetic circuit functionality in the malicious 2PC and dishonest-majority MPC settings. In this way, higher-level MPC protocols use the functionality to invoke arithmetic operations in a way that is agnostic to the choice of the underlying protocol. We extend the traditional arithmetic circuit abstraction to additionally support permutation operations, which we call an *arithmetic permutation circuit*. This allows the one-sided shuffle operation described above to be used as a black-box operation in higher-level MPC protocols.

Further, consider again the protocol SPDZ for the arithmetic circuit abstraction [DPSZ12]. SPDZ instantiates this abstraction in the online/offline model in which correlated randomness may be generated in an offline phase and then consumed in the online phase for fast online computation of arithmetic. Our protocol FLOSS extends SPDZ in the same online/offline model to support fast online computation of one-sided shuffles via correlated randomness generated in the offline phase. Figure 1 depicts this architecture: FLOSS reuses much of the SPDZ construction and integrates new techniques for permuting secret shares.

3.2 Fast Online Shuffling via Preprocessing

Shuffle tuples in the honest-but-curious model. Our starting point is the work of Chase et al. [CGP20, PRRS25] who proposed a novel preprocessing technique for generating correlated randomness referred to as *shuffle tuples*. Shuffle tuples encode a permutation π held by party P_0; these shuffle tuples are consumed during online evaluation to permute secret shares efficiently. A shuffle tuple for permutation π is constructed from a random vector $\vec{\mathsf{a}} \in \mathbb{F}^n$ selected by P_1 and an arbitrary permutation π held by P_0. The shuffle tuple consists of fresh shares of $\vec{\widetilde{\mathsf{a}}} = \pi(\vec{\mathsf{a}})$, such that P_0 holds π and $[\![\vec{\widetilde{\mathsf{a}}}]\!]_0$ and P_1 holds $\vec{\mathsf{a}}$ and $[\![\vec{\widetilde{\mathsf{a}}}]\!]_1$.

Constructing the shuffle tuple still requires permuting secret shares of $\vec{\mathsf{a}}$ during the offline phase; we discuss preprocessing costs shortly. However, the online cost of applying the shuffle tuple to permute during protocol execution is low.

To shuffle $[\![\vec{x}]\!]$, P_1 uses $\vec{\mathsf{a}}$ to blind their shares, $\vec{c} \leftarrow [\![\vec{x}]\!]_1 - \vec{\mathsf{a}}$, and sends blinded shares $\vec{c}$ to P_0. Party P_1 simply sets their new shares for $[\![\pi(\vec{x})]\!]_1 \leftarrow [\![\vec{\widetilde{\mathsf{a}}}]\!]_1$. Party P_0 applies permutation π to set their new shares as

$$[\![\pi(\vec{x})]\!]_0 \leftarrow \pi([\![\vec{x}]\!]_0) + \pi(\vec{c}) + [\![\vec{\widetilde{\mathsf{a}}}]\!]_0$$

The resulting shares are indeed additive shares of $\pi(\vec{x})$:

$$\begin{aligned}[\![\pi(\vec{x})]\!]_0 + [\![\pi(\vec{x})]\!]_1 &= \left(\pi([\![\vec{x}]\!]_0) + \pi(\vec{c}) + [\![\vec{\widetilde{\mathsf{a}}}]\!]_0\right) + [\![\vec{\widetilde{\mathsf{a}}}]\!]_1 \\ &= \pi([\![\vec{x}]\!]_0) + \pi([\![\vec{x}]\!]_1 - \vec{\mathsf{a}}) + [\![\pi(\vec{\mathsf{a}})]\!]_0 + [\![\pi(\vec{\mathsf{a}})]\!]_1 \\ &= \pi([\![\vec{x}]\!]_0) + \pi([\![\vec{x}]\!]_1) - \pi(\vec{\mathsf{a}}) + \pi(\vec{\mathsf{a}}) = \pi(\vec{x})\end{aligned}$$

Overall, a one-sided shuffle is done with one round of communication of n field elements and $O(n)$ field additions by each party.

Pitfall in lifting to malicious security model. A number of works have attempted to lift the notion of preprocessed shuffle tuples to the malicious security setting with active adversaries that may deviate from the protocol.

A natural approach is to build on the authenticated secret sharing strategy of the SPDZ malicious-secure protocol for MPC of arithmetic circuits [DPSZ12]. In SPDZ, parties hold secret shares of data elements $x \in \mathbb{F}$, where $\mathbb{F}$ is a large field (exponential in the security parameter). Additionally, parties hold additive shares of an authentication key $\gamma \in \mathbb{F}$ where $\gamma = [\![\gamma]\!]_0 + [\![\gamma]\!]_1$, and data shares $[\![x]\!]$ are paired with *authenticated shares* $[\![\gamma x]\!]$, which we denote together as $\langle x \rangle \leftarrow ([\![x]\!], [\![\gamma x]\!])$.

Consider now a pair of shuffle tuples over the same permutation π held by P_0 but for different random vectors selected by P_1, $\vec{\mathsf{a}}$ and $\vec{b}$. Party P_0 holds $(\pi, [\![\vec{\widetilde{\mathsf{a}}}]\!]_0, [\![\vec{\widetilde{b}}]\!]_0)$ and P_1 holds $(\vec{\mathsf{a}}, \vec{b}, [\![\vec{\widetilde{\mathsf{a}}}]\!]_1, [\![\vec{\widetilde{b}}]\!]_1)$. To shuffle $\langle \vec{x} \rangle = ([\![\vec{x}]\!], [\![\gamma\vec{x}]\!])$, parties undergo the same procedure as above, using $\vec{\mathsf{a}}$ to shuffle the

$\mathsf{P}_0.\mathsf{BlindAuth}([\![\gamma]\!]_0, \langle\alpha\rangle_0, [\langle z_i\rangle_0]_i^n)$
$\leftrightarrow \mathsf{P}_1.\mathsf{BlindAuth}([\![\gamma]\!]_1, \langle\alpha\rangle_1, [\langle z_i\rangle_1]_i^n)$

1. P_0 and P_1 compute linear combination of inputs $([\![z_i]\!], [\![\gamma z_i]\!]) \leftarrow \langle z_i\rangle$ blinded by random $([\![\alpha]\!], [\![\gamma\alpha]\!]) \leftarrow \langle\alpha\rangle$:
   (a) P_0 and P_1 each sample $r_b \leftarrow\!\!\$\ \mathbb{F}$.
   (b) P_0 and P_1 each commit to r_b by invoking $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Comm}$.
   (c) P_0 and P_1 open commitments to receive r_{1-b} from each other via $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Open}$.
   (d) P_0 and P_1 compute $r \leftarrow r_0 + r_1$.
   (e) P_0 and P_1 compute linear combination:
   $$[\![\ell]\!] \leftarrow [\![\alpha]\!] + \sum_{i=1}^{n} r^{i-1} \cdot [\![z_i]\!], \quad [\![\gamma\ell]\!] \leftarrow [\![\gamma\alpha]\!] + \sum_{i=1}^{n} r^{i-1} \cdot [\![\gamma z_i]\!]$$
2. P_0 and P_1 check $[\![\gamma\ell]\!]$ against $[\![\ell]\!]$:
   (a) P_0 and P_1 open ℓ by sending $[\![\ell]\!]$ to each other.
   (b) P_0 and P_1 compute $[\![\epsilon]\!] \leftarrow [\![\gamma\ell]\!] - [\![\gamma]\!] \cdot \ell$.
   (c) P_0 and P_1 each commit to $[\![\epsilon]\!]_b$ by invoking $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Comm}$.
   (d) P_0 and P_1 open commitments to receive $[\![\epsilon]\!]_{1-b}$ from each other via $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Open}$.
   (e) P_0 and P_1 output `abort` if: $[\![\epsilon]\!]_0 + [\![\epsilon]\!]_1 \neq 0$

Figure 2: The blind authentication check subprotocol used in FLOSS for checking a vector $\vec{z}$ of unopened data shares against authentication shares. b denotes the party index. The protocol also takes as input shares of a random blinding factor α.

data shares and retrieve $[\![\pi(\vec{x})]\!]$ and using $\vec{b}$ to shuffle the authenticated shares and retrieve $[\![\pi(\gamma\vec{x})]\!]$ which together form $\langle\pi(\vec{x})\rangle$. Of course, adversaries may deviate from the protocol and the resulting vectors may not actually have this form. One strategy in prior work [EB22, Lau21] is to perform a *blind authentication check* on the claimed permuted shares (e.g., such as the one presented in Figure 2). A blind authentication check checks that the authenticated shares are indeed valid with respect to the data shares and secret authentication key γ without revealing the values underlying the shares.

Permutation guessing selective failure attack. Unfortunately, this blind authentication check on its own does not prevent malicious behavior. Song et al. [SYB+24] describe a selective failure attack that allows a malicious P_1 to guess partial mappings of the secret permutation π used by P_0.[1] We detail the attack, as it provides intuition for our final construction.

Consider party P_1 who guesses index i maps to index j in permutation π. Party P_1 additively perturbs their share at index i: $[\![x_i']\!]_1 \leftarrow [\![x_i]\!]_1 + \delta$. The parties execute the above shuffling procedure using $\vec{a}$ to shuffle data shares $\vec{x}'$ (with perturbed x_i' at index i) and $\vec{b}$ to shuffle authenticated data shares $\gamma\vec{x}$ (unmodified by P_1). Following this, parties receive shares $[\![\pi(\vec{x}')]\!]$ and $[\![\pi(\gamma\vec{x})]\!]$. Prior to the authentication check, P_1 unperturbs index j of the permuted data shares: $[\![\pi(\vec{x}')_j']\!]_1 \leftarrow [\![\pi(\vec{x}')_j]\!]_1 - \delta$. Party P_1 proceeds with the blind authentication check using $[\![\pi(\vec{x}')']\!]$ (perturbed index i pre-shuffle and unperturbed index j post-shuffle) and $[\![\pi(\gamma\vec{x})]\!]$ (unmodified). With these perturbations, authentication succeeds if and only if π maps index i to j (except with a small probability), which allows P_1 to learn about π.

Prior solution: Repeated Shuffles and Cut-and-Choose. Song et al. [SYB+24] propose a new approach to address this selective failure attack via repeated shuffling and cut-and-choose. First, they observe that for a *random* permutation, the chance that the attacker's guess is correct is not too high—it is just $\tilde{O}(n^{-1})$. Even if the attacker's guess is incorrect and the protocol aborts, some information about the other party's permutation has leaked. To reduce the attacker's success probability while also preventing leakage, Song et al. [SYB+24] apply B shuffle tuples for random independently-sampled permutations $(\pi_1, \ldots, \pi_B)$ in sequence performing an authentication check after each permutation. It is unlikely for an attacker to guess correctly for all B permutations, and if they fail early, say on permutation i, they learn information about $(\pi_1, \ldots, \pi_i)$, but this reveals nothing about the composed permutation $\pi \leftarrow \pi_B \circ \cdots \circ \pi_1$.

Second, Song et al. [SYB+24] employ the honest-but-curious oblivious puncturable matrix (OPM) preprocessing algorithm of Chase et al. [CGP20] to produce shuffle tuples efficiently. However, the OPM preprocessing approach is not malicious-secure: there is some failure probability that an attacker can learn information about the underlying encoded permutation. To mitigate leakage from "leaky" shuffle tuples, they introduce a cut-and-choose selection process in which a large number ($\gg B$) of "leaky" shuffle tuples are preprocessed, and parties randomly select B tuples to apply to perform a shuffle via repeated shuffling (referred to as OPM in Table 1). A side effect of this technique is that the composed permutation $\pi \leftarrow \pi_B \circ \cdots \circ \pi_1$ cannot be chosen by a party. It is necessarily random which will limit flexibility down the line.

Looking forward, we take a different approach from Song et al. [SYB+24]. We prevent the permutation-guessing attack via a new explicit authentication mechanism. This avoids the high concrete costs of repeated shuffling and cut-and-choose and enables greater flexibility in permutation selection.

Our approach: Shuffle tuples in the malicious model. A key characteristic of the permutation-guessing selective failure attack is the malicious party P_1's use of additive perturbation. A natural strategy for preventing attacks via additive perturbation is using secret shares for an unknown multiplicative factor. This is the strategy that SPDZ [DPSZ12] itself relies on with authenticated shares $[\![\gamma x]\!]$ for secret multiplicative authentication key $[\![\gamma]\!]$.

Consider the following extension to the shuffle tuple construction from before. Party P_1 again selects a random vector $\vec{a} \leftarrow\!\!\$\ \mathbb{F}^n$. Party P_0 holds an arbitrary permutation π but also samples a random multiplicative authentication key

[1] The general attack strategy had also been previously described by Araki et al. [AFO+21].

$\kappa \leftarrow\$ \mathbb{F}$. The "$\kappa$-authenticated" shuffle tuple consists of fresh shares of $\vec{\tilde{a}} = \pi(\vec{a})$ as before along with fresh shares of $\kappa\vec{\tilde{a}} = \kappa \cdot \pi(\vec{a})$. All together, P_0 holds $(\pi, \kappa, [\![\vec{\tilde{a}}]\!]_0, [\![\kappa\vec{\tilde{a}}]\!]_0)$ and P_1 holds $(\vec{a}, [\![\vec{\tilde{a}}]\!]_1, [\![\kappa\vec{\tilde{a}}]\!]_1)$.

Consider first using this shuffle tuple to shuffle the data share $[\![\vec{x}]\!]$ of $([\![\vec{x}]\!], [\![\gamma\vec{x}]\!]) \leftarrow \langle\vec{x}\rangle$. The authenticated shares $[\![\gamma\vec{x}]\!]$ are handled separately. As before, P_1 sends blinded shares $\vec{c} \leftarrow [\![\vec{x}]\!]_1 - \vec{a}$ to P_0. Party P_1 sets new shares:

$$[\![\pi(\vec{x})]\!]_1 \leftarrow [\![\vec{\tilde{a}}]\!]_1, \qquad [\![\kappa \cdot \pi(\vec{x})]\!]_1 \leftarrow [\![\kappa\vec{\tilde{a}}]\!]_1$$

Party P_0 uses π and κ to compute its new shares:

$$[\![\pi(\vec{x})]\!]_0 \leftarrow \pi([\![\vec{x}]\!]_0) + \pi(\vec{c}) + [\![\vec{\tilde{a}}]\!]_0$$

$$[\![\kappa \cdot \pi(\vec{x})]\!]_0 \leftarrow \kappa \cdot \pi([\![\vec{x}]\!]_0) + \kappa \cdot \pi(\vec{c}) + [\![\kappa\vec{\tilde{a}}]\!]_0$$

We can verify that P_0's use of κ indeed results in valid shares of $\kappa \cdot \pi(\vec{x})$:

$$\begin{aligned}
&[\![\kappa \cdot \pi(\vec{x})]\!]_0 + [\![\kappa \cdot \pi(\vec{x})]\!]_1 \\
&\quad = \kappa \cdot \pi([\![\vec{x}]\!]_0) + \kappa \cdot \pi(\vec{c}) + [\![\kappa\vec{\tilde{a}}]\!]_0 + [\![\kappa\vec{\tilde{a}}]\!]_1 \\
&\quad = \kappa \cdot \pi([\![\vec{x}]\!]_0) + \kappa \cdot \pi([\![\vec{x}]\!]_1) - \kappa \cdot \pi(\vec{a}) + \pi(\kappa \cdot \vec{a}) = \kappa \cdot \pi(\vec{x})
\end{aligned}$$

In this manner, P_0 and P_1 obtain shares $[\![\kappa \cdot \pi(\vec{x})]\!]$. If P_0 and P_1 have another κ-authenticated shuffle tuple for $\vec{b}$, they perform the same process on the authenticated shares $[\![\gamma\vec{x}]\!]$ to obtain shares $[\![\gamma \cdot \pi(\vec{x})]\!]$ and $[\![\gamma\kappa \cdot \pi(\vec{x})]\!]$.

The key insight is now P_0 and P_1 verify correct application of the permutation by performing the blind authentication check of γ on the κ-shares $\langle\kappa \cdot \pi(\vec{x})\rangle \leftarrow ([\![\kappa \cdot \pi(\vec{x})]\!], [\![\gamma\kappa \cdot \pi(\vec{x})]\!])$ instead of on $\langle\pi(\vec{x})\rangle$. Let us provide some intuition for why this eliminates the permutation-guessing attack. Recall when P_1 mounts the attack guessing index i maps to index j, P_1 perturbs $[\![x_i']\!]_1 \leftarrow [\![x_i]\!]_1 + \delta$ and later attempts to unperturb $[\![\pi(\vec{x}')'_j]\!]_1 \leftarrow [\![\pi(\vec{x}')_j]\!]_1 - \delta$ prior to the blind authentication check over $\langle\pi(\vec{x})\rangle$. Now since the blind authentication check is performed over $\langle\kappa \cdot \pi(\vec{x})\rangle$, to correct the original perturbation of δ, P_1 must unperturb with $\delta\kappa$ for the attack to succeed; P_1 cannot guess $\delta\kappa$ since κ is random and unknown.

Overall, the application of the κ-authenticated shuffle tuple is simple and efficient, avoiding the additional costs of repeated application of shuffle tuples of Song et al. [SYB+24]. The protocol is provided in Figure 3.

Next we discuss strategies for constructing κ-authenticated shuffle tuples during preprocessing.

3.3 Authenticated Shuffle Tuple Preprocessing

κ-authenticated shuffle tuples enable efficient online shuffling. However, this online efficiency comes at the cost of a more expensive preprocessing protocol. Creation of authenticated shuffle tuples can be bootstrapped from any malicious-secure shuffling protocol that operates over SPDZ authenticated shares [DPSZ12] (which includes all that we are aware of). For example, consider the generation of the κ-authenticated shuffle tuple for vector $\vec{a}$ selected by P_1. Party P_1 inputs $\vec{a}$ to create shares $\langle\vec{a}\rangle \leftarrow ([\![\vec{a}]\!], [\![\gamma\vec{a}]\!])$. Parties use an existing malicious-secure shuffling protocol to get shares of $\langle\vec{\tilde{a}}\rangle = \langle\pi(\vec{a})\rangle$ for π held by P_0. To add

$\mathsf{P}_0.\mathsf{ApplyShufTup}\left(\pi, \kappa, \langle\vec{a}\rangle_0^{(\kappa)}, \langle\vec{\tilde{b}}\rangle_0^{(\kappa)}, \langle\vec{v}\rangle\right)$
$\leftrightarrow \mathsf{P}_1.\mathsf{ApplyShufTup}\left(\vec{a}, \vec{b}, \langle\vec{a}\rangle_1^{(\kappa)}, \langle\vec{\tilde{b}}\rangle_1^{(\kappa)}, \langle\vec{v}\rangle\right)$

1. P_0 and P_1 parse $([\![\vec{\tilde{a}}]\!], [\![\kappa\vec{\tilde{a}}]\!]) \leftarrow \langle\vec{\tilde{a}}\rangle^{(\kappa)}$ and $([\![\vec{\tilde{b}}]\!], [\![\kappa\vec{\tilde{b}}]\!]) \leftarrow \langle\vec{\tilde{b}}\rangle^{(\kappa)}$.
2. P_1 sends vectors $\vec{c}$ and $\vec{d}$ blinding shares $([\![\vec{v}]\!]_1, [\![\gamma\vec{v}]\!]_1) \leftarrow \langle\vec{v}\rangle_1$:
   $\vec{c} \leftarrow [\![\vec{v}]\!]_1 - \vec{a}$ $\qquad$ $\vec{d} \leftarrow [\![\gamma\vec{v}]\!]_1 - \vec{b}$;
3. P_1 sets the following:
   - $[\![\vec{\tilde{v}}]\!]_1 \leftarrow [\![\vec{\tilde{a}}]\!]_1$
   - $[\![\gamma\vec{\tilde{v}}]\!]_1 \leftarrow [\![\vec{\tilde{b}}]\!]_1$
   - $[\![\kappa\vec{\tilde{v}}]\!]_1 \leftarrow [\![\kappa\vec{\tilde{a}}]\!]_1$
   - $[\![\gamma\kappa\vec{\tilde{v}}]\!]_1 \leftarrow [\![\kappa\vec{\tilde{b}}]\!]_1$
4. P_0 permutes and computes the following:
   - $[\![\vec{\tilde{v}}]\!]_0 \leftarrow \pi([\![\vec{v}]\!]_0) + \pi(\vec{c}) + [\![\vec{\tilde{a}}]\!]_0$
   - $[\![\gamma\vec{\tilde{v}}]\!]_0 \leftarrow \pi([\![\gamma\vec{v}]\!]_0) + \pi(\vec{d}) + [\![\vec{\tilde{b}}]\!]_0$
   - $[\![\kappa\vec{\tilde{v}}]\!]_0 \leftarrow \kappa \cdot \pi([\![\vec{v}]\!]_0) + \kappa \cdot \pi(\vec{c}) + [\![\kappa\vec{\tilde{a}}]\!]_0$
   - $[\![\gamma\kappa\vec{\tilde{v}}]\!]_0 \leftarrow \kappa \cdot \pi([\![\gamma\vec{v}]\!]_0) + \kappa \cdot \pi(\vec{d}) + [\![\kappa\vec{\tilde{b}}]\!]_0$
5. P_0 and P_1 defer blind authentication check on $\langle\kappa\vec{\tilde{v}}\rangle$ (see Fig. 2).
6. P_0 and P_1 output $\langle\vec{\tilde{v}}\rangle \leftarrow ([\![\vec{\tilde{v}}]\!], [\![\gamma\vec{\tilde{v}}]\!])$ and $\langle\kappa\vec{\tilde{v}}\rangle \leftarrow ([\![\kappa\vec{\tilde{v}}]\!], [\![\gamma\kappa\vec{\tilde{v}}]\!])$.

Figure 3: Fast online application of preprocessed authenticated shuffle tuple to permute secret shares in FLOSS.

κ authentication, P_0 inputs κ to share $\langle\kappa\rangle$, and parties perform authenticated multiplication, e.g., via multiplication triples, to compute shares $\langle\kappa\vec{\tilde{a}}\rangle$. As a final step, parties verify all computed shares $[\![\vec{\tilde{a}}]\!]$ and $[\![\kappa\vec{\tilde{a}}]\!]$ were computed correctly by performing a blind authentication check on $\langle\vec{\tilde{a}}\rangle$ and $\langle\kappa\vec{\tilde{a}}\rangle$, before discarding the γ-authenticated shares.[2] The protocol is repeated for random vector $\vec{b}$ using the same κ and π.

There are two high level strategies taken in prior work for shuffling (summarized in Table 1): (1) Homomorphic encryption schemes (with proofs of knowledge) and (2) secure computation of a permutation network. Mohassel et al. [MSS14] and Dittmer et al. [DNO25] both propose linear cost protocols that rely on encryption schemes with homomorphic properties. Mohassel et al. [MSS14] use MPC to encode and encrypt secret data into El Gamal elliptic curve ciphertexts [SA19] and then have the permuting party verifiably shuffle the ciphertexts using a proof of knowledge [Nef01, Gro09, Gro10, BG12]. Dittmer et al. [DNO25] propose a malicious-secure protocol using a linearly-homomorphic encryption (LHE) scheme to encrypt and rerandomize shares. Their protocol is secure if the LHE scheme satisfies a "linear-only" property—that is, only linear operations can be performed over ciphertexts—and supports efficient ciphertext validity checks. However, it is unclear how to instantiate an LHE scheme with these properties over fields. Instead, one

[2] The authentication key γ used for the preprocessing is necessary to prevent malicious behavior during preprocessing but it is not part of the produced κ-authenticated shuffle tuple. It may be different from the authentication key used in the downstream FLOSS protocol.

can extend a candidate LHE scheme with zero-knowledge proofs of knowledge to restrict to linear-only operations or to prove ciphertext validity. Ultimately, actively-secure protocols taking the homomorphic encryption approach are concretely expensive and have not yet been implemented.

Alternatively, securely computing a permutation network [Wak68] incurs $O(n \log n)$ costs for shuffling n items due to the number of switches in the permutation network. This approach can be instantiated efficiently in practice. To achieve a constant-round protocol, Mohassel et al. [MS13, MSS14] introduce a permutation-encoding technique that avoids running the permutation network as a $(\log n)$-depth circuit. Song et al. [SYB+24], as discussed above, integrate higher degree OPM switches into the permutation network; then, they must handle imperfect OPM usage with a cut-and-choose amplification step.

Therefore, the two permutation network approaches are the most practical candidates for preprocessing shuffle tuples. However, looking forward to the next section, we explore shuffling in higher level protocols—we find that it will be useful to apply the same permutation repeatedly or apply the correlated inverse permutation of a previously applied permutation. Creating shuffle tuples with these correlated permutations using the circuit-description permutation network approach [MSS14] is simple, as the control bits encoding the permutation network can be reused across multiple runs (and used in reverse). On the other hand, Song et al.'s leaky OPM approach [SYB+24] requires a random cut-and-choose selection which means that the permutation is random and will not correlate with prior permutations. Therefore, we construct our preprocessing protocol for authenticated shuffle tuples from circuit-description permutation networks.

3.4 Building Applications on Shuffling

Our motivation for studying shuffling is the vast set of higher level protocols that use shuffling as a building block. However, shuffling breaks the common arithmetic circuit interface (i.e., secure computation of addition and multiplication) that many MPC protocols (e.g., SPDZ [DPSZ12]) target. Thus, we introduce an *arithmetic permutation circuit* interface that explicitly includes (one-sided) permutations as a supported operation. We formalize interactive arithmetic permutation circuits as an ideal functionality in the universal composability (UC) model (see Figure 13) and prove FLOSS secure with respect to the functionality.

We present *secret-shared sorting* as a case study to demonstrate the utility of our new abstraction. Sorting is a building block used to construct higher level protocols for data analytics including filtering and joins [MRR20, BDG+22, CJK+23] and histogram computation [ACD+24, AHI+22]. We demonstrate how to express the efficient radix-sort-and-shuffle protocol of Asharov et al. [AHI+22] as an arithmetic permutation circuit.

First, consider a subprotocol for interpreting a secret-shared vector as an arbitrary permutation ρ and applying that permutation to secret shares of a data vector $\vec{x}$. This subprotocol is used in the radix sort algorithm and is commonly used in other algorithms as well. The radix sort algorithm produces shares of a permutation that represents the stable sorting for a single bit position. Then to compute the final sorting permutation, the algorithm applies each bit-position's stable sorting permutation in sequence. Observe that this task differs from what our preprocessed shuffle tuples accomplish. First, this permutation is unknown to the parties at the preprocessing stage, and second, the permutation is encoded as secret shares.

Asharov et al. [AHI+22] propose a neat approach to instantiating this subprotocol using random permutations that we observe can be preprocessed (see Figure 25). Parties "blind" the permutation ρ by applying random permutations π_0 and π_1 held by each party, respectively, and then reconstruct the blinded permutation $\pi_1 \circ \pi_0 \circ \rho$. Parties apply the reconstructed permutation locally to their shares of $\vec{x}$. Then, they remove the blinding permutations by sequentially applying inverse permutations π_1^{-1} and π_0^{-1}. Naively, this does not fit our model: parties can apply random preprocessed permutations π_0 and π_1 but it is unclear how to apply the inverse permutations π_1^{-1} and π_0^{-1}. One solution is that we can construct an alternate protocol that avoids the use of inverse permutations, but this approach incurs eight applications of forward permutations (see Appendix F). If we incorporate inverse permutations directly into our preprocessing and the arithmetic permutation circuit abstraction, we reduce the overhead by $2\times$. In particular, the Mohassel et al. [MSS14] permutation network approach (or any circuit-description permutation network) can be easily adapted to compute correlated inverse shuffle tuples[3] by reusing the permutation network control bits and running the permutation network circuit in reverse.

Our arithmetic permutation circuit functionality allows parties to input permutations during the preprocessing stage by specifying the number of forward and inverse shuffle tuples to generate. Inverse permutations enable optimized descriptions of higher level applications in the arithmetization. In Section E, we complete the description of the radix-sort-and-shuffle algorithm as an arithmetic permutation circuit.

4 Arithmetic Permutation Circuits

In this section, we introduce our new abstraction for an *arithmetic permutation circuit* functionality. This functionality extends the common *arithmetic circuit* functionality that underlies many prior works (e.g., the arithmetic black box (ABB) [DN03] model), where the interactive functionality interface exposes black-box access to compute arithmetic "gates". Wire values in the circuit are represented by identifier

[3]The recent ORQ analytics engine [BBM+25] likewise uses correlated inverse shuffle tuples, in the semi-honest setting.

handles, which reference internal field values that are maintained by the functionality and not publicly exposed through the interface. While the values referenced by handles are private, the structure of the circuit (the sequence of gates and their input handles) is public and enforced by the functionality. No party can rewire the circuit or equivocate on which gates to run.

We begin by describing the interface for *arithmetic circuits* (see full description of $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ in Figure 10), which provides black-box access to these primary gates:

- $\mathsf{Input}(\mathsf{id}, v \in \mathbb{F})$: Stores a private input into a new handle.
- $\mathsf{Add}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$: Stores the sum of two handle values into a new handle.
- $\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$: Stores the product of two handle values into a new handle.
- $\mathsf{Reconstruct}(\mathsf{id})$: Reveals the value of a handle.

The *arithmetic permutation circuit* extends the arithmetic circuit interface to include permutation operations (see full description of $\mathsf{F}_{\mathsf{ArithPermCirc}}[\mathbb{F}]$ in Figure 13).

- $\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$: Applies a permutation π to the values referenced by m handles and stores the result in m new handles.
- $\mathsf{PermuteInv}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$: Applies the inverse permutation π^{-1} to the values referenced by m handles and stores the result in m new handles.

Finally, our functionality supports the online/offline model where correlated randomness is generated in an offline preprocessing phase and subsequently consumed when invoking the above gate operations.

- $\mathsf{Prep}(n_{i,0}, n_{i,1}, n_m, \mathsf{in}_\pi)$: Generates a number of input tuples, multiplication triples, shuffle tuples, and inverse shuffle tuples to be consumed by Input, Mul, Permute, and PermuteInv, respectively.

One view of our new arithmetic permutation circuit abstraction is in creating a new low-level interface on which higher-level MPC protocols may be built. An alternative approach would be to use the existing arithmetic circuit abstraction and define a new stand-alone permutation/one-sided shuffle functionality on top of an arithmetic circuit that may be used by higher-level protocols in conjunction with arithmetic circuit gates. While this approach is valid, it limits how the one-sided shuffle construction can be instantiated. In particular, the shuffle must be agnostic to the underlying protocol implementing the arithmetic circuit functionality.

Looking forward, FLOSS is not agnostic to an underlying arithmetic circuit functionality. FLOSS opens up the black-box of the SPDZ protocol for arithmetic circuits [DPSZ12] and directly integrates support for permutation gates in a way that is compatible with SPDZ. Ultimately, our approach enables a concretely more efficient online shuffling protocol.

5 Malicious-Secure 2PC for Arithmetic Permutation Circuits using Shuffle Tuples

In this section, we present FLOSS, our malicious-secure 2PC protocol for the arithmetic permutation circuit functionality described in the previous section. Similar to SPDZ [DPSZ12], our protocol operates in the online/offline model offloading expensive preprocessing computations to an input-independent offline phase. We extend the SPDZ offline phase to support efficient generation of shuffle tuples using the permutation network approach of Mohassel et al. [MSS14] (Section 5.1). These shuffle tuples are consumed during the FLOSS online phase to apply permutations to authenticated shares, enabling fast online shuffling (Section 5.2).

5.1 Offline Construction of Shuffle Tuples

The correlated randomness needed for our protocol (as described in Section 3) to enable fast online shuffling is authenticated shuffle tuples. An authenticated shuffle tuple for permutation π chosen by party P_0 consists of the following values for P_0 (left) and P_1 (right) where P_0 chooses random $\kappa \leftarrow\!\!\$\, \mathbb{F}$ and P_1 chooses random vectors $\vec{a}, \vec{b} \leftarrow\!\!\$\, \mathbb{F}^n$:
$\left(\pi, \kappa, \langle \pi(\vec{a}) \rangle_0^{(\kappa)}, \langle \pi(\vec{b}) \rangle_0^{(\kappa)}\right), \quad \left(\vec{a}, \vec{b}, \langle \pi(\vec{a}) \rangle_1^{(\kappa)}, \langle \pi(\vec{b}) \rangle_1^{(\kappa)}\right).$
We describe a functionality $\mathsf{F}_{\mathsf{ShufTuplePrep}}[\mathbb{F}]$ for producing correlated randomness of this form in Figure 15.

Each party selects permutations for which to generate shuffle tuples. We employ a permutation network approach to generate authenticated shuffle tuples of the above form. W.l.o.g. given π and κ from P_0 and $\vec{a}$ and $\vec{b}$ from P_1, consider a circuit description of a permutation network that takes $\vec{a}$ as input from P_1 and control bits for the network encoding π from P_0. Using the Mohassel et al. [MSS14] construction for malicious-secure permutation networks which itself builds on the SPDZ malicious-secure arithmetic circuit computation protocol [DPSZ12], the parties can securely compute authenticated shares of $\langle \pi(\vec{a}) \rangle^{(\gamma)}$ with respect to some authentication key γ used by the SPDZ protocol.

Further, for each permutation π, parties may also specify the number of shuffle tuples to generate for π and the number of shuffle tuples to generate for the inverse permutation π^{-1}. Parties reapply the same permutation to $\vec{b}$ or to fresh $(\vec{a}', \vec{b}')$ values to generate additional shuffle tuples for π. They simply reuse the inputted authenticated control bits to rerun the permutation network circuit. Similarly, parties will reuse the same control bits and reverse traversal order to run an inverse permutation network circuit. This generates shares of inverse permutations $\langle \pi^{-1}(\vec{a}) \rangle^{(\gamma)}$.

To complete the generation of the authenticated shuffle tuples, P_0 inputs the authentication key κ for the tuple and parties run secret-shared multiplication of the corresponding $(\langle \pi(\vec{a}) \rangle^{(\gamma)}, \langle \pi(\vec{b}) \rangle^{(\gamma)})$ with $\langle \kappa \rangle$ to obtain the authenticated shares $(\langle \kappa\pi(\vec{a}) \rangle^{(\gamma)}, \langle \kappa\pi(\vec{b}) \rangle^{(\gamma)})$. Finally, the parties per-

form a blind authentication check with $[\![\gamma]\!]$ over the produced vectors $(\langle\pi(\vec{a})\rangle^{(\gamma)}, \langle\pi(\vec{b})\rangle^{(\gamma)}, \langle\kappa\pi(\vec{a})\rangle^{(\gamma)}, \langle\kappa\pi(\vec{b})\rangle^{(\gamma)})$. If the check passes, parties discard the γ-authenticated shares and output $\langle\pi(\vec{a})\rangle^{(\kappa)}, \langle\pi(\vec{b})\rangle^{(\kappa)}$.

We defer the full construction description—including the description of a permutation network as a circuit—and security argument of our authenticated shuffle tuple generation protocol to Appendix D. We provide the following corollary from the security proof of the original permutation network protocol of Mohassel et al. [MSS14], on which our construction is based:

Corollary 1 (Informal)**.** *The protocol* $\mathsf{PermNetSTPrep}[\mathbb{F}]$ *(Figure 19) UC-realizes the ideal functionality* $\mathsf{F}_{\mathsf{ShufTuplePrep}}$ *(Figure 15) in the* $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$*-hybrid model.*

Sketch: Aside from the reuse of control bits across multiple runs of the permutation network circuit, our protocol departs from Mohassel et al. only at the end when multiplying by the shuffle tuple key κ and performing the blind authentication check. The multiplication is secure by the security of SPDZ multiplication, and the blind authentication check ensures that any tampering during the permutation network protocol or κ-multiplication is detected with overwhelming probability.

5.2 Online Application of Shuffle Tuples

Our protocol FLOSS extends the SPDZ protocol [DPSZ12] for arithmetic circuits to support permutation gates via preprocessed authenticated shuffle tuples. As such, the two main protocol extensions to SPDZ consist of (1) the application of authenticated shuffle tuples to realize a permutation gate (subroutine ApplyShufTup shown in Figure 3), and (2) the blind authentication check used to detect malicious behavior during shuffling (subroutine BlindAuth shown in Figure 2). As in SPDZ, the authentication checks incurred during a permutation gate application are deferred until a value reconstruction; this minimizes the number of communication rounds as many authentication checks can be batched together.

We formally prove FLOSS secure with respect to a new arithmetic permutation circuit functionality (described in Section 4). We defer the complete protocol description and security proof to Appendix C.Our main result is captured by the following theorem:

Theorem 2 (Informal)**.** *The protocol* $\mathsf{FLOSS}[\mathbb{F}]$ *(Figure 14) UC-realizes the ideal functionality* $\mathsf{F}_{\mathsf{ArithPermCirc}}[\mathbb{F}]$ *(Figure 13) in the* $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{ShufTuplePrep}}, \mathsf{F}_{\mathsf{Comm}})$*-hybrid model.*

Proof sketch. For the standard arithmetic gates, security follows from SPDZ. The proof relies on two key properties: (1) all messages received by the adversary except for the reconstructed outputs are uniformly random (enabling simulation), and (2) any deviation in a value sent to the honest party is caught by the subsequent authentication check with overwhelming probability (ensuring correctness and preventing selective failure attacks).

The main challenge is extending deviation-detection to permutation gates, which is non-trivial when the honest party applies the permutation. Recall from Section 3 that a naive approach, which performs authentication checks directly on $\langle\pi(\vec{x})\rangle$, is vulnerable to a selective failure attack: the adversary can introduce a perturbation before the shuffle and "fix" it afterward by guessing where the perturbation lands under π, passing the authentication check with non-negligible probability.

Our proof shows that by checking validity of the κ-multiplied shares instead, any adversarial deviation is caught unless the adversary guesses κ. Recall that the adversary sends blinded shares $\vec{c}$ and $\vec{d}$ to the honest party (see Figure 3). Suppose the adversary perturbs these by $\vec{\delta}_c$ and $\vec{\delta}_d$, respectively. The blind authentication check computes a random linear combination over the κ-multiplied shares (using a public random challenge r) and verifies that the parties' shares of a verification value ϵ sum to zero. Tracing through shows that the honest party's share takes the form

$$[\![\epsilon]\!]_1 = (\text{terms independent of } \kappa) + \kappa \cdot \sum_{j=1}^{m} r^{j-1} \pi(\vec{\delta}_d - \gamma\vec{\delta}_c)_j.$$

Assuming the adversary introduced some nonzero perturbation and did not guess γ, the vector $\pi(\vec{\delta}_d - \gamma\vec{\delta}_c)$ is nonzero, so the sum is a nonzero polynomial in r. By the Schwartz-Zippel lemma, this univariate polynomial has at most $m-1$ roots, so it evaluates to a nonzero value with overwhelming probability over the random choice of r. Thus $[\![\epsilon]\!]_1$ is a non-constant affine function of the hidden key κ, making it uniformly random from the adversary's view. The adversary must output $-[\![\epsilon]\!]_1$ to pass, which succeeds with negligible probability. We use this intuition to build a full simulation-based proof in Appendix C.

6 Evaluation

We evaluate FLOSS along two axes: *offline* costs and *online* costs. We define offline costs as computations that may be performed before the inputs are known. This consists of generation of preprocessed correlated randomness, including generation of shuffle tuples. For offline costs, we focus on monetary costs as a combination of computation time and bandwidth. Parties have flexibility to run offline computations when convenient so specific wall-clock time measurements are less meaningful. In contrast, online costs are computations performed after the inputs are known, and wall-clock time (latency) is often the most important metric.

We estimate monetary costs using AWS pricing as of February 2026. For offline compute costs, we assume execution on a `c5.18xlarge` spot instance priced at $\$0.93/\mathrm{hr}$, and we multiply this rate by the measured runtime of the offline portion.

For storage costs, we compute the total volume of offline data generated and multiply it by the S3 Standard storage rate of $\$0.023/(\text{GB}\cdot\text{month})$, prorated for the total duration (offline and online) of the protocol.

We wish to answer the following questions about FLOSS in comparison to malicious-secure 2PC baselines:

- *Online shuffling overhead:* How much faster and more communication-efficient is FLOSS's one-sided shuffle compared to the baselines?
- *Offline shuffling overhead:* What is the offline cost required to enable online efficiency?
- *Higher level applications:* Can we use FLOSS as an interface to construct a sorting protocol with favorable monetary, computation, and communication costs to 2PC sorting alternatives?

We find that FLOSS greatly improves online costs without incurring significant offline cost overheads compared to prior approaches. In fact, for large vector sizes, FLOSS achieves both lower online *and* lower offline costs than all other prior approaches targeting fast online time. Our results are summarized in Figure 4 for shuffling and Figure 6 for sorting which plot offline monetary cost against online time; the plots use reverse axes so that better outcomes (cheaper and faster) trend toward the upper right. For shuffling 2^{20} values, FLOSS improves by $800\times$ in online time with $2\times$ less offline cost.

6.1 Implementation

We implement FLOSS in Rust. We provide a base implementation of the SPDZ protocol [DPSZ12] for arithmetic circuits (see Figure 11 and Appendix B.2). Our implementation of arithmetic permutation circuits uses relevant SPDZ gates/gadgets like addition and multiplication and extends to include permutation gates (see Figure 14). We provide an example implementation of a custom sorting gate constructed with addition, multiplication, and permutation gates. The implementation enables further higher-level custom gates to be defined on top of the base gates we provide to construct more complex interactive arithmetic permutation circuits. Our implementation is field-agnostic using the `arkworks` library; our experiments use the 128-bit prime $p = 2^{127} + 55 \cdot 2^{15} + 1$ used by MP-SPDZ [mps].

To generate SPDZ multiplication triples and input tuples, we use the MP-SPDZ library [mps] implementations of MASCOT [KOS16] and LowGear [KPR18]. While LowGear is more efficient, the MP-SPDZ implementation of LowGear does not support generation of small batches of triples ($< 10^6$). Thus, our benchmarks use MASCOT for smaller vector sizes and LowGear for larger vector sizes. We run both protocols with statistical security parameter $\lambda = 40$. For the offline generation of shuffle tuples in FLOSS (see Figure 15 and Appendix D), we instantiate the permutation network approach using the naive $\log(n)$-round circuit evaluation protocol (in contrast, to the online-optimized constant-online-round protocol of Mohassel et al. [MSS14]). We select the Waksman network [Wak68] to use as the underlying permutation network topology.

In total, our implementation of FLOSS consists of ≈15000 lines of code (LoC); it is available open source.[4]

Shuffling baselines. We compare to three baselines for malicious-secure 2PC shuffling. All baselines shuffle 128-bit field elements over the same prime field.

First, we consider the constant online-round circuit-based permutation network construction proposed by Mohassel et al. [MSS14]. We refer to this baseline as *PermNet*. We reimplement this construction on top of our SPDZ interface in ≈1300 LoC.

The second baseline is the malicious-secure adaptation of the Chase et al. [CGP20] honest-but-curious shuffle tuple construction proposed by Song et al. [SYB+24]. We use a prototype implementation provided by the authors.We refer to this baseline as *OPM*.

OPM has a number of parameters that influence its costs. First, the size T of the oblivious punctured matrix used in the offline portion is parameterized. Roughly, larger T incurs longer compute time but lower communication cost during the offline computation. We refer to Table III and IV in [SYB+24] for details; our experiments use $T = 2^4$. Second, the cut-and-choose parameter N together with T determine the number of shuffle tuples B to be applied for each shuffle. Larger N and larger T result in smaller B, but require a larger one-time offline cost in which NB shuffle tuples are created and shared. For example, to shuffle a vector of length $n = 2^{18}$ for $T = 2^4$ and $N = 2^{10}$ times, the one-time offline cost requires computing and sending $2^{10} \cdot B$ shuffle tuples for $B = 108$ shuffle tuples, taking $\approx 2^{10} \cdot 171\text{s} = 2$ days of time and incurring $\approx 2^{10} \cdot 2.0\text{GB} = 2\text{TB}$ in communication costs. We omit this one-time cost in our experiments. The prototype implementation computes the *marginal* offline cost, which is the cost to generate and apply B shuffle tuples excluding the N initial repetitions. We report on the marginal cost setting $N = 2^4$. This estimates what an "amortized" cost for a single shuffle would be if $N = 2^4$ shuffles are performed.

Lastly, we also include comparison against the simple $\log(n)$-round permutation network evaluated as a circuit. This provides a baseline for low offline costs and a straightforward but expensive online protocol. We refer to this as SimpleNet.

Sorting baselines. We evaluate three baseline sorting algorithms: Radix-Sort-and-Shuffle [AHI+22], Shuffle-then-Sort [HKI+12], and a sorting network [JKU11].

Both Radix-Sort-and-Shuffle and Shuffle-then-Sort rely on a shuffling subroutine. We describe the Radix-Sort-and-Shuffle algorithms in Section E as an arithmetic permutation circuit. Similarly, Shuffle-then-Sort can also be described

[4] https://doi.org/10.5281/zenodo.20278557

Sorting algorithm	Shuffle			
	FLOSS	OPM	PermNet	SimpleNet
Radix-Sort-and-Shuffle [AHI+22]	✓		✓	✓
Shuffle-then-Sort [HKI+12]	✓	✓	✓	✓

Table 2: Summary of compatible shuffling protocols for instantiating shuffling-based sorting algorithms.

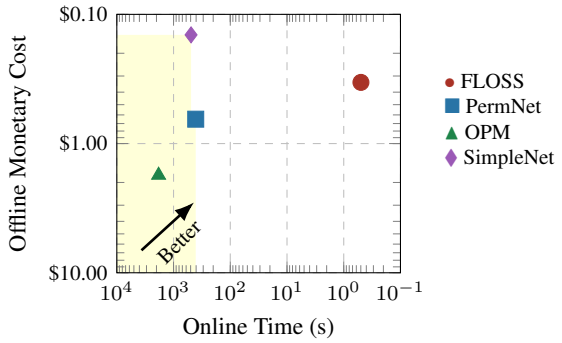


Figure 4: Cost summary of protocols for shuffling 2^{20} 128-bit field entries on two machines (25Gbps, single-digit ms latency); OPM uses $T = 2^4$. FLOSS has $\geq 800\times$ faster online time than prior protocols, and $2\times$ cheaper offline monetary cost than prior protocols outside of SimpleNet. We highlight prior work in yellow.

generically as an arithmetic permutation circuit. Here, inputs are shuffled and then sorted using a quicksort algorithm in which comparison bits are reconstructed to determine sorting. We implement both of these sorting algorithms instantiating the underlying shuffling subroutine using the shuffling protocols: FLOSS, PermNet, SimpleNet, and OPM. Note, we do not instantiate the Radix-Shuffle-and-Sort algorithm using OPM because it is not clear how to do this. OPM only allows applying random permutations, and these permutations can only be applied once. Table 2 summarizes which shuffling protocols are compatible with each sorting algorithm. We refer to these baselines Radix[FLOSS], Radix[PermNet], Radix[SimpleNet], StS[FLOSS], StS[PermNet], StS[SimpleNet], and StS[OPM], respectively.

Lastly, we compare to the canonical sorting network [JKU11] implemented in MP-SPDZ [mps] in which comparison bits are held secret and values are swapped through the sorting network obliviously. We refer to this baseline as SortNet. It does not use a shuffling subroutine.

Experimental setup. We run the two parties on two different Amazon EC2 `c5.18xlarge` instances, each with 72 vCPUs. Our benchmarks are run multithreaded. Between the two instances, bandwidth is 25 Gbps (high-bandwidth) and latency is single-digit milliseconds [Ama26].

6.2 Online Shuffling Costs

We microbenchmark the online costs of performing a single one-sided shuffle in Figure 5. As expected, FLOSS outperforms the baselines in both computation time and communication costs, due to efficient application of authenticated shuffle tuples making up the online costs. For input size 2^{20}, the online protocol completes in < 500 ms and sends 33.6MB, which corresponds approximately to sending two field vectors of size 2^{20}.

Of the 500 ms reported for the online time of FLOSS, the majority of the cost ≈400 ms comes from executing the permutation gate, while the remaining ≈100 ms corresponds to the cost of the deferred blind authentication check. Intuitively FLOSS outperforms the baselines because PermNet incurs additional costs based on the size of the permutation network (as we discuss next) and requires interaction to prepare the inputs and outputs, while OPM incurs higher concrete costs due to repeated application of B shuffle tuples from its cut-and-choose analysis.

As far as scaling, the online computational costs of FLOSS scale linearly in the vector size n, while OPM and PermNet scale with $O(n \log n)$ due to parameters that depend on a logarithmic depth permutation network. PermNet grows linearly in communication costs as it only sends field elements corresponding to n input and output wires of the network. We observe sublinear scaling in FLOSS for small vector lengths from 2^{10} to 2^{18} in compute time. From 2^{18} to 2^{20}, there is a superlinear $5\times$ increase in compute time instead of the expected $4\times$. We hypothesize this superlinear scaling is due to cache thrashing; at 2^{20}, these operations consume 32MB, which exceeds the 24.8MB L3 cache of our `c5.18xlarge` instance [Lab26, Ama26]. We extend our experiments to verify behavior after 2^{20} elements, and we see that computational overhead stabilizes with linear growth. Shuffling 2^{20} elements with FLOSS takes 0.50s, shuffling 2^{21} elements takes 0.98s, and shuffling 2^{22} elements takes 1.99s. We expect that linear growth continues from there.

6.3 Offline Shuffling Costs

As shown in Figure 4, FLOSS achieves the best offline cost efficiency among all baselines targeting fast online time. As shown in Figure 5, for lower input sizes ($\leq 2^{14}$), OPM has a less costly offline computation than FLOSS. However, OPM's offline and total costs exceed FLOSS at larger input sizes ($\geq 2^{16}$) due to increasing cut-and-choose overhead.

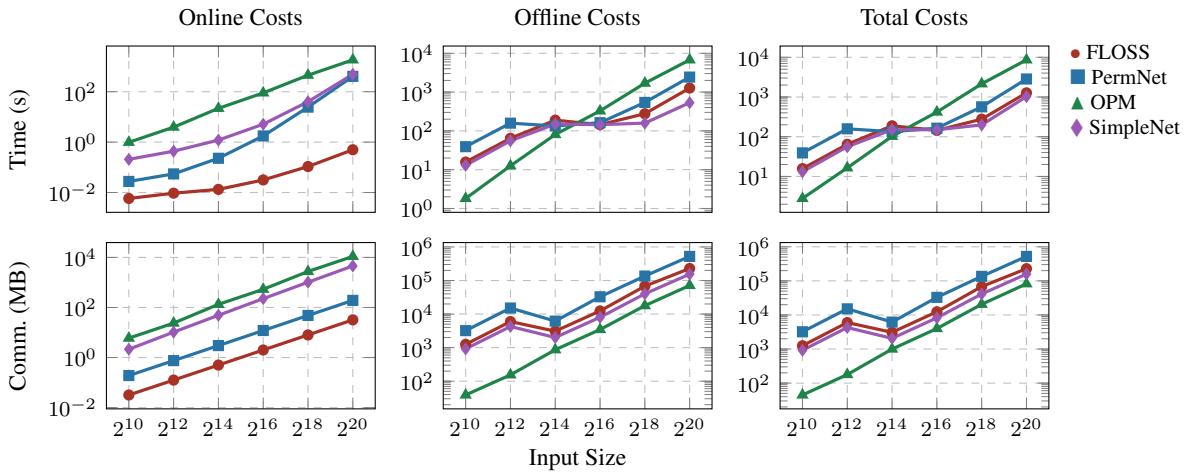


Figure 5: Computation time and communication costs for shuffling 128-bit field entries on two machines (25Gbps, single-digit ms latency) varying from 2^{10} to 2^{20} broken down by online and offline phases; OPM uses $T = 2^4$. FLOSS has lower online costs than all baselines.

	FLOSS offline (s)			PermNet offline (s)		
Size	SPDZ	Circuit	Total	SPDZ	Circuit	Total
2^{10}	15.30	0.43	15.73	38.98	0.20	39.17
2^{12}	63.64	0.82	64.46	157.75	0.53	158.28
2^{14}	186.76	2.01	188.77	131.94	1.88	133.82
2^{16}	137.02	7.27	144.29	152.07	10.45	162.52
2^{18}	230.92	45.95	276.88	461.12	80.48	541.60
2^{20}	755.84	514.54	1270.38	1762.53	673.53	2436.06

Table 3: Breakdown of offline cost for FLOSS and PermNet between cost to generate SPDZ correlated randomness preprocessing and permutation-specific preprocessing on two machines (25Gbps, single-digit ms latency). FLOSS requires ≈2× less SPDZ preprocessing than PermNet.

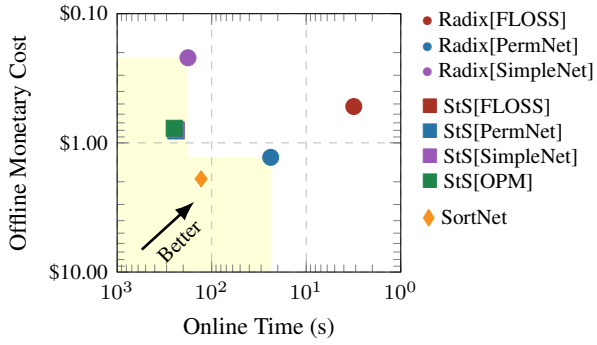


Figure 6: Cost summary of protocols for sorting 2^{13} 32-bit decomposed values on two machines (25Gbps, single-digit ms latency); OPM uses $T = 2^4$. Radix-Sort-and-Shuffle instantiated with FLOSS has $\geq 5\times$ faster online time and $1.5\times$ cheaper offline monetary cost than prior protocols outside of SimpleNet. We highlight prior work in yellow.

Table 3 breaks SPDZ-based offline costs into SPDZ-specific correlated randomness (e.g., multiplication triples) and permutation-specific preprocessing (e.g., authenticated shuffle tuples for FLOSS); the former dominates at small sizes, while the latter becomes comparable by 2^{20}.

Recall, that our implementation replaces MASCOT SPDZ preprocessing in the offline phase with LowGear at a certain instance size. This surfaces as a dip in the plots in Figure 5. Lastly, in terms of scaling, the offline computational and communication costs of FLOSS, PermNet, SimpleNet, and OPM scale on the order of $O(n \log n)$, as all constructions rely on logarithmic-depth permutation networks.

6.4 Sorting Costs

We compare the costs of combining different shuffling protocols with different sorting protocols for sorting bit-decomposed 32-bit values. As shown in Figure 6, which summarizes the offline AWS compute cost vs. online-time tradeoff for sorting input size 2^{13}, Radix[FLOSS] achieves the best offline cost efficiency.

Radix[FLOSS] achieves the lowest online shuffle cost across all evaluated input sizes. For inputs ranging from 2^9 to 2^{13}, the online shuffle time of Radix[FLOSS] increases $3\times$, while online communication increases $16\times$. In contrast, Radix[PermNet] incurs significantly higher online shuffle overheads, with shuffle time increasing $6\times$ and communication increasing $15.5\times$. The online shuffle cost of PermNet comprises the majority of its online sorting cost (approximately 85–90%), whereas the online shuffle cost of FLOSS accounts for only a small fraction (approximately 20–30%) of the total online sorting cost. Asymptotically, the online computational overhead of Radix[FLOSS] is $O(v \cdot n)$, while all other evaluated online shuffles incur $O(v \cdot n \log n)$ overhead. The $O(n \log n)$ online cost arises from dependence on logarithmic-depth network structures, as in SortNet sorting and OPM, and PermNet shuffling.

As expected, offline costs dominate the total runtime across

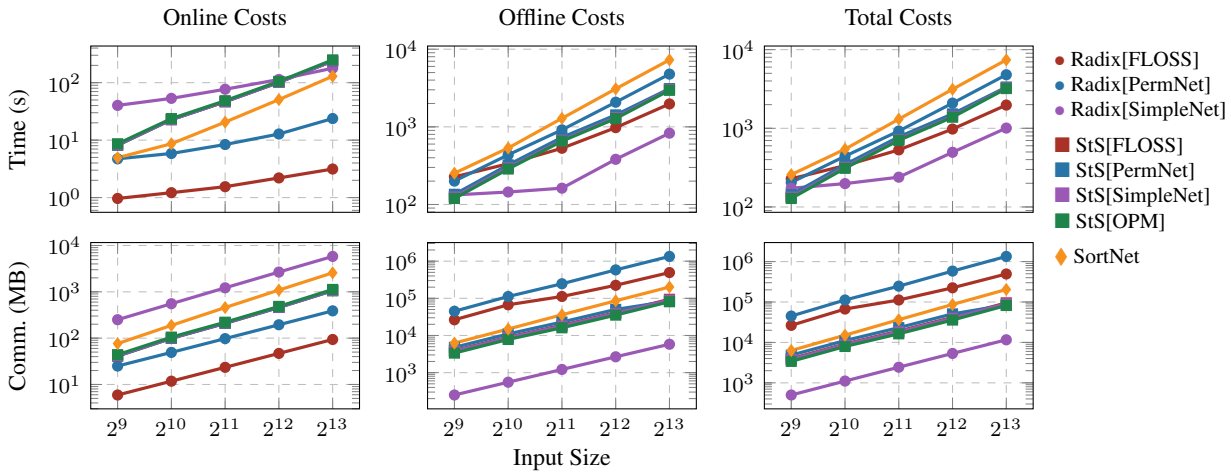


Figure 7: Computation time and communication costs for sorting 32-bit decomposed values on two machines (25Gbps, single-digit ms latency) varying from 2^9 to 2^{13} broken down by online and offline phases; OPM uses $T = 2^4$. FLOSS when combined with the Radix-Sort-and-Shuffle algorithm has lower online costs than all baselines.

all constructions. Radix[FLOSS]'s higher communication cost reflects the additional communication cost for permutation network evaluation via multiplication triples. Despite this, Radix[FLOSS] achieves the lowest growth in total cost as input size increases from 2^9 to 2^{13}. Radix[PermNet] exhibits higher total growth. Shuffle-then-Sort baselines incur similar total growth. In comparison, SortNet scales poorly. Overall, all preprocessing protocols exhibit $O(n \log n)$ offline scaling due to reliance on logarithmic-depth permutation networks, while the smaller constant factors of permutation networks and radix sorting explain the favorable growth behavior of Radix[FLOSS] and Radix[PermNet].

7 Concurrent Work

In addition to the prior work on secret-shared shuffling described in Section 3 (and summarized in Table 1), we highlight two recent, closely related malicious-secure constructions [SLD+26, GZZ24].

MOSAC [SLD+26] is concurrent and independent work in the same malicious 2PC setting over SPDZ shares. The key idea in MOSAC is that its preprocessed shuffle tuple is itself an authenticated SPDZ sharing: for a permutation π held by one party, the tuple provides authenticated shares $\langle \vec{a} \rangle$ of a uniformly random mask and $\langle \vec{\tilde{a}} \rangle$ of its image $\vec{\tilde{a}} = \pi(\vec{a})$. The mask is thus secret-shared, unlike FLOSS, where P_1 holds $\vec{a}$ in the clear.

In MOSAC's most efficient variant [SLD+26], to shuffle $\langle \vec{x} \rangle$, the variant first forms a key-multiplied copy $\langle \psi \vec{x} \rangle$ via an online Beaver multiplication with a secret shared random field element $\langle \psi \rangle$. The parties then apply π to each of $\langle \vec{x} \rangle$ and $\langle \psi \vec{x} \rangle$: they open the blinded $\vec{\delta} = \vec{x} - \vec{a}$ (which hides $\vec{x}$ since $\vec{a}$ is uniform), the holder of π permutes it and re-shares $\pi(\vec{\delta})$ as a fresh authenticated input, and they locally compute $\langle \pi(\vec{x}) \rangle = \langle \pi(\vec{\delta}) \rangle + \langle \vec{\tilde{a}} \rangle$; $\langle \psi \vec{x} \rangle$ is permuted the same way under a second tuple mask, giving $\langle \pi(\psi \vec{x}) \rangle$. Finally, the parties verify the layer-two MAC, checking that $\pi(\psi \vec{x}) = \psi \cdot \pi(\vec{x})$. We point out that use of the multiplicative key ψ serves two purposes here: it prevents the permutation guessing selective failure attack (see Section 3), as well as binds the permuter to input the correct $\pi(\vec{\delta})$ back into the protocol. FLOSS instead bakes its multiplicative key, κ, into the tuple and applies the permutation within local share arithmetic, avoiding both the online multiplication and the re-share: a FLOSS permutation gate is a single round of communication with a deferred check, against the two additional rounds MOSAC needs.

Gao et al. [GZZ24] apply a similar multiplicative-key authentication in the online phase against the selective-failure attack, but emphasize generality and extensibility: their permute-in-turn construction is generic over the secret-sharing scheme and the number of parties, with the composed permutation hidden from all. FLOSS instead specializes to and optimizes for the malicious two-party SPDZ setting.

Acknowledgements

We thank Xiangfu Song and the other authors of [SYB+24] for granting us access to their code for the OPM construction and permission to redistribute it, as well as for their invaluable feedback on an earlier version of this paper. We also thank Stefano Tessaro for his helpful suggestions on the protocol design and proof, and the anonymous USENIX Security reviewers for their insights and shepherding.

This work was funded in part by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA –

390781972.

Ethical Considerations

In this work, we present FLOSS, a protocol for faster linear-time online shuffles in the malicious-secure 2PC setting. Secure shuffling is a core building block in many privacy-preserving protocols, including those for privacy-preserving analytics, advertising, and anonymous broadcasting. Consequently, FLOSS may eventually be deployed to process sensitive and socially significant data.

Stakeholders. We identify the stakeholder groups that may be affected by our research. End-users may have their data processed by privacy-preserving protocols utilizing FLOSS. This can be customers whose demographic or browser data is collected and analyzed via privacy-preserving protocols. On the other side, we will also consider companies and organizations that wish to collect data on their users.

Impacts. Most services currently collect and analyze data in-the-clear. Providing more efficient privacy-preserving primitives can lead to wider deployment of privacy-preserving collection. If services migrate from data collection in-the-clear to privacy-preserving data collection, this is beneficial to end-users' privacy. We anticipate that as privacy-preserving protocols become more efficient, there may even be standards and regulations that encourage/require their use.

At the same time, migrating an existing system to a privacy-compliant one is challenging. MPC-based systems are still very new and are hard to implement correctly. Any sort of cryptographic deployment requires significant resources for a company to ensure that a system is securely deployed and hardened. If not, end-users may receive weaker privacy protections than promised, potentially leaking sensitive data that otherwise would not have been trusted to share.

Mitigations. FLOSS is a protocol for privacy-preserving computation. It can be used to compute any function over user data. Users should be wary of companies increasing collection of sensitive statistics under the guise of privacy-preserving computation. Companies and organizations should think carefully about the privacy implications of the statistics they are computing over user data and be transparent about what aggregated data is being computed. As such, privacy-preserving computation systems should be paired with transparency systems to help mitigate unexpected data collection.

Decision. We believe that the benefits of privacy-preserving computation to end-user privacy are large and the potential harms can be appropriately mitigated.

Open Science

We provide open access to all artifacts described in this paper.

All code required to reproduce the FLOSS benchmarks is available via a public DOI link.[5] The artifact includes scripts to automatically set up the AWS instances and install prerequisites, as well as a README file with instructions on how to run the experiments and tests. It also includes:

- Our full system implementation of FLOSS, and a framework for constructing 2PC arithmetic permutation circuits.
- MP-SPDZ compilation instructions for MASCOT and LowGear.
- Our MP-SPDZ circuits for Quicksort and sorting network.
- The source code and benchmarks for OPM, obtained from the authors of [SYB+24].
- Scripts to run and plot the aforementioned experiments.

[5] https://doi.org/10.5281/zenodo.20278557

References

[ACD+24] Erik Anderson, Melissa Chase, F. Betül Durak, Kim Laine, and Chenkai Weng. Precio: Private aggregate measurement via oblivious shuffling. In *CCS*, 2024.

[AFO+21] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *CCS*, 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. In *CCS*, 2022.

[AKK+23] Pranav Shriram A, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, Bhavish Raj Gopal, and Somya Sangal. Ruffle: Rapid 3-party shuffle protocols. *PoPETS*, 2023.

[AKTZ17] Nikolaos Alexopoulos, Aggelos Kiayias, Riivo Talviste, and Thomas Zacharias. Mcmix: Anonymous messaging via secure multiparty computation. In *USENIX Security*, 2017.

[Ama26] Amazon Web Services. Specifications for Amazon EC2 compute optimized instances, 2026.

[BBM+25] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. ORQ: complex analytics on private data with strong security guarantees. In *SOSP*, 2025.

[BD02] Bruno Beauquier and Éric Darrot. On arbitrary size waksman networks and their vulnerability. *Parallel Process. Lett.*, 2002.

[BDG+22] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. Secret-shared joins with multiplicity from aggregation trees. In *CCS*, 2022.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO*, 1991.

[BG12] Stephanie Bayer and Jens Groth. Efficient zero-knowledge argument for correctness of a shuffle. In *EUROCRYPT*, 2012.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *STOC*, 1988.

[BKKS25] Andreas Brüggemann, Nishat Koti, Varsha Bhat Kukkala, and Thomas Schneider. Multicent: Secure and scalable computation of centrality measures on multilayer graphs. *PoPETS*, 2025.

[BLT14] Dan Bogdanov, Sven Laur, and Riivo Talviste. A practical analysis of oblivious sorting algorithms for secure multi-party computation. In *NordSec*, 2014.

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *CCS*, 1993.

[BTW12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. Deploying secure multi-party computation for financial data analysis - (short paper). In *FC*, 2012.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *FOCS*, 2001.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In *NSDI*, 2017.

[CCL15] Ran Canetti, Asaf Cohen, and Yehuda Lindell. A simpler variant of universally composable security for standard multiparty computation. In *CRYPTO*, 2015.

[CGP20] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *ASIACRYPT*, 2020.

[CHI+19] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. An efficient secure three-party sorting protocol with an honest majority. *IACR ePrint Archive*, Paper 2019/695, 2019.

[CJK+23] Benjamin M. Case, Richa Jain, Alex Koshelev, Andy Leiserson, Daniel Masny, Ben Savage, Erik Taubeneck, Martin Thomson, and Taiki Yamaguchi. Interoperable private attribution: A distributed attribution and aggregation protocol. *IACR ePrint Archive*, Paper 2023/437, 2023.

[Cle86] Richard Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *STOC*, 1986.

[div] Divvi up: A privacy respecting telemetry service. https://divviup.org/.

[DKL+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure MPC for dishonest majority - or: Breaking the SPDZ limits. In *ESORICS*, 2013.

[DN03] Ivan Damgård and Jesper Buus Nielsen. Universally composable efficient multiparty computation from threshold homomorphic encryption. In *CRYPTO*, 2003.

[DNO25] Samuel Dittmer, Rohit Nema, and Rafail Ostrovsky. Linear secret-shared shuffle with malicious security. *IACR ePrint Archive*, Paper 2025/2137, 2025.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel P. Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *CRYPTO*, 2012.

[EB22] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *NDSS*, 2022.

[FJKW15] Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In *ASIACRYPT*, 2015.

[FO21] Brett Hemenway Falk and Rafail Ostrovsky. Secure merge with $O(n \log \log n)$ secure operations. In *ITC*, 2021.

[FOSZ23] Brett Hemenway Falk, Rafail Ostrovsky, Matan Shtepel, and Jacob Zhang. Gigadoram: Breaking the billion address barrier. In *USENIX Security*, 2023.

[GHJ+25] Daniel Günther, Marco Holz, Benjamin Judkewitz, Helen Möllering, Benny Pinkas, Thomas Schneider, and Ajith Suresh. Privacy-preserving epidemiological modeling on mobile graphs. *IEEE Trans. Inf. Forensics Secur.*, 2025.

[GKLX22] S. Dov Gordon, Jonathan Katz, Mingyu Liang, and Jiayu Xu. Spreading the privacy blanket: - differentially oblivious shuffling for differential privacy. In *ACNS*, 2022.

[GL05] Shafi Goldwasser and Yehuda Lindell. Secure multi-party computation without agreement. *J. Cryptol.*, 2005.

[GPP+25] Tim Geoghegan, Christopher Patton, Brandon Pitman, Eric Rescorla, and Christopher A. Wood. Distributed Aggregation Protocol for Privacy Preserving Measurement. Internet-Draft draft-ietf-ppm-dap-14, Internet Engineering Task Force, February 2025. Work in Progress.

[Gro09] Jens Groth. Linear algebra with sub-linear zero-knowledge arguments. In *CRYPTO*, 2009.

[Gro10] Jens Groth. A verifiable secret shuffle of homomorphic encryptions. *J. Cryptol.*, 2010.

[GZZ24] Jiacheng Gao, Yuan Zhang, and Sheng Zhong. Multiparty shuffle: Linear online phase is almost for free. *IACR ePrint Archive*, Paper 2024/1936, 2024.

[HKI+12] Koki Hamada, Ryo Kikuchi, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. Practically efficient multi-party sorting protocols from comparison sort algorithms. In *ICISC*, 2012.

[HKRS20] Marco Holz, Ágnes Kiss, Deevashwer Rathee, and Thomas Schneider. Linear-complexity private function evaluation is practical. In *ESORICS*, 2020.

[JKU11] Kristján Valur Jónsson, Gunnar Kreitz, and Misbah Uddin. Secure multi-party sorting and applications. *IACR ePrint Archive*, Paper 2011/122, 2011.

[KKPG24] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Graphiti: Secure graph computation made more scalable. In *CCS*, 2024.

[KLDF16] Albert Kwon, David Lazar, Srinivas Devadas, and Bryan Ford. Riffle: An efficient communication system with strong anonymity. *PoPETS*, 2016.

[KNOS25] Alexander Kyster, Frederik Huss Nielsen, Sabine Oechsner, and Peter Scholl. Rushing at SPDZ: on the practical security of malicious MPC implementations. In *IEEE S&P*, 2025.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. MASCOT: faster malicious arithmetic secure computation with oblivious transfer. In *CCS*, 2016.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: Making SPDZ great again. In *EUROCRYPT*, 2018.

[KS08] Vladimir Kolesnikov and Thomas Schneider. A practical universal circuit construction and secure evaluation of private functions. In *FC*, 2008.

[Lab26] Primate Labs. Amazon EC2 c5.xlarge benchmark results (Intel Xeon platinum 8124M), 2026. Geekbench Browser — L3 Cache = 24.8 MB for Intel Xeon Platinum 8124M on EC2.

[Lau21] Peeter Laud. Linear-time oblivious permutations for SPDZ. In *CANS*, 2021.

[Lin17] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. In *Tutorials on the Foundations of Cryptography*. Springer, 2017.

[LYK+19] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. Honeybadgermpc and asynchromix: Practical asynchronous MPC and its application to anonymous communication. In *CCS*, 2019.

[mps] `MP-SPDZ`: Multi-protocol spdz library. https://github.com/data61/MP-SPDZ.

[MRR20] Payman Mohassel, Peter Rindal, and Mike Rosulek. Fast database joins and PSI for secret shared data. In *CCS*, 2020.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *EUROCRYPT*, 2013.

[MSS14] Payman Mohassel, Seyed Saeed Sadeghian, and Nigel P. Smart. Actively secure private function evaluation. In *ASIACRYPT*, 2014.

[Nef01] C. Andrew Neff. A verifiable secret shuffle and its application to e-voting. In *CCS*, 2001.

[NWI+15] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. Graphsc: Parallel secure computation made easy. In *IEEE S&P*, 2015.

[PLS+25] Andrew Paseltiner, Andy Leiserson, Benjamin Savage, Charlie Harrison, and Martin Thomson. Privacy-Preserving Attribution: Level 1. First-public-working-draft, World Wide Web Consortium, April 2025. Work in Progress.

[PRRS25] Stanislav Peceny, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. Efficient permutation correlations and batched random access for two-party computation. In *PKC*, 2025.

[SA19] Nigel P. Smart and Younes Talibi Alaoui. Distributing any elliptic curve based protocol. In *IMACC*, 2019.

[SLD+26] Xiangfu Song, Xiaojian Liang, Ye Dong, Jianli Bai, Pu Duan, Changyu Dong, Tianwei Zhang, and Ee-Chien Chang. Secret-shared shuffle from authenticated correlations. volume 16552 of *Lecture Notes in Computer Science*, pages 321–354. Springer, 2026.

[SYB+24] Xiangfu Song, Dong Yin, Jianli Bai, Changyu Dong, and Ee-Chien Chang. Secret-shared shuffle with malicious security. In *NDSS*, 2024.

[Wak68] Abraham Waksman. A permutation network. *J. ACM*, 1968.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, Abhi Shelat, and Elaine Shi. SCORAM: oblivious RAM for secure computation. In *CCS*, 2014.

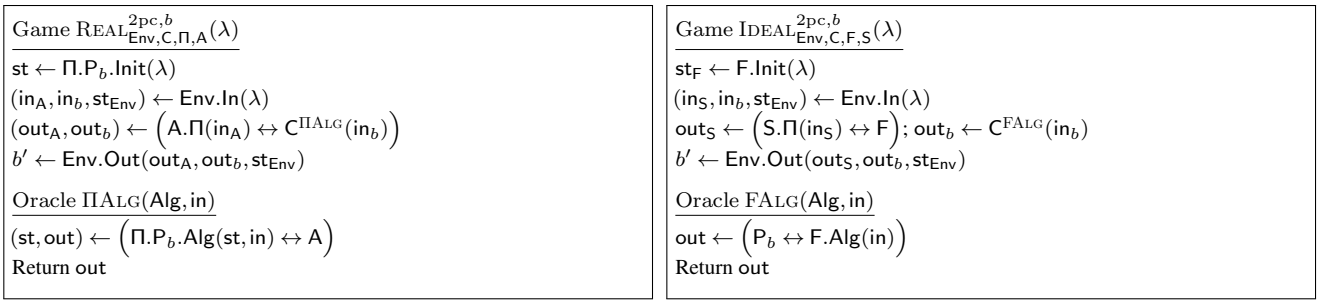


Figure 8: Two-party UC security definition.

A Universal Composability Security Model

We use Universal Composability (UC) [Can01] to prove the security of all our protocols. In particular, we adapt the Simplified UC framework (SUC) of Canetti et al. [CCL15]. Our adaptations include further simplifications due to only considering two-party protocols with static corruptions, as well as added support for control functions to model adversary-controlled and adaptively chosen circuits.

In this section we provide a brief overview, but assume some familiarity with the UC framework; we refer the reader to [Can01, CCL15] for more details.

Modeling corrupt parties. The security of a protocol Π is defined by comparing a "real" execution of Π to an "ideal" execution of a trusted and interactive functionality F. In the real execution, an adversary A corrupts one party P_b (where $b \in \{0,1\}$) statically at the beginning of the protocol execution. The adversary controls all messages sent by the corrupted party, and it can deviate arbitrarily from the protocol specification and adaptively choose its strategy based on observed protocol messages. In the ideal execution, a simulator S plays the role of the adversary but interacts with an ideal functionality F instead of a real protocol.

Modeling honest parties. We assume a strong threat model where the adversary controls the circuit being evaluated. To model this, we use the notion of control functions [Can01], where an adversary-controlled algorithm C decides which operations to invoke based on intermediate results (e.g., choosing what to multiply or open based on a previously opened value). In the real world, C controls the behavior of the honest party; in the ideal world, it makes direct calls to the ideal functionality.

Modeling communication. We assume the adversary controls the entire communication channel. This is natural in the two-party setting, since all communication between the parties in the real world includes a corrupt party.

In the ideal world, the simulator additionally controls the communication between the honest party and the ideal functionality, including the ability to delay or drop messages. Importantly, however, the simulator cannot read or modify the contents of these messages. This standard abstraction of adversarial control immediately captures the adversary's ability to prevent the honest party from receiving the final output, which is inherent in the MPC with abort setting [Cle86, GL05].

Composition with hybrid functionalities. Protocols may be analyzed in hybrid models where they have access to ideal functionalities. In this model, protocol execution may additionally involve oracle calls to ideal functionalities. Similarly to the ideal world, the adversary controls the routing of communication between the honest party and any hybrid functionalities (but cannot read or modify the contents of these messages).

Formally, we write $\Pi[\mathsf{F}_1, \ldots, \mathsf{F}_k]$ to denote that protocol Π operates in a hybrid model with access to ideal functionalities $\mathsf{F}_1, \ldots, \mathsf{F}_k$. Standard composition theorems [Can01, CCL15] guarantee that if $\Pi[\mathsf{F}]$ securely realizes some functionality F' in the F-hybrid model, and ρ securely realizes F, then the composed protocol $\Pi[\rho]$ (where all calls to F are replaced by executions of ρ) securely realizes F'.

Distinguishing advantage. The distinguishing party in UC is called the environment (Env), which controls all possible external contexts in which a protocol might be executed. The environment chooses the inputs for both parties and sees the output of both parties. Crucially, the adversary can send messages to the environment during protocol execution, so the simulator must produce a consistent intermediate view without rewinding or forking.

We say that protocol Π UC-realizes functionality F if for every adversary A against Π, there exists a simulator S such that no efficient environment Env can distinguish between the two executions. We define that formally below.

Definition 1 (Distinguishing Advantage for UC Security of Two-Party Protocols)**.** *Let* $\text{REAL}^{2\text{pc},b}_{\mathsf{Env},\mathsf{C},\Pi,\mathsf{A}}$ *and* $\text{IDEAL}^{2\text{pc},b}_{\mathsf{Env},\mathsf{C},\mathsf{F},\mathsf{S}}$ *be the probabilistic games defined in Figure 8. The distinguishing advantage of* Env *between a real execution of protocol* Π *with* A *and*

an ideal execution of functionality F *with simulator* S *when party* P_b *is corrupted, is defined as:*

$$\mathsf{Adv}^{2\text{pc},b}_{\mathsf{Env},\mathsf{C},\Pi,\mathsf{A},\mathsf{F},\mathsf{S}}(\lambda) = |\Pr[\textsc{Real}^{2\text{pc},b}_{\mathsf{Env},\mathsf{C},\Pi,\mathsf{A}}(\lambda) = 1] - \Pr[\textsc{Ideal}^{2\text{pc},b}_{\mathsf{Env},\mathsf{C},\mathsf{F},\mathsf{S}}(\lambda) = 1]|$$

Definition 2 (UC Security for Two-Party Protocols)**.** *A two-party protocol* Π *UC-realizes an ideal functionality* F *if for all* $b \in \{0,1\}$*, for all probabilistic polynomial time (PPT) control functions* C *for the honest party* P_{1-b}*, and for all PPT adversaries* A *corrupting party* P_b*, there exists a PPT simulator* S *such that for all PPT environments* Env*:*

$$\mathsf{Adv}^{2\text{pc},b}_{\mathsf{Env},\mathsf{C},\Pi,\mathsf{A},\mathsf{F},\mathsf{S}}(\lambda) \leq \mathrm{negl}(\lambda)$$

where $\mathsf{negl}(\lambda)$ *is a negligible function in the UC security parameter of the protocol* λ*.*

B Building Block Protocols

B.1 Commitment Functionality and Protocol

In Figure 9 we describe the ideal functionality and a hash-and-reveal protocol for a hiding and binding commitment scheme. This scheme and functionality appear in Appendix A of [DKL+13], where it is also proved UC-secure in the random oracle model [BR93].

Functionality $\mathsf{F}_{\mathsf{Comm}}$

$\mathsf{Init}(\lambda)$: Initialize $T \leftarrow \{\cdot\}$ mapping id to committed values.

$\mathsf{Comm}(\mathsf{id}, v)$ from P_b:

1. Require $\mathsf{id} \notin T$.
2. Store $T[\mathsf{id}] \leftarrow (b, v)$.
3. Send $(\texttt{comm}, \mathsf{id}, b)$ to all parties.

$\mathsf{Open}(\mathsf{id})$ from P_b:

1. Require $\mathsf{id} \in T$ and $(b', v) \leftarrow T[\mathsf{id}]$; $b' = b$.
2. Send $(\texttt{open}, \mathsf{id}, v)$ to all parties.

Protocol $\mathsf{HComm}[\mathsf{H}]$

$\mathsf{Init}(\lambda)$:

1. Initialize $T \leftarrow \{\cdot\}$ mapping id to commitment material.
2. Initialize $C \leftarrow \{\cdot\}$ mapping id to received commitments.

$\mathsf{P}_b.\mathsf{Comm}(\mathsf{id}, v) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{RcvComm}()$:

1. Require $\mathsf{id} \notin T \cup C$.
2. Sample $r \leftarrow\!\!\$\ \{0,1\}^\lambda$ and compute $cm \leftarrow \mathsf{H}(v \,\|\, r)$.
3. $T[\mathsf{id}] \leftarrow (v, r, cm)$.
4. P_b sends (id, cm) to P_{1-b} who stores $C[\mathsf{id}] \leftarrow cm$.
5. Output $(\texttt{comm}, \mathsf{id}, b)$.

$\mathsf{P}_b.\mathsf{Open}(\mathsf{id}) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{VerOpen}(\mathsf{id})$:

1. Require $\mathsf{id} \in T$ and $\mathsf{id} \in C$.
2. P_b reads $(v, r, cm) \leftarrow T[\mathsf{id}]$ and sends (id, v, r).
3. P_{1-b} reads $cm \leftarrow C[\mathsf{id}]$ and checks $cm = \mathsf{H}(v \,\|\, r)$. If the check fails, P_{1-b} aborts.
4. Output $(\texttt{open}, \mathsf{id}, v)$.

Figure 9: Commitment functionality $\mathsf{F}_{\mathsf{Comm}}$ and hash and reveal protocol $\mathsf{HComm}[\mathsf{H}]$ where hash function H is modeled as a random oracle.

B.2 SPDZ Protocol for Arithmetic Circuits

For completeness, we present a description of the SPDZ protocol for malicious-secure computation of arithmetic circuits over a finite field $\mathbb{F}$ that is of exponential size in the security parameter [DPSZ12, DKL+13]. The ideal functionality $\mathsf{F}_{\mathsf{ArithCirc}}$ for arithmetic circuit computation is given in Figure 10, the SPDZ protocol realizing this functionality is given in Figure 11, and the preprocessing functionality and authentication check subprotocol are given in Figure 12. We briefly describe the key components of SPDZ below.

Inspiration: Semi-honest MPC. The inspiration for SPDZ is the following method to compute arithmetic circuits in the semi-honest setting. At each point in the circuit, parties hold additive shares $[\![x]\!]_0$ and $[\![x]\!]_1$ such that $x = [\![x]\!]_0 + [\![x]\!]_1$ for the value x of each wire. Each party can input a value x by giving the other party an additive share, and parties can locally evaluate addition and scalar multiplication gates on secret shared data without interaction. To compute a multiplication gate on inputs v_1 and v_2, parties use preprocessed Beaver triples [Bea91]: each party P_b holds input shares $[\![v_1]\!]_b$ and $[\![v_2]\!]_b$, and consumes shares of a triple $([\![a]\!]_b, [\![b]\!]_b, [\![c]\!]_b)$ satisfying $c = ab$, with a and b uniformly random and unknown to either party. Each party computes

and sends $[\![d]\!]_b = [\![v_1]\!]_b - [\![a]\!]_b$ and $[\![e]\!]_b = [\![v_2]\!]_b - [\![b]\!]_b$ to the other party, which together open $d = v_1 - a$ and $e = v_2 - b$. Each party then locally computes their share of the product as $[\![v_1 \cdot v_2]\!]_b = [\![c]\!]_b + d \cdot [\![b]\!]_b + e \cdot [\![a]\!]_b + d \cdot e/2$. To open a value, parties send their shares to each other and sum them using a commit and reveal protocol.

The efficiency of this protocol relies on the ability to generate Beaver triples, which is a well studied problem (e.g. [DPSZ12, DKL+13, KOS16, KPR18]) and is beyond the scope of this overview. However, a key observation is that the triples can be generated in an offline preprocessing phase before the inputs are known, allowing the online phase to be efficient.

SPDZ: Arithmetic circuits in the malicious setting. The SPDZ protocol lifts the semi-honest protocol to the malicious setting by adding authentication tags to all secret-shared values. At each point in the circuit, parties hold *authenticated shares*: each party P_b holds an additive share $[\![x]\!]_b$ of a secret value x together with a share $[\![\gamma x]\!]_b$ of the authentication tag γx, where $\gamma = [\![\gamma]\!]_0 + [\![\gamma]\!]_1$ is a global authentication key unknown to either party individually. We denote the pair $\langle x \rangle_b = ([\![x]\!]_b, [\![\gamma x]\!]_b)$ as an authenticated share.

The preprocessing functionality $\mathsf{F}_{\mathsf{SPDZPrep}}$ (Figure 12, left) generates shares of a random authentication key $[\![\gamma]\!]_b$ for each party, authenticated Beaver triples $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ where $c = ab$, and input tuples of the form $(r, \langle r \rangle)$ where r is known to one of the parties and is secret shared between both.

To input a value x, party P_b sends $\epsilon = x - r$ to P_{1-b}, and both parties locally compute $\langle x \rangle = \langle r \rangle + \epsilon$. From there, evaluating the circuit follows the same structure as the semi-honest protocol, but operates over authenticated shares. Linear operations are performed locally on both the data shares and tag shares. To compute a multiplication gate on inputs v_1 and v_2, each party P_b holds authenticated input shares $\langle v_1 \rangle_b$ and $\langle v_2 \rangle_b$, and consumes an authenticated triple $(\langle a \rangle_b, \langle b \rangle_b, \langle c \rangle_b)$. Each party computes and sends $[\![d]\!]_b = [\![v_1]\!]_b - [\![a]\!]_b$ and $[\![e]\!]_b = [\![v_2]\!]_b - [\![b]\!]_b$ to open d and e. Each party then locally computes their authenticated share of the product as $\langle v_1 \cdot v_2 \rangle_b = \langle c \rangle_b + d \cdot \langle b \rangle_b + e \cdot \langle a \rangle_b + d \cdot e/2$.

To reconstruct a value z, parties must first verify that all previously opened values (the d and e values from multiplications) have not been tampered with. Verification uses the authentication check Auth (Figure 12, right): given opened values $z_1, \ldots, z_n$ and their tag shares $[\![\gamma z_i]\!]_b$, parties check that the tags are consistent with the values and the secret key γ, without revealing γ. Concretely, parties jointly sample a random coefficient r via committed coin-tossing, compute the random linear combination $\ell = \sum_i r^{i-1} z_i$ and tag share $[\![\gamma \ell]\!]_b = \sum_i r^{i-1} [\![\gamma z_i]\!]_b$, then each party computes $[\![\epsilon]\!]_b = [\![\gamma \ell]\!]_b - [\![\gamma]\!]_b \cdot \ell$ and parties verify $[\![\epsilon]\!]_0 + [\![\epsilon]\!]_1 = 0$ by exchanging committed shares. If any value was tampered with, the check fails with overwhelming probability, and both parties must abort the protocol. After the pending values pass verification, parties exchange shares $[\![z]\!]_b$ to reconstruct z, then run Auth again to verify z itself before concluding that it is the correct output. We point out that a circuit may contain many reconstruct gates, and evaluation can continue after this step.

We refer the reader to Figure 11 for the details of the SPDZ protocol.

Pitfall: MAC key leakage via failed checks. Kyster et al. [KNOS25] identify a vulnerability in SPDZ implementations that use parallelism or error recovery. They observe that the authentication check only hides γ when it *succeeds*. However, at the cost of failing the check, an adversary can learn the authentication key. Consider an adversary that sends an incorrect share $[\![x]\!]'_b = [\![x]\!]_b + \delta$ when opening x, causing parties to reconstruct $x' = x + \delta$. In the authentication check, after the honest party opens their commitment to $[\![\epsilon]\!]_{1-b}$, the adversary can compute $[\![\epsilon]\!]_b + [\![\epsilon]\!]_{1-b} = \gamma x - \gamma x' = -\gamma\delta$ and recover γ.

This cannot be exploited in sequential execution since the adversary learns γ only after the check fails. However, with multi-threading or error recovery, the adversary can learn γ in one context and exploit it to forge MACs in another. To fix this, Kyster et al. propose masking ϵ with a random value before opening: if $\epsilon = 0$ the masked value remains zero, otherwise it is randomized and reveals nothing about γ.

Both this vulnerability and the fix apply to our protocol FLOSS as well, both in the blind and non-blind authentication checks. However, since it is orthogonal to our contribution, we do not discuss it further.

C Security Proof for Online Protocol

In this section we prove the security of our online protocol FLOSS, assuming the security of the offline protocol to generate shuffle tuples, as well as the security of the commitment scheme used.

Theorem 2 (Security of FLOSS)**.** Let $\mathbb{F}$ be a finite field with $|\mathbb{F}| \geq 2^\lambda$. The protocol $\mathsf{FLOSS}[\mathbb{F}, \mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{ShufTuplePrep}}, \mathsf{F}_{\mathsf{Comm}}]$ (Figure 14) UC-realizes the ideal functionality $\mathsf{F}_{\mathsf{ArithPermCirc}}[\mathbb{F}]$ (Figure 13) in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{ShufTuplePrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid model.

Specifically, for all $b \in \{0, 1\}$ and for all PPT adversaries A corrupting party P_b, there exists a PPT simulator S such that for all PPT environments Env the advantage $\mathsf{Adv}^{\mathrm{2pc},b}_{\mathsf{Env},\mathsf{C},\mathsf{FLOSS},\mathsf{A},\mathsf{F}_{\mathsf{ArithPermCirc}},\mathsf{S}}(\lambda)$ is negligible in λ.

Functionality $\mathsf{F}_{\mathsf{ArithCirc}}$ [$\mathbb{F}$]

$\mathsf{Init}(\lambda)$:

1. $T \leftarrow \{\cdot\}$: maps id to values in $\mathbb{F}$.
2. $ictr_0 \leftarrow 0$; $ictr_1 \leftarrow 0$: input operation counter.
3. $mctr \leftarrow 0$: multiplication operation counter.
4. $prep \leftarrow 0$: preprocessing flag.

$\mathsf{Prep}(n_{i,0}, n_{i,1}, n_m)$ from P_b:

1. Require $prep = 0$, preprocessing only called once.
2. If no pending message from P_{1-b}, set $(\texttt{prep}, n_{i,0}, n_{i,1}, n_m)$ from P_b as pending and return. Else parse $(\mathsf{op}, n'_{i,0}, n'_{i,1}, n'_m)$ from P_{1-b} and require $\mathsf{op} = \texttt{prep}$.
3. If mismatch, $\exists b'$, $n_{i,b} \neq n'_{i,b}$ or $n_m \neq n'_m$, send $\texttt{abort}$ to all parties.
4. Else, $ictr_b \leftarrow n_{i,b}$ and $mctr \leftarrow n_m$, $prep \leftarrow 1$.
5. Send $(\texttt{prep}, n_{i,0}, n_{i,1}, n_m)$ to all parties.

$\mathsf{ScalAdd}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$ from P_b:

1. Require $\mathsf{id}_1 \in T$, $\mathsf{id}_2 \notin T$, and no pending message from P_b.
2. If no pending message from P_{1-b}, set $(\texttt{scadd}, \mathsf{id}_1, \alpha, \mathsf{id}_2)$ from P_b as pending and return. Else parse $(\mathsf{op}, \mathsf{id}'_1, \alpha', \mathsf{id}'_2)$ from P_{1-b} and require $\mathsf{op} = \texttt{scadd}$.
3. If mismatch, $\vee_{j\in\{1,2\}}\ \mathsf{id}_j \neq \mathsf{id}'_j$, or $\alpha \neq \alpha'$, store $T[\mathsf{id}_2], T[\mathsf{id}'_2] \leftarrow \perp$.
4. Else, store $T[\mathsf{id}_2] \leftarrow T[\mathsf{id}_1] + \alpha$.
5. Send $(\texttt{scadd}, \mathsf{id}_1, \alpha, \mathsf{id}_2)$ to P_b and $(\texttt{scadd}, \mathsf{id}'_1, \alpha', \mathsf{id}'_2)$ to P_{1-b}.

$\mathsf{Add}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from P_b:

1. Require $\mathsf{id}_1, \mathsf{id}_2 \in T$, $\mathsf{id}_3 \notin T$, and no pending message from P_b.
2. If no pending message from P_{1-b}, set $(\texttt{add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from P_b as pending and return. Else parse $(\mathsf{op}, \mathsf{id}'_1, \mathsf{id}'_2, \mathsf{id}'_3)$ from P_{1-b} and require $\mathsf{op} = \texttt{add}$.
3. If mismatch, $\vee_{j\in\{1,2,3\}}\ \mathsf{id}_j \neq \mathsf{id}'_j$, store $T[\mathsf{id}_3], T[\mathsf{id}'_3] \leftarrow \perp$.
4. Else, store $T[\mathsf{id}_3] \leftarrow T[\mathsf{id}_1] + T[\mathsf{id}_2]$.
5. Send $(\texttt{add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to P_b and $(\texttt{add}, \mathsf{id}'_1, \mathsf{id}'_2, \mathsf{id}'_3)$ to P_{1-b}.

$\mathsf{Input}(\mathsf{id}, v \in \mathbb{F})$ from P_b:

1. Require $\mathsf{id} \notin T$ and $ictr_b > 0$.
2. Store $T[\mathsf{id}] \leftarrow v$.
3. Decrement $ictr_b \leftarrow ictr_b - 1$.
4. Send $(\texttt{inp}, \mathsf{id})$ to all parties.

$\mathsf{Reconstruct}(\mathsf{id})$ from P_b:

1. Require $\mathsf{id} \in T$.
2. If no pending message from P_{1-b}, set $(\texttt{reconstruct}, \mathsf{id})$ from P_b as pending and return. Else parse $(\mathsf{op}, \mathsf{id}')$ from P_{1-b} and require $\mathsf{op} = \texttt{reconstruct}$.
3. If $\mathsf{id} \neq \mathsf{id}'$ or $T[\mathsf{id}] = \perp$, send $\texttt{abort}$ to all parties.
4. Else, send $(\texttt{reconstruct}, \mathsf{id}, T[\mathsf{id}])$ to all parties.

$\mathsf{ScalMul}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$ from P_b:

1. Require $\mathsf{id}_1 \in T$, $\mathsf{id}_2 \notin T$, and no pending message from P_b.
2. If no pending message from P_{1-b}, set $(\texttt{scmul}, \mathsf{id}_1, \alpha, \mathsf{id}_2)$ from P_b as pending and return. Else parse $(\mathsf{op}, \mathsf{id}'_1, \alpha', \mathsf{id}'_2)$ from P_{1-b} and require $\mathsf{op} = \texttt{scmul}$.
3. If mismatch, $\vee_{j\in\{1,2\}}\ \mathsf{id}_j \neq \mathsf{id}'_j$, or $\alpha \neq \alpha'$, store $T[\mathsf{id}_2], T[\mathsf{id}'_2] \leftarrow \perp$.
4. Else, store $T[\mathsf{id}_2] \leftarrow \alpha T[\mathsf{id}_1]$.
5. Send $(\texttt{scmul}, \mathsf{id}_1, \alpha, \mathsf{id}_2)$ to P_b and $(\texttt{scmul}, \mathsf{id}'_1, \alpha', \mathsf{id}'_2)$ to P_{1-b}.

$\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from P_b:

1. Require $\mathsf{id}_1, \mathsf{id}_2 \in T$, $\mathsf{id}_3 \notin T$, $mctr > 0$, and no pending message from P_b.
2. If no pending message from P_{1-b}, set $(\texttt{mul}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from P_b as pending and return. Else parse $(\mathsf{op}, \mathsf{id}'_1, \mathsf{id}'_2, \mathsf{id}'_3)$ from P_{1-b} and require $\mathsf{op} = \texttt{mul}$.
3. If mismatch, $\vee_{j\in\{1,2,3\}}\ \mathsf{id}_j \neq \mathsf{id}'_j$, store $T[\mathsf{id}_3], T[\mathsf{id}'_3] \leftarrow \perp$.
4. Else, store $T[\mathsf{id}_3] \leftarrow T[\mathsf{id}_1] \cdot T[\mathsf{id}_2]$.
5. Decrement $mctr \leftarrow mctr - 1$.
6. Send $(\texttt{mul}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to P_b and $(\texttt{mul}, \mathsf{id}'_1, \mathsf{id}'_2, \mathsf{id}'_3)$ to P_{1-b}.

Figure 10: Functionality for computation of arithmetic circuits over $\mathbb{F}$.

Protocol $\mathsf{SPDZ}[\mathbb{F}, \mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}}]$

$\mathsf{P}_b.\mathsf{Init}(\lambda)$:

1. $T \leftarrow \{\cdot\}$: maps id to shares in $\mathbb{F}$.
2. $L_{inp,0}, L_{inp,1} \leftarrow [\cdot]$: list of preprocessed input tuples.
3. $L_{mul} \leftarrow [\cdot]$: list of preprocessed multiplication tuples.
4. $L_{auth} \leftarrow [\cdot]$: list of pending authentication checks.
5. $[\![\gamma]\!]_b \leftarrow \perp$: share of authentication key.

$\mathsf{P}_0.\mathsf{Prep}(n_{inp,0}, n_{inp,1}, n_m) \leftrightarrow \mathsf{P}_1.\mathsf{Prep}(n_{inp,0}, n_{inp,1}, n_m)$:

1. P_0 and P_1 invoke $\mathsf{F}_{\mathsf{SPDZPrep}}(n_{inp,0}, n_{inp,1}, n_m)$ for each P_b to receive (see Fig. 12):
   - $L_{inp,b} \leftarrow \left[\left(r_{i,b}, \langle r_{i,b}\rangle_b = ([\![r_{i,b}]\!]_b, [\![\gamma r_{i,b}]\!]_b)\right)\right]_i^{n_{inp,b}}$
   - $L_{inp,1-b} \leftarrow \left[\left(\langle r_{i,1-b}\rangle_b = ([\![r_{i,1-b}]\!]_b, [\![\gamma r_{i,1-b}]\!]_b)\right)\right]_i^{n_{inp,1-b}}$
   - $L_{mul} \leftarrow [(\langle a_i\rangle_b, \langle b_i\rangle_b, \langle c_i\rangle_b)]_i^{n_m}$
   - $[\![\gamma]\!]_b$
2. Output $(\texttt{prep}, n_i, n_m)$.

$\mathsf{P}_b.\mathsf{ScalAdd}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$:

1. Require $\mathsf{id}_1 \in T$, $\mathsf{id}_2 \notin T$.
2. P_b with $([\![v_1]\!], [\![\gamma v_1]\!]) \leftarrow T[\mathsf{id}_1]$ computes:
   $$\langle v_2\rangle = ([\![v_2]\!], [\![\gamma v_2]\!]) \leftarrow ([\![v_1]\!] + \alpha/2, [\![\gamma v_1]\!] + [\![\gamma]\!] \cdot \alpha/2)$$
3. P_b stores $T[\mathsf{id}_2] \leftarrow \langle v_2\rangle$.
4. Output $(\texttt{scadd}, \mathsf{id}_1, \alpha, \mathsf{id}_2)$.

$\mathsf{P}_b.\mathsf{Add}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$:

1. Require $\mathsf{id}_1, \mathsf{id}_2 \in T$, $\mathsf{id}_3 \notin T$.
2. P_b with $([\![v_1]\!], [\![\gamma v_1]\!]) \leftarrow T[\mathsf{id}_1]$ and $([\![v_2]\!], [\![\gamma v_2]\!]) \leftarrow T[\mathsf{id}_2]$ computes:
   $$\langle v_3\rangle = ([\![v_3]\!], [\![\gamma v_3]\!]) \leftarrow ([\![v_1]\!] + [\![v_2]\!], [\![\gamma v_1]\!] + [\![\gamma v_2]\!])$$
3. P_b stores $T[\mathsf{id}_3] \leftarrow \langle v_3\rangle$.
4. Output $(\texttt{add}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$.

$\mathsf{P}_b.\mathsf{ScalMul}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$:

1. Require $\mathsf{id}_1 \in T$, $\mathsf{id}_2 \notin T$.
2. P_b with $([\![v_1]\!], [\![\gamma v_1]\!]) \leftarrow T[\mathsf{id}_1]$ computes:
   $$\langle v_2\rangle = ([\![v_2]\!], [\![\gamma v_2]\!]) \leftarrow ([\![v_1]\!] \cdot \alpha, [\![\gamma v_1]\!] \cdot \alpha)$$
3. P_b stores $T[\mathsf{id}_2] \leftarrow \langle v_2\rangle$.
4. Output $(\texttt{scmul}, \mathsf{id}_1, \alpha, \mathsf{id}_2)$.

$\mathsf{P}_b.\mathsf{Input}(\mathsf{id}, v \in \mathbb{F}) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{RcvInput}(\mathsf{id})$:

1. Require $\mathsf{id} \notin T$ and $L_{inp,b} \neq [\cdot]$.
2. P_b pops $(r, \langle r\rangle_b) \leftarrow L_{inp,b}$ and P_{1-b} pops $\langle r\rangle_{1-b} \leftarrow L_{inp,b}$.
3. P_b computes $\epsilon \leftarrow v - r$ and sends ϵ to P_{1-b}.
4. P_0 and P_1 with $([\![r]\!], [\![\gamma r]\!]) \leftarrow \langle r\rangle$ compute:
   $$\langle v\rangle = ([\![v]\!], [\![\gamma v]\!]) \leftarrow ([\![r]\!] + \epsilon, [\![\gamma r]\!] + [\![\gamma]\!] \cdot \epsilon)$$
5. P_0 and P_1 store $T[\mathsf{id}] \leftarrow \langle v\rangle$.
6. Output $(\texttt{inp}, \mathsf{id})$.

$\mathsf{P}_0.\mathsf{Reconstruct}(\mathsf{id}) \leftrightarrow \mathsf{P}_1.\mathsf{Reconstruct}(\mathsf{id})$:

1. Require $\mathsf{id} \in T$.
2. P_0 and P_1 run authentication check on all pending multiplication opens in L_{auth} (see Fig. 12):
   (a) $[(z_i, [\![\gamma z_i]\!])]_i \leftarrow L_{auth}$; $L_{auth} \leftarrow [\cdot]$
   (b) $\mathsf{P}_0.\mathsf{Auth}([\![\gamma]\!], [(z_i, [\![\gamma z_i]\!])]_i) \leftrightarrow \mathsf{P}_1.\mathsf{Auth}([\![\gamma]\!], [(z_i, [\![\gamma z_i]\!])]_i)$
3. P_0 and P_1 open $\langle v\rangle \leftarrow T[\mathsf{id}]$ to v by sending $[\![v]\!]$ to each other.
4. P_0 and P_1 run authentication check on reconstructed value:
   $$\mathsf{P}_0.\mathsf{Auth}([\![\gamma]\!], [(v, [\![\gamma v]\!])]) \leftrightarrow \mathsf{P}_1.\mathsf{Auth}([\![\gamma]\!], [(v, [\![\gamma v]\!])])$$
5. Output $(\texttt{reconstruct}, \mathsf{id}, v)$.

$\mathsf{P}_0.\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3) \leftrightarrow \mathsf{P}_1.\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$:

1. Require $\mathsf{id}_1, \mathsf{id}_2 \in T$, $\mathsf{id}_3 \notin T$, and $L_{mul} \neq [\cdot]$.
2. P_0 and P_1 pop $(\langle a\rangle, \langle b\rangle, \langle c\rangle) \leftarrow L_{mul}$.
3. P_0 and P_1 with $\langle v_1\rangle \leftarrow T[\mathsf{id}_1]$ and $\langle v_2\rangle \leftarrow T[\mathsf{id}_2]$ compute:
   $$\langle d\rangle \leftarrow \langle v_1\rangle - \langle a\rangle \quad \langle e\rangle \leftarrow \langle v_2\rangle - \langle b\rangle$$
4. P_0 and P_1 open d and e by sending $[\![d]\!]$ and $[\![e]\!]$ to each other.
5. P_0 and P_1 compute:
   $$\langle v_3\rangle \leftarrow \langle c\rangle + d \cdot \langle b\rangle + e \cdot \langle a\rangle + d \cdot e$$
6. P_0 and P_1 store $T[\mathsf{id}_3] \leftarrow \langle v_3\rangle$.
7. P_0 and P_1 push $(d, [\![\gamma d]\!])$ and $(e, [\![\gamma e]\!])$ to L_{auth} for authentication check.
8. Output $(\texttt{mul}, \mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$.

Figure 11: SPDZ [DPSZ12, DKL+13] protocol for secure computation of arithmetic circuits over $\mathbb{F}$.

Functionality $\mathsf{F}_{\mathsf{SPDZPrep}}[\mathbb{F}]$

$\mathsf{Main}(n_{i,0}, n_{i,1}, n_m)$ from P_b:

1. If no pending message from P_{1-b}, set $(n_{i,0}, n_{i,1}, n_m)$ from P_b as pending and return. Else parse $(n'_{i,0}, n'_{i,1}, n'_m)$ from P_{1-b}.
2. If mismatch, $n_{i,b} \neq n'_{i,b}$ or $n_m \neq n'_m$, send `abort` to all parties.
3. Sample shares of random authentication key $[\![\gamma]\!]$:
   (a) Sample $\gamma \leftarrow\!\!\$\ \mathbb{F}$
   (b) $[\![\gamma]\!]_0 \leftarrow\!\!\$\ \mathbb{F}$
   (c) $[\![\gamma]\!]_1 \leftarrow \gamma - [\![\gamma]\!]_0$
4. Compute input tuples for each $\mathsf{P}_{b'}$:
   - $L_{inp,b',b'} \leftarrow \left[\left(r_{i,b'}, \langle r_{i,b'}\rangle_{b'} = ([\![r_{i,b'}]\!]_{b'}, [\![\gamma r_{i,b'}]\!]_{b'})\right)\right]_i^{n_{i,b'}}$
   - $L_{inp,1-b',b'} \leftarrow \left[\left(\langle r_{i,b'}\rangle_{1-b'} = ([\![r_{i,b'}]\!]_{1-b'}, [\![\gamma r_{i,b'}]\!]_{1-b'})\right)\right]_i^{n_{i,b'}}$

   (a) Sample $r_{i,b'} \leftarrow\!\!\$\ \mathbb{F}$
   (b) $[\![r_{i,b'}]\!]_{b'} \leftarrow\!\!\$\ \mathbb{F}$, $[\![r_{i,b'}]\!]_{1-b'} \leftarrow r_{i,b'} - [\![r_{i,b'}]\!]_{b'}$
   (c) $[\![\gamma r_{i,b'}]\!]_{b'} \leftarrow\!\!\$\ \mathbb{F}$, $[\![\gamma r_{i,b'}]\!]_{1-b'} \leftarrow \gamma r_{i,b'} - [\![\gamma r_{i,b'}]\!]_{b'}$
5. Compute multiplication tuples for each $\mathsf{P}_{b'}$:
   - $L_{mul,b'} \leftarrow [(\langle a_i\rangle_{b'}, \langle b_i\rangle_{b'}, \langle c_i\rangle_{b'})]_i^{n_m}$

   (a) Sample $a_i, b_i \leftarrow\!\!\$\ \mathbb{F}$ and set $c_i \leftarrow a_i \cdot b_i$
   (b) $\langle a_i\rangle_{b'} \leftarrow\!\!\$\ \mathbb{F}^2$, $\langle a_i\rangle_{1-b'} \leftarrow (a_i - [\![a_i]\!]_{b'}, \gamma a_i - [\![\gamma a_i]\!]_{b'})$
   (c) $\langle b_i\rangle_{b'} \leftarrow\!\!\$\ \mathbb{F}^2$, $\langle b_i\rangle_{1-b'} \leftarrow (b_i - [\![b_i]\!]_{b'}, \gamma b_i - [\![\gamma b_i]\!]_{b'})$
   (d) $\langle c_i\rangle_{b'} \leftarrow\!\!\$\ \mathbb{F}^2$, $\langle c_i\rangle_{1-b'} \leftarrow (c_i - [\![c_i]\!]_{b'}, \gamma c_i - [\![\gamma c_i]\!]_{b'})$
6. Send
   - $([\![\gamma]\!]_0, L_{inp,0,0}, L_{inp,0,1}, L_{mul,0})$ to P_0.
   - $([\![\gamma]\!]_1, L_{inp,1,0}, L_{inp,1,1}, L_{mul,1})$ to P_1.

$\mathsf{P}_0.\mathsf{Auth}([\![\gamma]\!]_0, [(z_i, [\![\gamma z_i]\!]_0)]_i^n) \leftrightarrow \mathsf{P}_1.\mathsf{Auth}([\![\gamma]\!]_1, [(z_i, [\![\gamma z_i]\!]_1)]_i^n)$

1. P_0 and P_1 compute linear combination of inputs: $\ell, [\![\gamma\ell]\!]$
   (a) P_0 and P_1 each sample $r_b \leftarrow\!\!\$\ \mathbb{F}$.
   (b) P_0 and P_1 each commit to r_b by invoking $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Comm}$.
   (c) P_0 and P_1 open commitments to receive r_{1-b} from each other via $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Open}$.
   (d) P_0 and P_1 compute $r \leftarrow r_0 + r_1$.
   (e) P_0 and P_1 compute linear combination:
   $$\ell \leftarrow \sum_{i=1}^{n} r^{i-1} \cdot z_i \qquad [\![\gamma\ell]\!] \leftarrow \sum_{i=1}^{n} r^{i-1} \cdot [\![\gamma z_i]\!]$$
2. P_0 and P_1 check $[\![\gamma\ell]\!]$ against ℓ:
   (a) P_0 and P_1 compute $[\![\epsilon]\!] \leftarrow [\![\gamma\ell]\!] - [\![\gamma]\!] \cdot \ell$.
   (b) P_0 and P_1 each commit to $[\![\epsilon]\!]_b$ by invoking $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Comm}$.
   (c) P_0 and P_1 open commitments to receive $[\![\epsilon]\!]_{1-b}$ from each other via $\mathsf{F}_{\mathsf{Comm}}.\mathsf{Open}$.
   (d) P_0 and P_1 output `abort` if:
   $$[\![\epsilon]\!]_0 + [\![\epsilon]\!]_1 \neq 0$$

Figure 12: (Left) Functionality of preprocessing needed for the SPDZ [DPSZ12, DKL+13] protocol for secure computation of arithmetic circuits over $\mathbb{F}$. (Right) The authentication check subprotocol used in SPDZ for checking opened data values against authentication shares, in the $\mathsf{F}_{\mathsf{Comm}}$-hybrid model.

Functionality $\mathsf{F}_{\mathsf{ArithPermCirc}}$ [$\mathbb{F}$]

$\mathsf{Init}(\lambda)$:

1. $T \leftarrow \{\cdot\}$: maps id to values in $\mathbb{F}$.
2. $T_{\pi,0}, T_{\pi,1} \leftarrow \{\cdot\}$: maps id to permutations.
3. $ictr_0 \leftarrow 0$; $ictr_1 \leftarrow 0$: input operation counter.
4. $mctr \leftarrow 0$: multiplication operation counter.
5. $prep \leftarrow 0$: preprocessing flag.

$\mathsf{Prep}(n_{i,0}, n_{i,1}, n_m, \mathsf{in}_\pi)$ from P_b:

1. Parse in_π:
   - $[(\mathsf{id}_{\pi,b,i}, n_{\pi,b,i}, n_{\pi,inv,b,i}, \pi_{b,i} \in \mathsf{Perm}(m_{b,i}))]_i^{n_{\pi,b}}$
   - $[(\mathsf{id}_{\pi,1-b,i}, n_{\pi,1-b,i}, n_{\pi,inv,1-b,i})]_i^{n_{\pi,1-b}}$
2. Require $prep = 0$, preprocessing only called once.
3. If no pending message from P_{1-b}, set $(\texttt{prep}, n_{i,0}, n_{i,1}, n_m, \mathsf{in}_\pi)$ from P_b as pending and return. Else parse $(\mathsf{op}, n'_{i,0}, n'_{i,1}, n'_m, \mathsf{in}'_\pi)$ from P_{1-b} and require $\mathsf{op} = \texttt{prep}$.
4. If mismatch, $\exists b'$, $n_{i,b'} \neq n'_{i,b'}$, $n_m \neq n'_m$, $\exists\ i, b'$, $\mathsf{id}_{\pi,b',i} \neq \mathsf{id}'_{\pi,b',i}$, $n_{\pi,b',i} \neq n'_{\pi,b',i}$, $n_{\pi,inv,b',i} \neq n'_{\pi,inv,b',i}$, send $\texttt{abort}$ to all parties.
5. Else,
   - $ictr_b \leftarrow n_{i,b}$ and $mctr \leftarrow n_m$, $prep \leftarrow 1$.
   - $\forall i \in [n_{\pi,b'}]$ $T_{\pi,b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left(n_{\pi,b',i}, n_{\pi,inv,b',i}, \pi_{b',i}, m_{b',i}\right)$
6. Send $(\texttt{prep})$ to all parties.

$\mathsf{Input}(\mathsf{id}, v \in \mathbb{F})$ from P_b

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see Fig. 10).

$\mathsf{Reconstruct}(\mathsf{id})$ from P_b:

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see Fig. 10).

$\mathsf{ScalAdd}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$ from P_b

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see Fig. 10).

$\mathsf{Add}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from P_b

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see Fig. 10).

$\mathsf{ScalMul}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$ from P_b

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see Fig. 10).

$\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ from P_b

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see Fig. 10).

$\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ from P_b:

1. Require $\forall i\ \mathsf{id}_i \in T, \widetilde{\mathsf{id}}_i \notin T$ and no pending message from P_b.
2. Require $\mathsf{id}_\pi \in T_{\pi,b'}$ and $(ctr, invctr, \pi, m_\pi) \leftarrow T_{\pi,b'}[\mathsf{id}_\pi]$; $ctr > 0, m = m_\pi$.
3. If no pending message from P_{1-b}, set $(\texttt{perm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ from P_b as pending and return. Else parse $(\mathsf{op}, [\mathsf{id}'_i]_i^m, [\widetilde{\mathsf{id}}'_i]_i^m, \mathsf{id}'_\pi, b'')$ from P_{1-b} and require $\mathsf{op} = \texttt{perm}$.
4. If mismatch, $\exists i$, $\mathsf{id}_i \neq \mathsf{id}'_i, \widetilde{\mathsf{id}}_i \neq \widetilde{\mathsf{id}}'_i$, $\mathsf{id}_\pi \neq \mathsf{id}'_\pi$, then store $\forall\ i \in [m]$, $T[\widetilde{\mathsf{id}}_i], T[\widetilde{\mathsf{id}}'_i] \leftarrow \perp$.
5. Else apply permutation, store $\forall\ i \in [m]$, $T[\widetilde{\mathsf{id}}_{\pi(i)}] \leftarrow T[\mathsf{id}_i]$.
6. Decrement $ctr \leftarrow ctr - 1$.
7. Send
   - $(\texttt{perm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ to P_b.
   - $(\texttt{perm}, [\mathsf{id}'_i]_i^m, [\widetilde{\mathsf{id}}'_i]_i^m, \mathsf{id}'_\pi, b'')$ to P_{1-b}.

$\mathsf{PermuteInv}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ from P_b:

1. Require $\forall i\ \mathsf{id}_i \in T, \widetilde{\mathsf{id}}_i \notin T$ and no pending message from P_b.
2. Require $\mathsf{id}_\pi \in T_{\pi,b'}$ and $(ctr, invctr, \pi, m_\pi) \leftarrow T_{\pi,b'}[\mathsf{id}_\pi]$; $invctr > 0, m = m_\pi$.
3. If no pending message from P_{1-b}, set $(\texttt{invperm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ from P_b as pending and return. Else parse $(\mathsf{op}, [\mathsf{id}'_i]_i^m, [\widetilde{\mathsf{id}}'_i]_i^m, \mathsf{id}'_\pi, b'')$ from P_{1-b} and require $\mathsf{op} = \texttt{invperm}$.
4. If mismatch, $\exists i$, $\mathsf{id}_i \neq \mathsf{id}'_i, \widetilde{\mathsf{id}}_i \neq \widetilde{\mathsf{id}}'_i$, $\mathsf{id}_\pi \neq \mathsf{id}'_\pi$, then store $\forall\ i \in [m]$, $T[\widetilde{\mathsf{id}}_i], T[\widetilde{\mathsf{id}}'_i] \leftarrow \perp$.
5. Else, store $\forall\ i \in [m]$, $T[\widetilde{\mathsf{id}}_{\pi^{-1}(i)}] \leftarrow T[\mathsf{id}_i]$.
6. Decrement $invctr \leftarrow invctr - 1$.
7. Send
   - $(\texttt{invperm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ to P_b.
   - $(\texttt{invperm}, [\mathsf{id}'_i]_i^m, [\widetilde{\mathsf{id}}'_i]_i^m, \mathsf{id}'_\pi, b'')$ to P_{1-b}.

Figure 13: Functionality for computation of arithmetic circuits with additional permutation and inverse permutation gates over $\mathbb{F}$.

Protocol $\mathsf{FLOSS}[\mathbb{F}, \mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{ShufTuplePrep}}, \mathsf{F}_{\mathsf{Comm}}]$

$\mathsf{P}_b.\mathsf{Init}(\lambda)$:

1. $T \leftarrow \{\cdot\}$: maps id to shares in $\mathbb{F}$.
2. $L_{inp,0}, L_{inp,1} \leftarrow [\cdot]$: list of preprocessed input tuples.
3. $L_{mul} \leftarrow [\cdot]$: list of preprocessed multiplication tuples.
4. $T_{shf,0}, T_{shf,1} \leftarrow \{\cdot\}$: preprocessed shuffle tuples.
5. $T_{invshf,0}, T_{invshf,1} \leftarrow \{\cdot\}$: preprocessed inverse shuffle tuples.
6. $L_{auth} \leftarrow [\cdot]$: list of pending authentication checks.
7. $L_{bauth} \leftarrow [\cdot]$: list of pending blind authentication checks.
8. $[\![\gamma]\!]_b \leftarrow \perp$: share of authentication key.

$\mathsf{P}_0.\mathsf{Prep}(n_{inp,0}, n_{inp,1}, n_m, \mathsf{in}_\pi) \leftrightarrow \mathsf{P}_1.\mathsf{Prep}(n_{inp,0}, n_{inp,1}, n_m, \mathsf{in}_\pi)$:

1. P_0 and P_1 invoke $\mathsf{F}_{\mathsf{SPDZPrep}}(n_{inp,0}, n_{inp,1}, n_m)$ for each $\mathsf{P}_{b'}$ to receive (see Fig. 12):
   - $L_{inp,b'} \leftarrow \left[\left(r_{i,b'}, \langle r_{i,b'}\rangle_{b'}\right)\right]_i^{n_{inp,b'}}$
   - $L_{inp,1-b'} \leftarrow \left[\left(\langle r_{i,1-b'}\rangle_{b'}\right)\right]_i^{n_{inp,1-b'}}$
   - $L_{mul} \leftarrow [(\langle a_i\rangle_{b'}, \langle b_i\rangle_{b'}, \langle c_i\rangle_{b'})]_i^{n_m}$
   - $[\![\gamma]\!]_{b'}$
2. P_0 and P_1 as $\mathsf{P}_{b'}$ sample random shuffle tuple keys $\kappa_{i,j} \leftarrow\!\!\$\ \mathbb{F}$ and inverse shuffle tuple keys $\underline{\kappa}_{i,j} \leftarrow\!\!\$\ \mathbb{F}$ for each preprocessed shuffle of $\mathsf{P}_{b'}$ and random blinding vectors $\vec{a}_{i,j}, \vec{b}_{i,j} \leftarrow\!\!\$\ \mathbb{F}^{m_{1-b',i}}$ for each preprocessed shuffle of $\mathsf{P}_{1-b'}$, incorporate into input in_π, and then invoke $\mathsf{F}_{\mathsf{ShufTuplePrep}}(\mathsf{in}_\pi)$ for each $\mathsf{P}_{b'}$ to receive (see Fig. 15):
   - $T_{shf,b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i, \kappa_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$
   - $T_{shf,1-b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{a}_{i,j}, \vec{b}_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,1-b',i}}$
   - $T_{invshf,b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i^{-1}, \underline{\kappa}_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}\right)\right]_j^{n_{\pi,b',i}}$
   - $T_{invshf,1-b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{\underline{a}}_{i,j}, \vec{\underline{b}}_{i,j}, \langle\vec{\widetilde{\underline{a}}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}, \langle\vec{\widetilde{\underline{b}}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}\right)\right]_j^{n_{\pi,1-b',i}}$
3. Output $(\mathtt{prep})$.

$\mathsf{P}_b.\mathsf{Input}(\mathsf{id}, v \in \mathbb{F}) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{RcvInput}(\mathsf{id})$:

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see Fig. 11).

$\mathsf{P}_b.\mathsf{ScalAdd}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$:

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see Fig. 11).

$\mathsf{P}_b.\mathsf{Add}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$:

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see Fig. 11).

$\mathsf{P}_b.\mathsf{ScalMul}(\mathsf{id}_1, \alpha \in \mathbb{F}, \mathsf{id}_2)$:

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see Fig. 11).

$\mathsf{P}_0.\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3) \leftrightarrow \mathsf{P}_1.\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$:

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see Fig. 11).

$\mathsf{P}_0.\mathsf{Reconstruct}(\mathsf{id}) \leftrightarrow \mathsf{P}_1.\mathsf{Reconstruct}(\mathsf{id})$:

1. Require $\mathsf{id} \in T$.
2. P_0 and P_1 run authentication check on all pending multiplication opens in L_{auth} (see Fig. 12):
   (a) $[(z_i, [\![\gamma z_i]\!])]_i \leftarrow L_{auth}$; $L_{auth} \leftarrow [\cdot]$
   (b) $\mathsf{P}_0.\mathsf{Auth}([\![\gamma]\!], [(z_i, [\![\gamma z_i]\!])]_i) \leftrightarrow \mathsf{P}_1.\mathsf{Auth}([\![\gamma]\!], [(z_i, [\![\gamma z_i]\!])]_i)$
3. P_0 and P_1 run blind authentication check on all pending shuffle checks in L_{bauth} (see Fig. 2):
   (a) $\langle\alpha\rangle \leftarrow \mathsf{P}_b.\mathsf{Coin}(1) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Coin}(1)$
   (b) $[\langle z_i\rangle]_i \leftarrow L_{bauth}$; $L_{bauth} \leftarrow [\cdot]$
   (c) $\mathsf{P}_0.\mathsf{BlindAuth}([\![\gamma]\!], \langle\alpha\rangle, [\langle z_i\rangle_0]_i)$
   $\leftrightarrow$ $\mathsf{P}_1.\mathsf{BlindAuth}([\![\gamma]\!], \langle\alpha\rangle, [\langle z_i\rangle_1]_i)$
4. P_0 and P_1 open $\langle v\rangle \leftarrow T[\mathsf{id}]$ to v by sending $[\![v]\!]$ to each other.
5. P_0 and P_1 run authentication check on reconstructed value (Fig. 12):
   $\mathsf{P}_0.\mathsf{Auth}([\![\gamma]\!], [(v, [\![\gamma v]\!])]) \leftrightarrow \mathsf{P}_1.\mathsf{Auth}([\![\gamma]\!], [(v, [\![\gamma v]\!])])$
6. Output $(\mathtt{reconstruct}, \mathsf{id}, T[\mathsf{id}])$.

$\mathsf{P}_0.\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b') \leftrightarrow \mathsf{P}_1.\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$:

1. Require $\forall i\ \mathsf{id}_i \in T, \widetilde{\mathsf{id}}_i \notin T$, and $\mathsf{id}_\pi \in T_{shf,b'}$.
2. P_0 and P_1 retrieve shuffle tuples:
   - $\mathsf{P}_{b'}$ pops $\left(\pi, \kappa, \langle\vec{\widetilde{a}}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}\rangle_{b'}^{(\kappa)}\right) \leftarrow T_{shf,b'}[\mathsf{id}_\pi]$.
   - $\mathsf{P}_{1-b'}$ pops $\left(\vec{a}, \vec{b}, \langle\vec{\widetilde{a}}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}\rangle_{b'}^{(\kappa)}\right) \leftarrow T_{shf,b'}[\mathsf{id}_\pi]$.
3. P_0 and P_1 apply shuffle tuple to $\langle\vec{v}\rangle = [\langle v\rangle]_i^m \leftarrow T[[\mathsf{id}_i]_i^m]$ to get $\langle\vec{\widetilde{v}}\rangle$, $\langle\kappa\vec{\widetilde{v}}\rangle$ (Fig. 3):
   $\mathsf{P}_{b'}.\mathsf{ApplyShufTup}\left(\pi, \kappa, \langle\vec{\widetilde{a}}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}\rangle_{b'}^{(\kappa)}, \langle\vec{v}\rangle\right)$
   $\leftrightarrow \mathsf{P}_{1-b'}.\mathsf{ApplyShufTup}\left(\vec{a}, \vec{b}, \langle\vec{\widetilde{a}}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}\rangle_{b'}^{(\kappa)}, \langle\vec{v}\rangle\right)$
4. P_0 and P_1 store $T[[\widetilde{\mathsf{id}}_i]_i^m] \leftarrow \langle\vec{\widetilde{v}}\rangle$.
5. P_0 and P_1 push $\langle\kappa\widetilde{v}\rangle$ to L_{bauth} for blind authentication check.
6. Output $(\mathtt{perm})$.

$\mathsf{P}_b.\mathsf{PermuteInv}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$:

Same as Permute except P_0 and P_1 retrieve shuffle tuple from $T_{invshf,b'}[\mathsf{id}_\pi]$.

Figure 14: Our main construction FLOSS enabling secure computation of arithmetic circuits over $\mathbb{F}$ with special purpose efficient gates for permutations and inverse permutations.

Functionality $\mathsf{F}_{\mathsf{ShufTuplePrep}}[\mathbb{F}]$

$\mathsf{Main}(\mathsf{in}_\pi)$ from P_b:

1. Parse in_π:
   - $[(\mathsf{id}_{\pi,b,i}, n_{\pi,b,i}, n_{\pi,inv,b,i}, [\kappa_{i,j} \in \mathbb{F}]_j^{n_{\pi,b,i}}, [\underline{\kappa}_{i,j} \in \mathbb{F}]_j^{n_{\pi,inv,b,i}}, \pi_{b,i} \in \mathsf{Perm}(m_{b,i}))]_i^{n_{\pi,b}}$: Permutation parameters for permutations to be applied by P_b. P_b supplies the permutation and κ-authentication values, additionally specifying the number of forward and inverse applications to preprocess for each permutation.
   - $[(\mathsf{id}_{\pi,1-b,i}, n_{\pi,inv,1-b,i}, [\vec{a}_{i,j}, \vec{b}_{i,j} \in \mathbb{F}^{m_{1-b,i}}]_j^{n_{\pi,1-b,i}}, [\underline{\vec{a}}_{i,j}, \underline{\vec{b}}_{i,j} \in \mathbb{F}^{m_{1-b,i}}]_j^{n_{\pi,inv,1-b,i}},)]_i^{n_{\pi,1-b}}$: Permutation parameters for permutations to be applied by P_{1-b}. P_b does not hold the permutation but holds blinding vectors that will be used to construct shuffle tuples.
2. If no pending message from P_{1-b}, set $(\texttt{prep}, n_{i,0}, n_{i,1}, n_m, \mathsf{in}_\pi)$ from P_b as pending and return. Else parse $(\mathsf{op}, n'_{i,0}, n'_{i,1}, n'_m, \mathsf{in}'_\pi)$ from P_{1-b} and require $\mathsf{op} = \texttt{prep}$.
3. If mismatch, $\exists b'$, $n_{i,b'} \neq n'_{i,b'}$, $n_m \neq n'_m$, $\exists\, i, b'$, $\mathsf{id}_{\pi,b',i} \neq \mathsf{id}'_{\pi,b',i}$, $n_{\pi,b',i} \neq n'_{\pi,b',i}$, $n_{\pi,inv,b',i} \neq n'_{\pi,inv,b',i}$, send `abort` to all parties.
4. Compute shuffle tuples for each $b' \in \{b, 1-b\}$:
   - $\forall i \in [n_{\pi,b'}],\ T_{shf,b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i, \kappa_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$
   - $\forall i \in [n_{\pi,b'}],\ T_{shf,1-b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{a}_{i,j}, \vec{b}_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{1-b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{1-b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$

   (a) Take $\vec{a}_{i,j}, \vec{b}_{i,j}$ supplied by $\mathsf{P}_{1-b'}$ and permute to compute $\vec{\widetilde{a}}_{i,j} \leftarrow \pi_{b',i}(\vec{a}_{i,j})$ and $\vec{\widetilde{b}}_{i,j} \leftarrow \pi_{b',i}(\vec{b}_{i,j})$.

   (b) Create κ-authenticated shares of $\vec{\widetilde{a}}_{i,j}$ and $\vec{\widetilde{b}}_{i,j}$:
   - $\langle\vec{\widetilde{a}}_{i,j}\rangle_{b'} \leftarrow\!\!\$\ (\mathbb{F}^2)^{m_{1-b,i}}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{1-b'} \leftarrow (\vec{\widetilde{a}}_{i,j} - [\![\vec{\widetilde{a}}_{i,j}]\!]_{b'}, \kappa_{i,j}\vec{\widetilde{a}}_{i,j} - [\![\kappa_{i,j}\vec{\widetilde{a}}_{i,j}]\!]_{b'})$
   - $\langle\vec{\widetilde{b}}_{i,j}\rangle_{b'} \leftarrow\!\!\$\ (\mathbb{F}^2)^{m_{1-b,i}}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{1-b'} \leftarrow (\vec{\widetilde{b}}_{i,j} - [\![\vec{\widetilde{b}}_{i,j}]\!]_{b'}, \kappa_{i,j}\vec{\widetilde{b}}_{i,j} - [\![\kappa_{i,j}\vec{\widetilde{b}}_{i,j}]\!]_{b'})$
5. Compute inverse shuffle tuples for each $\mathsf{P}_{b'}$:
   - $\forall i \in [n_{\pi,b'}],\ T_{invshf,b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i^{-1}, \underline{\kappa}_{i,j}, \langle\vec{\widetilde{\underline{a}}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}, \langle\vec{\widetilde{\underline{b}}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}\right)\right]_j^{n_{\pi,b',i}}$
   - $\forall i \in [n_{\pi,b'}],\ T_{invshf,1-b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{\underline{a}}_{i,j}, \vec{\underline{b}}_{i,j}, \langle\vec{\widetilde{\underline{a}}}_{i,j}\rangle_{1-b'}^{(\underline{\kappa})}, \langle\vec{\widetilde{\underline{b}}}_{i,j}\rangle_{1-b'}^{(\underline{\kappa})}\right)\right]_j^{n_{\pi,b',i}}$

   (a) Take $\vec{\underline{a}}_{i,j}, \vec{\underline{b}}_{i,j}$ supplied by $\mathsf{P}_{1-b'}$ and permute to compute $\vec{\widetilde{\underline{a}}}_{i,j} \leftarrow \pi_{b',i}(\vec{\underline{a}}_{i,j})$ and $\vec{\widetilde{\underline{b}}}_{i,j} \leftarrow \pi_{b',i}(\vec{\underline{b}}_{i,j})$.

   (b) Create $\underline{\kappa}$-authenticated shares of $\vec{\widetilde{\underline{a}}}_{i,j}$ and $\vec{\widetilde{\underline{b}}}_{i,j}$:
   - $\langle\vec{\widetilde{\underline{a}}}_{i,j}\rangle_{b'} \leftarrow\!\!\$\ (\mathbb{F}^2)^{m_{1-b,i}}, \langle\vec{\widetilde{\underline{a}}}_{i,j}\rangle_{1-b'} \leftarrow (\vec{\widetilde{\underline{a}}}_{i,j} - [\![\vec{\widetilde{\underline{a}}}_{i,j}]\!]_{b'}, \underline{\kappa}_{i,j}\vec{\widetilde{\underline{a}}}_{i,j} - [\![\underline{\kappa}_{i,j}\vec{\widetilde{\underline{a}}}_{i,j}]\!]_{b'})$
   - $\langle\vec{\widetilde{\underline{b}}}_{i,j}\rangle_{b'} \leftarrow\!\!\$\ (\mathbb{F}^2)^{m_{1-b,i}}, \langle\vec{\widetilde{\underline{b}}}_{i,j}\rangle_{1-b'} \leftarrow (\vec{\widetilde{\underline{b}}}_{i,j} - [\![\vec{\widetilde{\underline{b}}}_{i,j}]\!]_{b'}, \underline{\kappa}_{i,j}\vec{\widetilde{\underline{b}}}_{i,j} - [\![\underline{\kappa}_{i,j}\vec{\widetilde{\underline{b}}}_{i,j}]\!]_{b'})$
6. Send
   - $(T_{shf,0,0}, T_{shf,0,1}, T_{invshf,0,0}, T_{invshf,0,1})$ to P_0.
   - $(T_{shf,1,0}, T_{shf,1,1}, T_{invshf,1,0}, T_{invshf,1,1})$ to P_1.

Figure 15: Functionality of preprocessing of authenticated shuffle tuples consumed in FLOSS for fast online permutations.

Proof. Given an adversary A in the real world, we explicitly construct a simulator S in the ideal world in Figure 16. Without loss of generality, we assume that party P_0 is corrupted, i.e., $b = 0$; the case where party P_1 is corrupted is symmetric. The simulator S receives the input in_A from the environment and internally executes A on this input. Whenever A attempts to invoke an operation via the control function C, the simulator intercepts that invocation and acts as described in Figure 16. Whenever A sends a message to the environment, the simulator forwards it to the environment. Finally, S outputs the output of A.

Without loss of generality, we assume that all messages from the adversary parse correctly and do not exceed the available preprocessing material (i.e., the number of input tuples, Beaver triples, and shuffle tuples allocated during Prep). If any message fails to parse or exceeds the available preprocessing, the simulator returns an abort signal to the adversary, which is consistent with the honest party's behavior in the real protocol. Similarly, we assume that commitments from the adversary verify correctly, else the simulator implicitly aborts, and that the adversary does not break the security of the commitment scheme (working in the F_Comm-hybrid model).

Throughout the protocol execution, the simulator maintains for each wire id the *expected adversary shares* $\mathsf{expShare}[\mathsf{id}] = (\llbracket z \rrbracket_0, \llbracket \gamma z \rrbracket_0)$. At each gate, the simulator computes $\mathsf{expShare}$ for the output wire by following what the adversary's shares of the output wire would be if the adversary acted honestly in this gate, given the adversary's prior messages. This can be done by the simulator since it knows the preprocessing shares (known from simulating the $(\mathsf{F}_\mathsf{SPDZPrep}, \mathsf{F}_\mathsf{ShufTuplePrep})$-hybrid model), the adversary's protocol messages, and the simulator's own randomness. When the adversary reveals a share, we write $\llbracket z \rrbracket_0^{\mathsf{real}}$ for the received value. The error $\delta_z = \llbracket z \rrbracket_0^{\mathsf{real}} - \llbracket z \rrbracket_0^{\mathsf{exp}}$ captures only a *new* deviation introduced at this message, but not prior deviations that have affected prior opened values (even if they were used to construct this share).

We now state two lemmas bounding the probability that the adversary can deviate without detection, which are essential to proving indistinguishability between the ideal and real worlds. These lemmas are information-theoretic and hold against unbounded adversaries.

Lemma 3 (Non-Blind MAC Check Security)**.** *Fix any $n \geq 1$ and any values $z_1, \ldots, z_n \in \mathbb{F}$ with corresponding shares $\{(\llbracket z_i \rrbracket_0, \llbracket z_i \rrbracket_1)\}_{i=1}^n$. Sample $\gamma \leftarrow\!\!\$\, \mathbb{F}$ and $\llbracket \gamma \rrbracket_0 \leftarrow\!\!\$\, \mathbb{F}$, and compute MAC shares $\llbracket \gamma z_i \rrbracket_0, \llbracket \gamma z_i \rrbracket_1$ satisfying $\llbracket \gamma z_i \rrbracket_0 + \llbracket \gamma z_i \rrbracket_1 = \gamma z_i$. Consider the following game with an (unbounded) adversary A:*

1. *A receives $\llbracket \gamma \rrbracket_0$ and $\{(\llbracket z_i \rrbracket_0, \llbracket \gamma z_i \rrbracket_0)\}_{i=1}^n$.*
2. *A outputs (possibly modified) value shares $\{\llbracket z_i \rrbracket_0'\}_{i=1}^n$.*
3. *A participates in the non-blind authentication check (Figure 12) in the F_Comm-hybrid model, with an honest party holding $\llbracket \gamma \rrbracket_1$ and $\{(\llbracket z_i \rrbracket_1, \llbracket \gamma z_i \rrbracket_1)\}_{i=1}^n$, using the opened values $z_i' = \llbracket z_i \rrbracket_0' + \llbracket z_i \rrbracket_1$.*

The adversary wins if $\exists i \in [n] : \llbracket z_i \rrbracket_0' \neq \llbracket z_i \rrbracket_0$ and the check passes. Then $\Pr[\mathsf{A}\ \textit{wins}] \leq n/|\mathbb{F}|$.

Lemma 4 (Blind MAC Check Security)**.** *Fix any $n \geq 1$ and any values $z_1, \ldots, z_n \in \mathbb{F}$ with corresponding shares $\{(\llbracket z_i \rrbracket_0, \llbracket z_i \rrbracket_1)\}_{i=1}^n$. Sample $\gamma \leftarrow\!\!\$\, \mathbb{F}$ and $\llbracket \gamma \rrbracket_0 \leftarrow\!\!\$\, \mathbb{F}$, and compute MAC shares $\llbracket \gamma z_i \rrbracket_0, \llbracket \gamma z_i \rrbracket_1$ satisfying $\llbracket \gamma z_i \rrbracket_0 + \llbracket \gamma z_i \rrbracket_1 = \gamma z_i$. Similarly, fix a blinding value $\alpha \in \mathbb{F}$ with shares $(\llbracket \alpha \rrbracket_0, \llbracket \alpha \rrbracket_1)$ and MAC shares satisfying $\llbracket \gamma\alpha \rrbracket_0 + \llbracket \gamma\alpha \rrbracket_1 = \gamma\alpha$. Consider the following game with an (unbounded) adversary A:*

1. *A receives $\llbracket \gamma \rrbracket_0$, $(\llbracket \alpha \rrbracket_0, \llbracket \gamma\alpha \rrbracket_0)$, and $\{(\llbracket z_i \rrbracket_0, \llbracket \gamma z_i \rrbracket_0)\}_{i=1}^n$.*
2. *Sample $r \leftarrow\!\!\$\, \mathbb{F}$ and give r to A.*
3. *Compute the honest shares $\llbracket \ell \rrbracket_1 \leftarrow \llbracket \alpha \rrbracket_1 + \sum_{i=1}^n r^{i-1} \llbracket z_i \rrbracket_1$ and $\llbracket \gamma\ell \rrbracket_1 \leftarrow \llbracket \gamma\alpha \rrbracket_1 + \sum_{i=1}^n r^{i-1} \llbracket \gamma z_i \rrbracket_1$, as well as the adversary expected share $\llbracket \ell \rrbracket_0^{\mathsf{exp}} \leftarrow \llbracket \alpha \rrbracket_0 + \sum_{i=1}^n r^{i-1} \llbracket z_i \rrbracket_0$.*
4. *Obtain $(\ell_0', \llbracket \epsilon \rrbracket_0')$ from A.*
5. *Compute $\ell' = \ell_0' + \llbracket \ell \rrbracket_1$ and $\llbracket \epsilon \rrbracket_1 = \llbracket \gamma\ell \rrbracket_1 - \llbracket \gamma \rrbracket_1 \cdot \ell'$.*
6. *The check passes if $\llbracket \epsilon \rrbracket_0' + \llbracket \epsilon \rrbracket_1 = 0$.*

The adversary wins if $\ell_0' \neq \llbracket \ell \rrbracket_0^{\mathsf{exp}}$ and the check passes. Then $\Pr[\mathsf{A}\ \textit{wins}] = 1/|\mathbb{F}|$.

The proofs of these lemmas appear after the main proof. We now use them to establish indistinguishability using the following steps. First, we show that all messages from the simulator to the adversary are distributed identically to the messages from the real protocol, conditioned on a bad event not happening, and we show $\Pr[\mathsf{bad}]$ is negligible. Next, we show that any deviation from the protocol by the adversary causes an authentication check to fail before the next reconstruction, except with negligible

probability. Finally, we show that conditioned on all deviations being caught, the real and the ideal world behave identically in the view of the environment.

Correct distribution of the adversary's view. We verify that each message sent by the simulator to the adversary has the same distribution as in the real protocol, except on very explicitly stated bad events that happen with negligible probability. Consequently, with high probability the adversary's view is identically distributed in both worlds, which implies that any message that it sends to the environment is identically distributed.

Preprocessing shares. The simulator samples all material (input tuples, Beaver triples, shuffle tuples) by following the protocol of the (idealized) hybrid functionality.

Input phase. In the ideal world when the honest party provides input, the simulator sends a uniformly random $\epsilon \leftarrow\!\!\$\, \mathbb{F}$. In the real world, the honest party sends $\epsilon = x - s$ where s is the input tuple mask. Since s is uniform and independent of the adversary's view, ϵ is uniformly random in both worlds.

Linear operations. For addition, scalar addition, and scalar multiplication, no messages are exchanged between the parties.

Multiplication. In the ideal world the simulator sends $(d_1, e_1) \leftarrow\!\!\$\, \mathbb{F}^2$. In the real protocol, $d_1 = (x - a) - d_0$ where a is the Beaver triple mask. Since a is uniform and independent of the adversary's view, d_1 is uniform. Similar logic applies for e_1.

Permutation. When the adversary applies π (case $b' = 0$), the simulator sends random blinding vectors $(c, d) \leftarrow\!\!\$\, \mathbb{F}^m \times \mathbb{F}^m$, exactly as the real protocol specifies. When the honest party applies π (case $b' = 1$), the adversary does not receive a message.

Reconstruction. The only non-trivial simulation occurs during reconstruction which involves the authentication checks, where the simulation must account for the adversary's possible deviations from earlier in the protocol. We analyze it part by part.

1. *Non-blind authentication check.* The checked values $z_1, \ldots, z_n$ are the opened $d = x - a$ and $e = y - b$ from multiplications, and recall that $\delta_i = [\![z_i]\!]_0^{\mathsf{real}} - [\![z_i]\!]_0^{\mathsf{exp}}$ is defined as the error introduced by the adversary in the reconstruction of each value.

   After r is determined, which happens in the same way in the real and the ideal world, define $\ell = \sum_i r^{i-1} z_i$, $\Delta = \sum_i r^{i-1} \delta_i$, and $[\![\epsilon]\!]_0^{\mathsf{exp}} = [\![\gamma\ell]\!]_0^{\mathsf{exp}} - [\![\gamma]\!]_0 \cdot \ell$. The simulator sets $[\![\epsilon]\!]_1 = -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma \cdot \Delta$.
   In the real protocol, the MAC shares satisfy $[\![\gamma z_i]\!]_0^{\mathsf{exp}} + [\![\gamma z_i]\!]_1 = \gamma(z_i - \delta_i)$, so $[\![\gamma z_i]\!]_1 = \gamma z_i - \gamma\delta_i - [\![\gamma z_i]\!]_0^{\mathsf{exp}}$. Thus, the honest party computes

   $$[\![\epsilon]\!]_1 = \sum_i r^{i-1} [\![\gamma z_i]\!]_1 - [\![\gamma]\!]_1 \ell = \gamma\ell - \gamma\Delta - [\![\gamma\ell]\!]_0^{\mathsf{exp}} - \gamma\ell + [\![\gamma]\!]_0 \ell = -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma \cdot \Delta,$$

   which equals the simulator's value.

2. *Blind authentication check.* The checked values $z_1, \ldots, z_n$ are permutation outputs, which are not opened individually. Instead, a blinding value $\alpha = \alpha_0 + \alpha_1$ is generated (each party inputs their contribution) and the linear combination $\ell = \alpha + \sum_i r^{i-1} z_i$ is opened.

   For the opening message $[\![\ell]\!]_1$: the simulator sends $[\![\ell]\!]_1 \leftarrow\!\!\$\, \mathbb{F}$. In the real protocol, $[\![\ell]\!]_1 = [\![\alpha_1]\!]_1 + [\![\alpha_0]\!]_1 + \sum_i r^{i-1} [\![z_i]\!]_1$. Since α_1 is the honest party's input with mask s_1, we have $[\![\alpha_1]\!]_1 = [\![s_1]\!]_1 = s_1 - [\![s_1]\!]_0$, which is uniform from the adversary's view. Thus $[\![\ell]\!]_1$ is uniform.

   For the authentication message $[\![\epsilon]\!]_1$: define $\Delta_\ell = \ell_0 - [\![\ell]\!]_0^{\mathsf{exp}}$ and $[\![\epsilon]\!]_0^{\mathsf{exp}} = [\![\gamma\ell]\!]_0^{\mathsf{exp}} - [\![\gamma]\!]_0 \ell$. We split into two cases.

   Case $\mathsf{permInputDev} = \mathsf{false}$: every $b' = 1$ gate satisfies $\vec{w} = (\vec{d} - \vec{d}^{\mathsf{exp}}) - \gamma(\vec{c} - \vec{c}^{\mathsf{exp}}) = \vec{0}$, so the γ-MAC relation holds for all blind auth check entries (MAC error $E_i = 0$). The simulator sets $[\![\epsilon]\!]_1 = -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma\Delta_\ell$. The same algebraic verification as in the non-blind case shows that the honest party's $[\![\epsilon]\!]_1$ in the real protocol equals the simulator's value.

   Case $\mathsf{permInputDev} = \mathsf{true}$: this is the more interesting case, where the adversary attempted to introduce an error into their input for one of the permutation gates. This is also where our proof would fall apart had our protocol permitted the permutation guessing selective failure attack (Section 3).

   In this case, there exists some shuffling gate where $b' = 1$ (the honest party knows the permutation) and $\vec{w} \neq \vec{0}$. The simulator sends $[\![\epsilon]\!]_1 \leftarrow\!\!\$\, \mathbb{F}$. We show that in the real world, $[\![\epsilon]\!]_1$ is also uniformly random from the adversary's view.

First, for simplicity, assume a single gate in the blind authentication check with m components. Let π denote the honest party's permutation and κ the honest party's multiplicative key. The adversary sends $(\vec{c}, \vec{d})$ with deviations $\vec{\delta}_c = \vec{c} - \vec{c}^{\mathsf{exp}}$ and $\vec{\delta}_d = \vec{d} - \vec{d}^{\mathsf{exp}}$, and recall $\vec{w} = \vec{\delta}_d - \gamma\vec{\delta}_c$.

Tracing through the real protocol (Fig. 3) with the deviated inputs, the κ-multiplied value shares reconstruct to $[\![\kappa\vec{\widetilde{v}}]\!]_0^{\mathsf{exp}} + [\![\kappa\vec{\widetilde{v}}]\!]_1 = \kappa\pi(\vec{v} + \vec{\delta}_c)$ and the γ-MAC on the κ-multiplied shares reconstructs to $[\![\gamma\kappa\vec{\widetilde{v}}]\!]_0^{\mathsf{exp}} + [\![\gamma\kappa\vec{\widetilde{v}}]\!]_1 = \gamma\kappa\pi(\vec{v}) + \kappa\pi(\vec{\delta}_d)$ (the adversary's expected output shares $[\![\cdot]\!]_0^{\mathsf{exp}}$ are set from preprocessing and are unaffected by the deviation).

The real world blind auth check processes m entries $\kappa\widetilde{v}_1, \ldots, \kappa\widetilde{v}_m$, and computes $[\![\epsilon]\!]_1 = [\![\gamma\ell]\!]_1 - [\![\gamma]\!]_1 \cdot \ell$ where $\ell = \alpha + \sum r^{i-1} z_i$. Each entry $\kappa\widetilde{v}_j$ has per-component MAC error $\kappa\pi(\vec{\delta}_d)_j - \gamma\kappa\pi(\vec{\delta}_c)_j = \kappa\pi(\vec{w})_j$. Using the same expansion as in the non-blind case:

$$[\![\epsilon]\!]_1 = -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma\Delta_\ell + \kappa \cdot \sum_{j=1}^{m} r^{j-1}\pi(\vec{w})_j.$$

Since $S := \sum_{j=1}^{m} r^{j-1}\pi(\vec{w})_j$ is a polynomial in r of degree at most $m-1$ with at least one nonzero coefficient, by the Schwartz-Zippel lemma this polynomial has at most $m-1$ roots, and so $\Pr_{r \leftarrow\$ \mathbb{F}}[S = 0] \leq (m-1)/|\mathbb{F}|$. We explicitly define $S = 0$ as a bad event, in which our simulation fails, but it happens with negligible probability. Conditioned on S being non-zero, $[\![\epsilon]\!]_1$ is an affine function of κ with nonzero coefficient, and since κ is uniformly random, so is $[\![\epsilon]\!]_1$.

We point out that this argument easily extends to multiple permutation gates, at the cost of notational complexity. The key point is that if there is any gate where $\vec{w} \neq \vec{0}$, then at least one of the resulting polynomials in r is non-zero, and thus the overall linear combination is non-zero except with negligible probability over r.

3. When opening v, the simulator sends $[\![v]\!]_1 = v - [\![v]\!]_0^{\mathsf{exp}}$ which is equal to the message in the real world since the simulator correctly tracks the expected adversary's share. Finally, for the authentication of the opened value the simulation is perfect since the same analysis as the non-blind authentication check applies (with $n = 1$).

Deviation detection. We bound the probability that the adversary deviates from the protocol without getting detected in the real world (and equivalently in the ideal world, since conditioned on bad not occurring the simulation is identical), that is the probability that the adversary introduces an error in any value it sends to the honest party without being caught by the subsequent authentication check. We enumerate all messages the adversary sends and, for each possible deviation, apply the appropriate MAC security lemma to bound the forgery probability.

Input from P_0. This step defines the adversary's input, which it can choose in both worlds, and against which we define consistency. No deviation is possible.

Multiplication. During multiplication, the adversary sends (d_0, e_0), possibly deviating from the expected $d_0^{\mathsf{exp}} = [\![x]\!]_0 - [\![a]\!]_0$, $e_0^{\mathsf{exp}} = [\![y]\!]_0 - [\![b]\!]_0$. This changes the opened values $d = d_0 + d_1$, $e = e_0 + e_1$, but does not change the MAC: the honest party's shares and MAC shares still authenticate the original (expected) opened values.

At $\mathsf{Reconstruct}$, the non-blind authentication check processes all n pending entries. Suppose some entry has error $\delta_i = [\![z_i]\!]_0^{\mathsf{real}} - [\![z_i]\!]_0^{\mathsf{exp}} \neq 0$ and the check passes. Then the adversary has successfully opened an incorrect value without failing authentication, which by Lemma 3 occurs with negligible probability.

Permutation. In $b' = 0$ gates the adversary sends no message, so no deviation is possible. In $b' = 1$ gates the adversary sends $(\vec{c}, \vec{d})$, possibly deviating from the expected values. Let $\vec{\delta}_c = \vec{c} - \vec{c}^{\mathsf{exp}}$, $\vec{\delta}_d = \vec{d} - \vec{d}^{\mathsf{exp}}$, and $\vec{w} = \vec{\delta}_d - \gamma\vec{\delta}_c$.

If $\vec{w} \neq \vec{0}$, then the correct distribution analysis shows that $[\![\epsilon]\!]_1$ in the blind auth check is uniformly random (conditioned on $\neg\mathsf{bad}$, which happens with overwhelming probability), so the check catches the deviation with probability $1 - 1/|\mathbb{F}|$.

If $\vec{w} = \vec{0}$ but $(\vec{\delta}_c, \vec{\delta}_d) \neq (\vec{0}, \vec{0})$, then the MAC relation is preserved, and the blind auth check passes. However, this requires the adversary to have guessed γ: for any nonzero $\vec{\delta}_c$, the constraint $\vec{\delta}_d = \gamma\vec{\delta}_c$ fixes γ to a single value, and since γ is information-theoretically hidden from the adversary's view, this occurs with probability at most $1/|\mathbb{F}|$.

Reconstruction. Three checks are invoked. The non-blind check on L_{auth} catches any additional error beyond the opened-value deviations analyzed above (Lemma 3). The blind check on L_{bauth} catches all newly introduced errors by Lemma 4 and the permutation analysis above. Finally, the adversary opens $[\![v]\!]_0'$, and the per-value non-blind check ($n = 1$) catches any deviation by Lemma 3.

Indistinguishability of the honest party's output. Conditioned on all deviations being caught, we argue that the real and ideal worlds are identical from the environment's view.

Recall that the environment receives communication from the adversary, along with the output of the honest party at the end of the protocol. Since the adversary's view is identically distributed in both worlds, any message it sends to the environment is identically distributed. We are left to argue that the honest party's output is identically distributed in both worlds.

Recall that the honest party's output is determined by individual reconstruction gates. In the ideal world, at each reconstruction gate the simulator forwards $\mathsf{Reconstruct}(\mathsf{id})$ to $\mathsf{F}_{\mathsf{ArithPermCirc}}$, receives the correct output v, and routes it to the honest party (or aborts if an authentication check fails).

In the real world, conditioned on no undetected deviations, if the protocol does not abort then the adversary did not deviate at any point, so all wire values are correctly computed and the honest party reconstructs $v' = [\![v]\!]'_0 + [\![v]\!]_1 = v$. Since the simulator extracts the same input $x_0 = r + \epsilon$ that the adversary uses in the real protocol, the output matches. The abort events are identically distributed by the correct distribution analysis (conditioned on bad not occurring).

Conclusion. Conditioned on bad not occurring and no deviation going undetected, the environment's complete view (the adversary's communication, the abort outcome, and the honest party's output) is identically distributed in both worlds. Since $\Pr[\mathsf{bad}]$ and the probability of an undetected deviation are both negligible in the security parameter (by the correct distribution and deviation detection analyses, respectively), the real and ideal worlds are indistinguishable except with negligible probability. □

We now prove the two lemmas used in the security proof.

Proof of Lemma 3. Let $\delta_i = [\![z_i]\!]'_0 - [\![z_i]\!]_0$ denote the adversary's deviation in the i-th value share. After opening, the reconstructed values are $z'_i = z_i + \delta_i$ and $\ell' = \sum_{i=1}^{n} r^{i-1} z'_i$, where r is a public random challenge.

The honest party computes $[\![\epsilon]\!]_1 = [\![\gamma\ell']\!]_1 - [\![\gamma]\!]_1 \cdot \ell' = -\gamma \cdot \Delta(r) - [\![\epsilon]\!]_0^{\mathsf{exp}}$, where $\Delta(r) = \sum_{i=1}^{n} r^{i-1}\delta_i$ and $[\![\epsilon]\!]_0^{\mathsf{exp}} = [\![\gamma\ell']\!]_0 - [\![\gamma]\!]_0 \cdot \ell'$.

The adversary then sends some $[\![\epsilon]\!]'_0$, and the check passes if and only if $[\![\epsilon]\!]'_0 + [\![\epsilon]\!]_1 = 0$. Thus, to pass the check, the adversary must send $[\![\epsilon]\!]'_0 = -[\![\epsilon]\!]_1 = \gamma \cdot \Delta(r) + [\![\epsilon]\!]_0^{\mathsf{exp}}$. Since the adversary can compute $[\![\epsilon]\!]_0^{\mathsf{exp}}$ from their view, passing the check allows the adversary to also learn $\gamma \cdot \Delta(r)$.

We bound the probability of a correct guess. By the winning condition, some $\delta_i \neq 0$, so $\Delta(X) = \sum_{i=1}^{n} X^{i-1}\delta_i$ is a nonzero polynomial of degree at most $n-1$. Define the event $E = \{\Delta(r) = 0\}$. By the Schwartz-Zippel lemma, $\Pr[E] \leq (n-1)/|\mathbb{F}|$.

Conditioned on E not happening, the value $\gamma \cdot \Delta(r)$ is uniformly distributed over $\mathbb{F}$, since $\gamma \leftarrow\!\!\$\, \mathbb{F}$ is independent of the adversary's view, and thus can be guessed with probability $1/|\mathbb{F}|$.

Therefore, $\Pr[\mathsf{A}\text{ wins}] \leq \Pr[E] + \Pr[\text{correct guess} \mid \neg E] \leq n/|\mathbb{F}|$. □

Proof of Lemma 4. Let $\Delta_\ell = \ell'_0 - [\![\ell]\!]_0^{\mathsf{exp}}$ denote the adversary's deviation from the expected share. The opened value is $\ell' = \ell'_0 + [\![\ell]\!]_1 = \ell + \Delta_\ell$.

The honest party computes $[\![\epsilon]\!]_1 = [\![\gamma\ell]\!]_1 - [\![\gamma]\!]_1 \cdot \ell' = -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma \cdot \Delta_\ell$, where $[\![\epsilon]\!]_0^{\mathsf{exp}} = [\![\gamma\ell]\!]_0 - [\![\gamma]\!]_0 \cdot \ell'$.

The adversary sends some $[\![\epsilon]\!]'_0$, and the check passes if and only if $[\![\epsilon]\!]'_0 + [\![\epsilon]\!]_1 = 0$. Thus, to pass the check, the adversary must send $[\![\epsilon]\!]'_0 = -[\![\epsilon]\!]_1 = [\![\epsilon]\!]_0^{\mathsf{exp}} + \gamma \cdot \Delta_\ell$. Since the adversary can compute $[\![\epsilon]\!]_0^{\mathsf{exp}} = [\![\gamma\ell]\!]_0 - [\![\gamma]\!]_0 \cdot \ell'$ from their view, passing the check requires guessing $\gamma \cdot \Delta_\ell$.

By the winning condition, $\Delta_\ell \neq 0$. Since $\gamma \leftarrow\!\!\$\, \mathbb{F}$ is independent of the adversary's view, the value $\gamma \cdot \Delta_\ell$ is uniformly distributed over $\mathbb{F}$, and thus can be guessed with probability $1/|\mathbb{F}|$.

Therefore, $\Pr[\mathsf{A}\text{ wins}] = 1/|\mathbb{F}|$. □

D Full Description of Preprocessing Protocol for Authenticated Shuffle Tuples

An authenticated shuffle tuple for permutation π chosen by party P_0 consists of the following values for P_0 (left) and P_1 (right) where P_0 chooses random $\kappa \leftarrow\!\!\$\, \mathbb{F}$ and P_1 chooses random vectors $\vec{a}, \vec{b} \leftarrow\!\!\$\, \mathbb{F}^n$:

$$\left(\pi, \kappa, \langle \pi(\vec{a}) \rangle_0^{(\kappa)}, \langle \pi(\vec{b}) \rangle_0^{(\kappa)}\right), \quad \left(\vec{a}, \vec{b}, \langle \pi(\vec{a}) \rangle_1^{(\kappa)}, \langle \pi(\vec{b}) \rangle_1^{(\kappa)}\right).$$

An authenticated inverse shuffle tuple for permutation π chosen by party P_0 consists of the following values for P_0 (left) and P_1 (right) where P_0 chooses random $\kappa \leftarrow\!\!\$\, \mathbb{F}$ and P_1 chooses random vectors $\vec{a}, \vec{b} \leftarrow\!\!\$\, \mathbb{F}^n$:

$$\left(\pi, \kappa, \langle \pi^{-1}(\vec{a}) \rangle_0^{(\kappa)}, \langle \pi^{-1}(\vec{b}) \rangle_0^{(\kappa)}\right), \quad \left(\vec{a}, \vec{b}, \langle \pi^{-1}(\vec{a}) \rangle_1^{(\kappa)}, \langle \pi^{-1}(\vec{b}) \rangle_1^{(\kappa)}\right).$$

In Figure 19, we describe our preprocessing protocol for generating authenticated shuffle tuples with the Mohassel et al. [MSS14] malicious-secure permutation network. We divide our preprocessing protocol into three parts: the generation of

Simulator $\mathsf{S}_{\mathrm{Online}}[\mathrm{Corrupt}\ \mathsf{P}_0](\mathsf{A})$:

$\mathsf{Init}(\lambda)$:

1. $\mathsf{expShare} \leftarrow \{\cdot\}$: maps id to expected adversary share $([\![v]\!]_0, [\![\gamma v]\!]_0)$.
2. $L_{auth} \leftarrow [\cdot]$: queue of pending (non-blind) auth checks from multiplication opens, each entry $(z, [\![z]\!]_0^{\mathsf{exp}}, [\![z]\!]_0^{\mathsf{real}}, [\![\gamma z]\!]_0^{\mathsf{exp}})$ storing opened value, expected adversary share, actual adversary share, and expected MAC share.
3. $L_{bauth} \leftarrow [\cdot]$: queue of pending blind auth checks from permutations, each entry $([\![z]\!]_0, [\![\gamma z]\!]_0)$ storing expected adversary value and MAC shares.
4. $\mathsf{permInputDev} \leftarrow \mathsf{false}$: deviation in input to permutation.

On $\mathsf{Prep}(n_{inp,0}, n_{inp,1}, n_m, \mathsf{in}_\pi)$:

1. Forward $\mathsf{Prep}(n_{inp,0}, n_{inp,1}, n_m, \mathsf{in}_\pi)$ to $\mathsf{F}_{\mathsf{ArithPermCirc}}$.
2. Sample $\gamma \leftarrow\!\!\$\ \mathbb{F}$ and $[\![\gamma]\!]_0 \leftarrow\!\!\$\ \mathbb{F}$.
3. Simulate $\mathsf{F}_{\mathsf{SPDZPrep}}$ output: sample random shares for input/multiplication tuples.
4. Simulate $\mathsf{F}_{\mathsf{ShufTuplePrep}}$: store adversary's input (permutations π, keys κ for tuples P_0 applies; blinding vectors (a, b) for tuples P_1 applies) and return random shares $([\![\tilde{a}]\!]_0, [\![\kappa\tilde{a}]\!]_0, [\![\tilde{b}]\!]_0, [\![\kappa\tilde{b}]\!]_0)$.
5. Return simulated preprocessing to A.

On $\mathsf{Input}(\mathsf{id}, v)$ from P_0:

1. Pop stored input tuple $(r, [\![r]\!]_0, [\![\gamma r]\!]_0)$.
2. Receive ϵ from adversary; compute $v \leftarrow r + \epsilon$.
3. Forward $\mathsf{Input}(\mathsf{id}, v)$ to $\mathsf{F}_{\mathsf{ArithPermCirc}}$.
4. Compute $[\![v]\!]_0 \leftarrow [\![r]\!]_0 + \epsilon$ and $[\![\gamma v]\!]_0 \leftarrow [\![\gamma r]\!]_0 + [\![\gamma]\!]_0 \cdot \epsilon$.
5. Store $\mathsf{expShare}[\mathsf{id}] \leftarrow ([\![v]\!]_0, [\![\gamma v]\!]_0)$.

On $\mathsf{Input}(\mathsf{id})$ from P_1:

1. Pop stored input tuple $([\![r]\!]_0, [\![\gamma r]\!]_0)$.
2. Forward $\mathsf{Input}(\mathsf{id})$ to $\mathsf{F}_{\mathsf{ArithPermCirc}}$.
3. Sample $\epsilon \leftarrow\!\!\$\ \mathbb{F}$ and send to adversary.
4. Compute $[\![v]\!]_0 \leftarrow [\![r]\!]_0 + \epsilon$ and $[\![\gamma v]\!]_0 \leftarrow [\![\gamma r]\!]_0 + [\![\gamma]\!]_0 \cdot \epsilon$.
5. Store $\mathsf{expShare}[\mathsf{id}] \leftarrow ([\![v]\!]_0, [\![\gamma v]\!]_0)$.

On ScalAdd, Add, ScalMul:

1. Forward operation to $\mathsf{F}_{\mathsf{ArithPermCirc}}$.
2. Update $\mathsf{expShare}$ for output id by applying the corresponding linear operation to the input entries.

On $\mathsf{Reconstruct}(\mathsf{id})$: The non-trivial part of the simulator. See Figure 17.

On $\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$:

1. Forward $\mathsf{Mul}(\mathsf{id}_1, \mathsf{id}_2, \mathsf{id}_3)$ to $\mathsf{F}_{\mathsf{ArithPermCirc}}$.
2. Pop Beaver triple shares $([\![a]\!]_0, [\![\gamma a]\!]_0, [\![b]\!]_0, [\![\gamma b]\!]_0, [\![c]\!]_0, [\![\gamma c]\!]_0)$.
3. Look up $([\![x]\!]_0, [\![\gamma x]\!]_0) \leftarrow \mathsf{expShare}[\mathsf{id}_1]$ and $([\![y]\!]_0, [\![\gamma y]\!]_0) \leftarrow \mathsf{expShare}[\mathsf{id}_2]$.
4. Sample $d_1, e_1 \leftarrow\!\!\$\ \mathbb{F}$ and send (d_1, e_1) to adversary.
5. Receive (d_0, e_0) from adversary; compute $d \leftarrow d_0 + d_1$ and $e \leftarrow e_0 + e_1$.
6. Compute adversary's output share:
   - $[\![z]\!]_0 \leftarrow [\![c]\!]_0 + d \cdot [\![b]\!]_0 + e \cdot [\![a]\!]_0 + d \cdot e$
   - $[\![\gamma z]\!]_0 \leftarrow [\![\gamma c]\!]_0 + d \cdot [\![\gamma b]\!]_0 + e \cdot [\![\gamma a]\!]_0 + [\![\gamma]\!]_0 \cdot d \cdot e$
7. Store $\mathsf{expShare}[\mathsf{id}_3] \leftarrow ([\![z]\!]_0, [\![\gamma z]\!]_0)$.
8. Compute expected shares for pending auth check:
   - $[\![d]\!]_0^{\mathsf{exp}} \leftarrow [\![x]\!]_0 - [\![a]\!]_0, \quad [\![\gamma d]\!]_0^{\mathsf{exp}} \leftarrow [\![\gamma x]\!]_0 - [\![\gamma a]\!]_0$
   - $[\![e]\!]_0^{\mathsf{exp}} \leftarrow [\![y]\!]_0 - [\![b]\!]_0, \quad [\![\gamma e]\!]_0^{\mathsf{exp}} \leftarrow [\![\gamma y]\!]_0 - [\![\gamma b]\!]_0$
9. Push $(d, [\![d]\!]_0^{\mathsf{exp}}, d_0, [\![\gamma d]\!]_0^{\mathsf{exp}})$ and $(e, [\![e]\!]_0^{\mathsf{exp}}, e_0, [\![\gamma e]\!]_0^{\mathsf{exp}})$ to L_{auth}.

On $\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$:

1. Forward $\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ to $\mathsf{F}_{\mathsf{ArithPermCirc}}$.
2. Look up input shares: $\forall i,\ ([\![v_i]\!]_0, [\![\gamma v_i]\!]_0) \leftarrow \mathsf{expShare}[\mathsf{id}_i]$.
3. **Case** $b' = 0$ (adversary applies π):
   (a) Pop shuffle tuple: $(\pi, \kappa, [\![\vec{\tilde{a}}]\!]_0, [\![\kappa\vec{\tilde{a}}]\!]_0, [\![\vec{\tilde{b}}]\!]_0, [\![\kappa\vec{\tilde{b}}]\!]_0)$.
   (b) Sample $\vec{c}, \vec{d} \leftarrow\!\!\$\ \mathbb{F}^m$ and send to adversary.
   (c) Compute expected output shares:
   - $[\![\vec{\tilde{v}}]\!]_0 \leftarrow \pi([\![\vec{v}]\!]_0) + \pi(\vec{c}) + [\![\vec{\tilde{a}}]\!]_0$
   - $[\![\gamma\vec{\tilde{v}}]\!]_0 \leftarrow \pi([\![\gamma\vec{v}]\!]_0) + \pi(\vec{d}) + [\![\vec{\tilde{b}}]\!]_0$
   - $[\![\kappa\vec{\tilde{v}}]\!]_0 \leftarrow \kappa \cdot \pi([\![\vec{v}]\!]_0) + \kappa \cdot \pi(\vec{c}) + [\![\kappa\vec{\tilde{a}}]\!]_0$
   - $[\![\gamma\kappa\vec{\tilde{v}}]\!]_0 \leftarrow \kappa \cdot \pi([\![\gamma\vec{v}]\!]_0) + \kappa \cdot \pi(\vec{d}) + [\![\kappa\vec{\tilde{b}}]\!]_0$
4. **Case** $b' = 1$ (honest party applies π):
   (a) Pop shuffle tuple: $(\vec{a}, \vec{b}, [\![\vec{\tilde{a}}]\!]_0, [\![\kappa\vec{\tilde{a}}]\!]_0, [\![\vec{\tilde{b}}]\!]_0, [\![\kappa\vec{\tilde{b}}]\!]_0)$.
   (b) Receive $(\vec{c}, \vec{d})$ from adversary.
   (c) Compute $\vec{c}^{\mathsf{exp}} \leftarrow [\![\vec{v}]\!]_0 - \vec{a}$, $\vec{d}^{\mathsf{exp}} \leftarrow [\![\gamma\vec{v}]\!]_0 - \vec{b}$. If $(\vec{d} - \vec{d}^{\mathsf{exp}}) - \gamma \cdot (\vec{c} - \vec{c}^{\mathsf{exp}}) \neq \vec{0}$, set $\mathsf{permInputDev} \leftarrow \mathsf{true}$.
   (d) Compute expected output shares:
   - $[\![\vec{\tilde{v}}]\!]_0 \leftarrow [\![\vec{\tilde{a}}]\!]_0$
   - $[\![\gamma\vec{\tilde{v}}]\!]_0 \leftarrow [\![\vec{\tilde{b}}]\!]_0$
   - $[\![\kappa\vec{\tilde{v}}]\!]_0 \leftarrow [\![\kappa\vec{\tilde{a}}]\!]_0$
   - $[\![\gamma\kappa\vec{\tilde{v}}]\!]_0 \leftarrow [\![\kappa\vec{\tilde{b}}]\!]_0$
5. Store $\forall i,\ \mathsf{expShare}[\widetilde{\mathsf{id}}_i] \leftarrow ([\![\tilde{v}_i]\!]_0, [\![\gamma\tilde{v}_i]\!]_0)$.
6. Push $([\![\kappa\vec{\tilde{v}}]\!]_0, [\![\gamma\kappa\vec{\tilde{v}}]\!]_0)$ to L_{bauth}.

On $\mathsf{PermuteInv}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$:

1. Same as $\mathsf{Permute}$, using inverse shuffle tuples and π^{-1}.

Figure 16: Simulator S for the online protocol FLOSS when P_0 is corrupted.

Simulator $\mathsf{S}_{\mathrm{Online-Reconstruct}}[\mathrm{Corrupt}\ \mathsf{P}_0](\mathsf{A})$:

On $\mathsf{Reconstruct}(\mathsf{id})$:

1. **Non-blind auth check** on L_{auth} (see Fig. 12):
   (a) Commit-open for r: Sample $r_1 \leftarrow\!\!\$\ \mathbb{F}$; commit via $\mathsf{F}_{\mathsf{Comm}}$. Receive adversary's commit; extract r_0. Open; receive adversary's open. Compute $r \leftarrow r_0 + r_1$.
   (b) For each $(z_i, [\![z_i]\!]_0^{\mathsf{exp}}, [\![z_i]\!]_0^{\mathsf{real}}, [\![\gamma z_i]\!]_0^{\mathsf{exp}}) \in L_{auth}$, compute $\delta_i \leftarrow [\![z_i]\!]_0^{\mathsf{real}} - [\![z_i]\!]_0^{\mathsf{exp}}$.
   (c) Compute $\ell \leftarrow \sum_i r^{i-1} z_i, \quad [\![\gamma\ell]\!]_0^{\mathsf{exp}} \leftarrow \sum_i r^{i-1} [\![\gamma z_i]\!]_0^{\mathsf{exp}}, \quad \Delta \leftarrow \sum_i r^{i-1}\delta_i$.
   (d) Compute $[\![\epsilon]\!]_0^{\mathsf{exp}} \leftarrow [\![\gamma\ell]\!]_0^{\mathsf{exp}} - [\![\gamma]\!]_0 \cdot \ell$.
   (e) Commit-open for ϵ: Receive adversary's commit; extract $[\![\epsilon]\!]_0$. Commit to $[\![\epsilon]\!]_1 \leftarrow -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma \cdot \Delta$. Open.
   (f) If $[\![\epsilon]\!]_0 + [\![\epsilon]\!]_1 \neq 0$, abort.
2. **Blind auth check** on L_{bauth} (see Fig. 2):
   (a) Generate random blind $\alpha = \alpha_0 + \alpha_1$:
      i. Pop $([\![s_1]\!]_0, [\![\gamma s_1]\!]_0)$. Sample $\epsilon_1 \leftarrow\!\!\$\ \mathbb{F}$ and send to adversary. Compute $[\![\alpha_1]\!]_0 \leftarrow [\![s_1]\!]_0 + \epsilon_1$ and $[\![\gamma\alpha_1]\!]_0 \leftarrow [\![\gamma s_1]\!]_0 + [\![\gamma]\!]_0 \cdot \epsilon_1$.
      ii. Pop $(s_0, [\![s_0]\!]_0, [\![\gamma s_0]\!]_0)$. Receive ϵ_0 from adversary. Compute $[\![\alpha_0]\!]_0 \leftarrow [\![s_0]\!]_0 + \epsilon_0$ and $[\![\gamma\alpha_0]\!]_0 \leftarrow [\![\gamma s_0]\!]_0 + [\![\gamma]\!]_0 \cdot \epsilon_0$.
      iii. Compute $[\![\alpha]\!]_0 \leftarrow [\![\alpha_0]\!]_0 + [\![\alpha_1]\!]_0$ and $[\![\gamma\alpha]\!]_0 \leftarrow [\![\gamma\alpha_0]\!]_0 + [\![\gamma\alpha_1]\!]_0$.
   (b) Commit-open for r: As in non-blind check.
   (c) Using $([\![z_i]\!]_0, [\![\gamma z_i]\!]_0)$ from L_{bauth}, compute:
      • $[\![\ell]\!]_0^{\mathsf{exp}} \leftarrow [\![\alpha]\!]_0 + \sum_i r^{i-1} [\![z_i]\!]_0$
      • $[\![\gamma\ell]\!]_0^{\mathsf{exp}} \leftarrow [\![\gamma\alpha]\!]_0 + \sum_i r^{i-1} [\![\gamma z_i]\!]_0$
   (d) Open ℓ: Send $\ell_1 \leftarrow\!\!\$\ \mathbb{F}$. Receive ℓ_0; compute $\ell \leftarrow \ell_0 + \ell_1$ and $\Delta_\ell \leftarrow \ell_0 - [\![\ell]\!]_0^{\mathsf{exp}}$.
   (e) Commit-open for ϵ: Compute $[\![\epsilon]\!]_0^{\mathsf{exp}} \leftarrow [\![\gamma\ell]\!]_0^{\mathsf{exp}} - [\![\gamma]\!]_0 \cdot \ell$. Receive adversary's commit; extract $[\![\epsilon]\!]_0$. If $\mathsf{permInputDev}$, commit to $[\![\epsilon]\!]_1 \leftarrow\!\!\$\ \mathbb{F}$; else commit to $[\![\epsilon]\!]_1 \leftarrow -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma \cdot \Delta_\ell$. Open.
   (f) If $[\![\epsilon]\!]_0 + [\![\epsilon]\!]_1 \neq 0$, abort.
3. **Reconstruct** v:
   (a) Forward $\mathsf{Reconstruct}(\mathsf{id})$ to $\mathsf{F}_{\mathsf{ArithPermCirc}}$; receive value v.
   (b) Look up $([\![v]\!]_0, [\![\gamma v]\!]_0) \leftarrow \mathsf{expShare}[\mathsf{id}]$.
   (c) Send $[\![v]\!]_1 \leftarrow v - [\![v]\!]_0$ to adversary. Receive $[\![v]\!]_0'$ from adversary.
   (d) Compute $v' \leftarrow [\![v]\!]_0' + [\![v]\!]_1$ and $\delta \leftarrow [\![v]\!]_0' - [\![v]\!]_0$.
   (e) Compute $[\![\epsilon]\!]_0^{\mathsf{exp}} \leftarrow [\![\gamma v]\!]_0 - [\![\gamma]\!]_0 \cdot v'$. Receive $[\![\epsilon]\!]_0$ from adversary. Send $[\![\epsilon]\!]_1 \leftarrow -[\![\epsilon]\!]_0^{\mathsf{exp}} - \gamma \cdot \delta$.
   (f) If $[\![\epsilon]\!]_0 + [\![\epsilon]\!]_1 \neq 0$, abort.
   (g) Route output v from $\mathsf{F}_{\mathsf{ArithPermCirc}}$ to the honest party.

Figure 17: Simulator for Reconstruct operation when P_0 is corrupted, used as a subprotocol in the simulator for the proof of security of the online protocol FLOSS.

forward shuffle tuples, the generation of inverse shuffle tuples, and the application of the shuffle tuple key. First, we describe how to generate forward shuffle tuples $(\langle\pi(\vec{a})\rangle, \langle\pi(\vec{b})\rangle)$ from P_0's permutation π and P_1's vectors $\vec{a}$ and $\vec{b}$ using $\mathsf{ShufTupGen}$ (Figure 20). Then, we describe how to generate inverse shuffle tuples $(\langle\pi^{-1}(\vec{a})\rangle, \langle\pi^{-1}(\vec{b})\rangle)$ from P_0's permutation π and P_1's vectors $\vec{a}$ and $\vec{b}$ using $\mathsf{InvShufTupGen}$ (Figure 21). Finally, we describe how to obtain κ-authenticated shares $(\langle\pi(\vec{a})\rangle^{(\kappa)}, \langle\pi(\vec{b})\rangle^{(\kappa)})$ (resp. $(\langle\pi^{-1}(\vec{a})\rangle^{(\kappa)}, \langle\pi^{-1}(\vec{b})\rangle^{(\kappa)})$) from the forward shuffle tuples $(\langle\pi(\vec{a})\rangle, \langle\pi(\vec{b})\rangle)$ (resp. inverse shuffle tuples $(\langle\pi^{-1}(\vec{a})\rangle, \langle\pi^{-1}(\vec{b})\rangle)$) using $\mathsf{ApplyKappaPrep}$ (Figure 22).

D.1 Switching and Permutation Networks

Switching Network. A switching network is a directed graph (V, E) defined by vertices V and edges E, where each vertex has two incoming edges and two outgoing edges. The binary switches are represented as vertices of the graph $(V = (v_1, \cdots, v_n))$, which we assume to be topologically ordered. The wires are represented as edges of the graph $(E = (e_1, \cdots, e_m))$. The value of the wire is represented by the mapping $\mathsf{Val} : E \to \mathbb{F}$. The control bits of the switches are represented by $C = (c_1, \cdots, c_n) \in \{0,1\}^n$. The input wires are represented by the set $I = \{i_1, \cdots, i_{m_{\mathsf{in}}}\} \subset E$, where m_{in} is the number of inputs of the switching network. The input wires are the leftmost wires in the switching network. The output wires are represented by the set $O = \{o_1, \cdots, o_{m_{\mathsf{out}}}\} \subset E$, where m_{out} is the number of outputs of the switching network. The output wires are the rightmost wires in the switching network.

We define two mappings from switch to edges: $\mathsf{In} : V \to E^2$ and $\mathsf{Out} : V \to E^2$.

For all $j \in [n]$, where n is the number of switches:

- $\mathsf{In}(v_j) \to (\mathsf{in}_0, \mathsf{in}_1)$ maps switch v_j to its input wire indices $(\mathsf{in}_0, \mathsf{in}_1)$, where in_0 is the index of the top input wire and in_1 is the index of the bottom input wire.
- $\mathsf{Out}(v_j) \to (\mathsf{out}_0, \mathsf{out}_1)$ maps switch v_j to its output wire indices $(\mathsf{out}_0, \mathsf{out}_1)$, where out_0 is the index of the top output wire and out_1 is the index of the bottom output wire.

From these definitions, when switch v_j with input wire indices $(\mathsf{in}_0, \mathsf{in}_1)$ and output wire indices $(\mathsf{out}_0, \mathsf{out}_1)$ is set to control bit b_j then:

- $\mathsf{Val}(\mathsf{in}_0) = \mathsf{Val}(\mathsf{out}_{c_j})$
- $\mathsf{Val}(\mathsf{in}_1) = \mathsf{Val}(\mathsf{out}_{1-c_j})$

In other words, the switch v_j sets the output wire values equal to its corresponding input wire values when $b_j = 0$. On the other hand, the switch sets the output wire values equal to the opposite input wire value when $b_j = 1$, "switching" the values of the wires.

Permutation Network. A permutation network is a switching network with three properties:

1. The number of input wires is equal to the number of output wires. For simplicity, we represent the number of input/output wires as m', where $m' = m_{\mathsf{in}} = m_{\mathsf{out}}$.
2. Soundness: $\forall c \in C,\ \exists \pi \in \mathbb{S}_{m'},\ \{(\mathsf{Val}(i_j), j)\}_{j\in[m']} = \{(\mathsf{Val}(o_{\pi^{-1}(j)}), \pi^{-1}(j))\}_{j\in[m']}$
   - In other words, all possible assignments of control bits in the switching network must correspond to some m'-length permutation, where the value of the input wire is the same as the value of the output wire indexed by the permutation applied to the input wire's index.
3. Completeness: $\forall \pi \in \mathbb{S}_{m'},\ \exists c \in C,\ \{(\mathsf{Val}(i_j), j)\}_{j\in[m']} = \{(\mathsf{Val}(o_{\pi^{-1}(j)}), \pi^{-1}(j))\}_{j\in[m']}$
   - In other words, all m'-length permutations must correspond to some assignment of control bits in the switching network, where the value of the input wire is the same as the value of the output wire indexed by the permutation applied to the input wire's index.

According to these definitions, there is a sequence of control bits $(c_1, \ldots, c_n)$ uniquely encoding $\pi \in \mathbb{S}_{m'}$ that the permuter locally computes from the permutation network topology (e.g. m'-length Waksman network [Wak68]). Given m' terminals (input/output wires), the Waksman permutation network contains $m' \cdot \lceil \log_2(m') \rceil - m' + 1$ switches.

$\mathsf{P}_0.\mathsf{Coin}(n \in \mathbb{F}) \leftrightarrow \mathsf{P}_1.\mathsf{Coin}(n \in \mathbb{F})$

Output: $\langle \vec{R} \rangle$, where $\vec{R} \leftarrow\!\!\$\ \mathbb{F}^n$

1. $\vec{R}_b \leftarrow\!\!\$\ \mathbb{F}^n$
2. $\langle \vec{R}_b \rangle \leftarrow \mathsf{P}_b.\mathsf{Input}(\langle \vec{R}_b \rangle_b, \vec{R}_b) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{RcvInput}(\langle \vec{R}_b \rangle_{1-b})$
3. $\langle \vec{R}_{1-b} \rangle \leftarrow \mathsf{P}_b.\mathsf{RcvInput}(\langle \vec{R}_{1-b} \rangle_b) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Input}(\langle \vec{R}_{1-b} \rangle_{1-b}, \vec{R}_{1-b})$
4. $\langle \vec{R} \rangle \leftarrow \langle \vec{R}_b \rangle + \langle \vec{R}_{1-b} \rangle$

Figure 18: Subroutine that generates γ-authenticated shares of a vector of n unknown random field elements.

D.2 Malicious-Secure Permutation Network Construction: Forward Shuffle Tuples

We describe the construction of the malicious-secure permutation network from Mohassel et al. [MSS14]. We can construct subprotocols to generate forward shuffle tuples and inverse shuffle tuples from the above permutation network construction, described in Figure 20 and Figure 21. Here, we describe first the construction of forward shuffle tuples $\mathsf{ShufTupGen}$ from the permutation network. We use subroutines from SPDZ since throughout the protocol, we use arithmetic circuit operations such as $\mathsf{Reconstruct}$ and Mul.

For simplicity, we describe a subroutine $\mathsf{P}_b.\mathsf{Coin}(n) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Coin}(n)$ to generate γ-authenticated shares of a length-n vector of unknown random field elements $\vec{R} \in \mathbb{F}^n$ to both parties. It is described in Figure 18. Note that when running the Mohassel et al. [MSS14] shuffle protocol standalone, the circuit preparation phase can be performed in offline preprocessing.

Preparing the circuit. This portion of the forward shuffle tuple generation protocol prepares the permutation network circuit, which is independent of the shuffle-tuple vectors $\vec{a}$ and $\vec{b}$. First, the parties must generate random authenticated values for each wire in the permutation network. For each wire $e \in E$, the parties must generate a random data blind $\langle r_e \rangle$ and random MAC blind $\langle l_e \rangle$. The parties call the $\mathsf{Coin}(|E|) = \mathsf{Coin}(2m\log_2(m) - m + 1)$ subroutine twice to generate data blinds and authentication blinds for all wires in the network. The MAC blind $\langle l_e \rangle$ is used to authenticate the data blind $\langle r_e \rangle$ as P_0 evaluates the permutation network. To ensure that the MAC blinds are consistent with the data blinds, the parties sample a random multiplicative key $\langle \iota \rangle$ using the $\mathsf{Coin}(1)$ subroutine. Since P_1 supplies two independent input vectors, $\vec{a}$ and $\vec{b}$, the parties must repeat the above process twice, generating new data blinds, MAC blinds, and multiplicative key for each vector.

The next step is for P_0 to locally parse π into topologically-ordered control bits $(c_1, \cdots, c_k) \in \{0,1\}^k$ for switching network of input length $m = |\pi|$. P_0 uses a permutation network, such as a Waksman network [Wak68], to convert its permutation into control bits. P_0 will then share its control bits with P_1 via SPDZ's Input subroutine. Using authenticated shares of the control bits $\langle c_i \rangle$, P_1 verifies that P_0 inputted properly formatted bits with the bit-check circuit $\langle c_i \rangle \cdot (1 - \langle c_i \rangle)$. P_0 and P_1 $\mathsf{Reconstruct}$ the output, and if $c_i \cdot (1 - c_i) \neq 0$, the parties output `abort`.

The last step is for P_0 and P_1 to route the blind values of the switches' input wires to the output wires using P_0's inputted control bits. Observe that for a switch in the permutation network, the value of the output wire only depends on the switch's control bit and the values of the input wires. Since the parties already precomputed blinds for each wire, and P_0 already inputted its control bits, the parties can compute the proper routing of the input blinds to output blinds of each switch in the network simultaneously.

For each switch index $i \in [k]$, the parties compute the following in parallel:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
\langle s_{\mathsf{in}_0} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{\mathsf{out}_0} \rangle - \langle r_{\mathsf{in}_0} \rangle) + \langle c_i \rangle \cdot (\langle r_{\mathsf{out}_1} \rangle - \langle r_{\mathsf{in}_0} \rangle) \\
\langle s_{\mathsf{in}_1} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{\mathsf{out}_1} \rangle - \langle r_{\mathsf{in}_1} \rangle) + \langle c_i \rangle \cdot (\langle r_{\mathsf{out}_0} \rangle - \langle r_{\mathsf{in}_1} \rangle)
\end{aligned}$$

For each switch index $i \in [k]$, the parties reconstruct the wire routing in parallel:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
s_{\mathsf{in}_0} &\leftarrow \mathsf{Reconstruct}(\langle s_{\mathsf{in}_0} \rangle) \\
s_{\mathsf{in}_1} &\leftarrow \mathsf{Reconstruct}(\langle s_{\mathsf{in}_1} \rangle)
\end{aligned}$$

For each switch index $i \in [k]$, the parties compute the following in parallel:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
\langle \tau_{\mathsf{in}_0} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{\mathsf{out}_0} \rangle - \langle l_{\mathsf{in}_0} \rangle) + \langle c_i \rangle \cdot (\langle l_{\mathsf{out}_1} \rangle - \langle l_{\mathsf{in}_0} \rangle) + s_{\mathsf{in}_0} \cdot \langle \iota \rangle \\
\langle \tau_{\mathsf{in}_1} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{\mathsf{out}_1} \rangle - \langle l_{\mathsf{in}_1} \rangle) + \langle c_i \rangle \cdot (\langle l_{\mathsf{out}_0} \rangle - \langle l_{\mathsf{in}_1} \rangle) + s_{\mathsf{in}_1} \cdot \langle \iota \rangle
\end{aligned}$$

For each switch index $i \in [k]$, the parties reconstruct the authenticated wire routing in parallel:

$$\begin{aligned}
\tau_{\mathsf{in}_0} &\leftarrow \mathsf{Reconstruct}(\langle \tau_{\mathsf{in}_0} \rangle) \\
\tau_{\mathsf{in}_1} &\leftarrow \mathsf{Reconstruct}(\langle \tau_{\mathsf{in}_1} \rangle)
\end{aligned}$$

Evaluating the permutation network. In this portion of the protocol, the parties prepare the inputs to the permutation network circuit by adding their input shares to the input wire blinds. We repeat this process twice for $x \in \{\vec{a}, \vec{b}\}$.
For each input index $j \in [m]$, the parties compute the following in parallel:

$$\langle t_{i_j} \rangle \leftarrow \langle x_j \rangle + \langle r_{i_j} \rangle$$

For each input index $j \in [m]$, the parties reconstruct the input wire values in parallel:

$$t_{i_j} \leftarrow \mathsf{Reconstruct}(\langle t_{i_j} \rangle)$$

For each input index $j \in [m]$, the parties compute the following in parallel:

$$\langle \tau_{i_j} \rangle \leftarrow \langle l_{i_j} \rangle + t_{i_j} \cdot \langle \iota \rangle$$

For each input index $j \in [m]$, the parties reconstruct the authenticated input wire values in parallel:

$$\tau_{i_j} \leftarrow \mathsf{Reconstruct}(\langle \tau_{i_j} \rangle)$$

Next, the evaluator locally evaluates the permutation network circuit using the input wire data values and MAC values, starting from the input wires and ending at the output wires.
For each switch index $i \in [k]$, P_0 evaluates the network via topologically-ordered switches:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
\langle t_{\mathsf{out}_{c_i}} \rangle &\leftarrow s_{\mathsf{in}_0} + t_{\mathsf{in}_0} \\
\langle t_{\mathsf{out}_{1-c_i}} \rangle &\leftarrow s_{\mathsf{in}_1} + t_{\mathsf{in}_1} \\
\langle \tau_{\mathsf{out}_{c_i}} \rangle &\leftarrow \Delta_{\mathsf{in}_0} + \tau_{\mathsf{in}_0} \\
\langle \tau_{\mathsf{out}_{1-c_i}} \rangle &\leftarrow \Delta_{\mathsf{in}_1} + \tau_{\mathsf{in}_1}
\end{aligned}$$

Finally, the parties perform an authentication check on the output wires' data values (t_{o_j}) and MAC values (τ_{o_j}) to verify the permuter computed the network correctly. For each output index $j \in [m]$, P_0 sends (t_{o_j}, τ_{o_j}) in parallel to P_1. Then, the parties compute a check circuit, corresponding to Step 4 in Figure 20. For each output index $j \in [m]$, the parties compute the following in parallel:

$$\langle \gamma_{o_j} \rangle \leftarrow \langle l_{o_j} \rangle + t_{o_j} \cdot \langle \iota \rangle - \tau_{o_j}$$

For each output index $j \in [m]$, the parties reconstruct the authentication check in parallel:

$$\gamma_{o_j} \leftarrow \mathsf{Reconstruct}(\gamma_{o_j})$$

Lastly, the parties locally check if $\gamma_{o_j} = 0$; if not, the parties output `abort`. If the check passes, P_0 and P_1 output their shares of the authenticated shuffle tuple:

$$\begin{aligned}
\langle \pi(x)_j \rangle_0 &\leftarrow t_{o_j} - \langle r_{o_j} \rangle_0 \\
\langle \pi(x)_j \rangle_1 &\leftarrow t_{o_j} - \langle r_{o_j} \rangle_1
\end{aligned}$$

D.3 Constructing Inverse Shuffle Tuples

The inverse shuffle tuple generation protocol $\mathsf{InvShufTupGen}$ is similar to the forward shuffle tuple generation protocol $\mathsf{ShufTupGen}$, but there are three main differences.

Reusing control bits. First, the parties must reuse the same shared control bits $(\langle c_i \rangle)_{i \in [k]}$ as in the forward shuffle tuple generation protocol. The same control bits $[c_i]$ imply that the same permutation π is used in both protocols. This ensures that the inverse shuffle tuples constructed by $\mathsf{InvShufTupGen}$ use the same permutation π as the forward shuffle tuples constructed by $\mathsf{ShufTupGen}$. Conveniently, reusing a permutation eliminates k total Input and $\mathsf{Reconstruct}$ operations, since there is no need to reshare the control bits and perform another bit-equality check.

Preparing the circuit: reversing the wire routing. Second, recall for a switch v_i, the right wires correspond to out_0 and out_1 from $\mathsf{Out}(v_i)$, and the left wires correspond to in_0 and in_1 from $\mathsf{In}(v_i)$. Because the permutation network evaluation is reversed, the right wires of the switch (out) are routed to the left wires (in), instead of routing left wires to right wires in $\mathsf{ShufTupGen}$. We modify the circuit generation step of our offline portion to instead compute the *reverse* wire routing:
For each switch index $i \in [k]$, the parties compute the following in parallel:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
\langle s_{\mathsf{out}_0} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{\mathsf{in}_0} \rangle - \langle r_{\mathsf{out}_0} \rangle) + \langle c_i \rangle \cdot (\langle r_{\mathsf{in}_1} \rangle - \langle r_{\mathsf{out}_0} \rangle) \\
\langle s_{\mathsf{out}_1} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{\mathsf{in}_1} \rangle - \langle r_{\mathsf{out}_1} \rangle) + \langle c_i \rangle \cdot (\langle r_{\mathsf{in}_0} \rangle - \langle r_{\mathsf{out}_1} \rangle)
\end{aligned}$$

For each switch index $i \in [k]$, the parties reconstruct the *reverse* wire routing in parallel:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
s_{\mathsf{out}_0} &\leftarrow \mathsf{Reconstruct}(\langle s_{\mathsf{out}_0} \rangle) \\
s_{\mathsf{out}_1} &\leftarrow \mathsf{Reconstruct}(\langle s_{\mathsf{out}_1} \rangle)
\end{aligned}$$

For each switch index $i \in [k]$, the parties compute the following in parallel:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
\langle \tau_{\mathsf{out}_0} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{\mathsf{in}_0} \rangle - \langle l_{\mathsf{out}_0} \rangle) + \langle c_i \rangle \cdot (\langle l_{\mathsf{in}_1} \rangle - \langle l_{\mathsf{out}_0} \rangle) + s_{\mathsf{out}_0} \cdot \langle \iota \rangle \\
\langle \tau_{\mathsf{out}_1} \rangle &\leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{\mathsf{in}_1} \rangle - \langle l_{\mathsf{out}_1} \rangle) + \langle c_i \rangle \cdot (\langle l_{\mathsf{in}_0} \rangle - \langle l_{\mathsf{out}_1} \rangle) + s_{\mathsf{out}_1} \cdot \langle \iota \rangle
\end{aligned}$$

For each switch index $i \in [k]$, the parties reconstruct the authenticated *reverse* wire routing in parallel:

$$\begin{aligned}
\tau_{\mathsf{out}_0} &\leftarrow \mathsf{Reconstruct}(\langle \tau_{\mathsf{out}_0} \rangle) \\
\tau_{\mathsf{out}_1} &\leftarrow \mathsf{Reconstruct}(\langle \tau_{\mathsf{out}_1} \rangle)
\end{aligned}$$

Evaluating the permutation network in reverse. Third, the parties must instantiate their inputs x_j with the output (rightmost) wires o_j in the permutation network structure, instead of the input (leftmost) wires i_j. We change the following steps of the online portion to initialize with the output wires:
For each output index $j \in [m]$, the parties compute the following in parallel:

$$\langle t_{o_j} \rangle \leftarrow \langle x_j \rangle + \langle r_{o_j} \rangle$$

For each output index $j \in [m]$, the parties reconstruct the output wire values in parallel:

$$t_{o_j} \leftarrow \mathsf{Reconstruct}(\langle t_{o_j} \rangle)$$

For each output index $j \in [m]$, the parties compute the following in parallel:

$$\langle \tau_{o_j} \rangle \leftarrow \langle l_{o_j} \rangle + t_{o_j} \cdot \langle \iota \rangle$$

For each output index $j \in [m]$, the parties reconstruct the authenticated output wire values in parallel:

$$\tau_{o_j} \leftarrow \mathsf{Reconstruct}(\langle \tau_{o_j} \rangle)$$

P_0 must compute the permutation network circuit starting from the output wires and ending at the input wires. In other words, P_0 must compute the permutation network circuit traversing from right to left, instead of left to right in $\mathsf{ShufTupGen}$. We change the following steps of the online portion so P_0 evaluates the network in reverse:
For each switch index $i \in [k_{\text{rev}}]$, P_0 evaluates the network via *reverse* topologically-ordered switches:

$$\begin{aligned}
\mathsf{in}_0, \mathsf{in}_1 &\leftarrow \mathsf{In}(v_i) \\
\mathsf{out}_0, \mathsf{out}_1 &\leftarrow \mathsf{Out}(v_i) \\
\langle t_{\mathsf{in}_{c_i}} \rangle &\leftarrow s_{\mathsf{out}_0} + t_{\mathsf{out}_0} \\
\langle t_{\mathsf{in}_{1-c_i}} \rangle &\leftarrow s_{\mathsf{out}_1} + t_{\mathsf{out}_1} \\
\langle \tau_{\mathsf{in}_{c_i}} \rangle &\leftarrow \Delta_{\mathsf{out}_0} + \tau_{\mathsf{out}_0} \\
\langle \tau_{\mathsf{in}_{1-c_i}} \rangle &\leftarrow \Delta_{\mathsf{out}_1} + \tau_{\mathsf{out}_1}
\end{aligned}$$

The parties perform an authentication check on the input wires' data values (t_{i_j}) and MAC values (τ_{i_j}) to verify the permuter computed the network correctly, instead of the output wires. P_1 sends (t_{i_j}, τ_{i_j}) in parallel to P_0. Then for each input index $j \in [m]$, the parties compute the following in parallel:

$$\langle \gamma_{i_j} \rangle \leftarrow \langle l_{i_j} \rangle + t_{i_j} \cdot \langle \iota \rangle - \tau_{i_j}$$

For each input index $j \in [m]$, the parties reconstruct the authentication check in parallel:

$$\gamma_{i_j} \leftarrow \mathsf{Reconstruct}(\gamma_{i_j})$$

Lastly, P_0 and P_1's shares of the inverse shuffle tuple are defined with respect to the input wire values (t_{i_j}):

$$\begin{aligned}
\langle \pi^{-1}(x)_j \rangle_0 &\leftarrow t_{i_j} - \langle r_{i_j} \rangle_0 \\
\langle \pi^{-1}(x)_j \rangle_1 &\leftarrow -\langle r_{i_j} \rangle_1
\end{aligned}$$

D.4 Constructing κ-Authenticated Shuffle Tuples

$\mathsf{ApplyKappaPrep}$ constructs κ-authenticated shuffle tuples $\langle \overrightarrow{\widetilde{\mathrm{a}}} \rangle^{(\kappa)}$ and $\langle \overrightarrow{\widetilde{b}} \rangle^{(\kappa)}$ from shuffle tuples $\langle \overrightarrow{\widetilde{\mathrm{a}}} \rangle$ and $\langle \overrightarrow{\widetilde{b}} \rangle$. First, P_0 shares κ with P_1 using Input. Then, the parties use Mul to multiply their shares $\langle \overrightarrow{\widetilde{\mathrm{a}}} \rangle$ and $\langle \overrightarrow{\widetilde{b}} \rangle$ with shares $\langle \kappa \rangle$ to receive $\langle \kappa \cdot \overrightarrow{\widetilde{\mathrm{a}}} \rangle$ and $\langle \kappa \cdot \overrightarrow{\widetilde{b}} \rangle$.

To verify $\langle \kappa \cdot \overrightarrow{\widetilde{\mathrm{a}}} \rangle$ and $\langle \kappa \cdot \overrightarrow{\widetilde{b}} \rangle$ were computed with unmodified shares of $\overrightarrow{\widetilde{\mathrm{a}}}$ and $\overrightarrow{\widetilde{b}}$, the parties perform a blind authentication check with respect to γ on the shuffle tuple shares $\langle \overrightarrow{\widetilde{\mathrm{a}}} \rangle$ and $\langle \overrightarrow{\widetilde{b}} \rangle$ and the κ-authenticated shares $\langle \kappa \cdot \overrightarrow{\widetilde{\mathrm{a}}} \rangle$ and $\langle \kappa \cdot \overrightarrow{\widetilde{b}} \rangle$. Parties concatenate all of their data shares $[\![\overrightarrow{\widetilde{\mathrm{a}}}]\!] || [\![\kappa \cdot \overrightarrow{\widetilde{\mathrm{a}}}]\!] || [\![\overrightarrow{\widetilde{b}}]\!] || [\![\kappa \cdot \overrightarrow{\widetilde{b}}]\!]$ and γ-authenticated shares $[\![\gamma \overrightarrow{\widetilde{\mathrm{a}}}]\!] || [\![\gamma\kappa \cdot \overrightarrow{\widetilde{\mathrm{a}}}]\!] || [\![\gamma \overrightarrow{\widetilde{b}}]\!] || [\![\gamma\kappa \cdot \overrightarrow{\widetilde{b}}]\!]$ to authenticate the shuffle tuples in one-shot. Lastly, the parties need to verify that the multiplications (from multiplying $\langle \kappa \rangle$ with shuffle tuple shares) were computed correctly; they do this by emptying the authentication check queue and running the Auth subroutine on the remaining opened multiplications. If Auth passes, the parties output shares of their κ-authenticated shuffle tuples:

$$\begin{aligned}
\langle \overrightarrow{\widetilde{\mathrm{a}}} \rangle_0^{(\kappa)} &\leftarrow ([\![\overrightarrow{\widetilde{\mathrm{a}}}]\!], [\![\kappa \cdot \overrightarrow{\widetilde{\mathrm{a}}}]\!]) \\
\langle \overrightarrow{\widetilde{b}} \rangle_0^{(\kappa)} &\leftarrow ([\![\overrightarrow{\widetilde{b}}]\!], [\![\kappa \cdot \overrightarrow{\widetilde{b}}]\!])
\end{aligned}$$

D.5 Security

We present a corollary for our preprocessing protocol $\mathsf{PermNetSTPrep}$ in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid model.

Corollary 1. Let $\mathbb{F}$ be a finite field with $|\mathbb{F}| \geq 2^\lambda$. The protocol $\mathsf{PermNetSTPrep}[\mathbb{F}]$ (Figure 19) UC-realizes the ideal functionality $\mathsf{F}_{\mathsf{ShufTuplePrep}}[\mathbb{F}]$ (Figure 15) in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid model.

Proof sketch: If the corrupted party is sender P_1, the simulator samples two random vectors $\vec{a}, \vec{b} \leftarrow\!\!\$\ \mathbb{F}^n$. Then, in either case, the simulator simulates $\mathcal{F}_{\mathsf{SPDZPrep}}$ and simulates arithmetic circuit operations using the SPDZ simulator (see $\mathsf{S}_{\mathrm{Online}}$ in Figure 16) exactly as in the protocol specification for steps 2 and 3 of PermNetSTPrep (Figure 19) to retrieve shares $\langle\vec{\tilde{a}}\rangle$ and $\langle\vec{\tilde{b}}\rangle$. The security of subprotocols ShufTupGen$[\mathbb{F}]$ and InvShufTupGen$[\mathbb{F}]$ follow directly from Theorem 3 of Mohassel et al. [MSS14]. The permuter P_0 must correctly guess ι in ShufTupGen$[\mathbb{F}]$ or $\underline{\iota}$ in InvShufTupGen$[\mathbb{F}]$ to deviate from either protocol without detection. By union bound, this occurs with probability $\leq \frac{2}{|\mathbb{F}|}$, which is negligible in λ since $|\mathbb{F}| \geq 2^\lambda$. In ApplyKappaPrep$[\mathbb{F}]$, if the corrupt party is the permuter P_0, the simulator samples $\kappa \leftarrow\!\!\$\ \mathbb{F}$. The simulator acts like $\mathsf{S}_{\mathrm{Online}}$ (Figure 16) and follows the protocol specification for steps 1–4 of ApplyKappaPrep (Figure 22) to retrieve shares $\langle\vec{\tilde{a}}^{(\kappa)}\rangle$ and $\langle\vec{\tilde{b}}^{(\kappa)}\rangle$.

E Application: Sorting

In this section, we construct a malicious-secure 2PC radix sort protocol as an arithmetic permutation circuit. We show that the high level strategy taken in the honest-majority 3PC radix sort protocol of Asharov et al. [AHI+22] maps naturally onto the arithmetic permutation circuit model. The radix sort protocol is provided in Figure 23.

In radix sort, parties accumulate a sorting permutation bit-by-bit. Parties begin by computing a stable sorting permutation on the least-significant bit. Then in sequence, parties will apply the accumulated sorting permutation to the next-significant bit, compute a stable sorting permutation for that bit position, then compose the current bit position's stable sorting permutation with the accumulated sorting permutation. We break this up into a few subcircuits.

The first subcircuit BitSortInv (see Figure 24) produces shares of (the inverse of) the stable sorting permutation for a vector of secret-shared bits. Asharov et al.'s approach for BitSortInv translates directly as it can be expressed entirely using arithmetic operations (additions and multiplications); it does not rely on permutations.

The other two subcircuits, ApplyPerm and ApplyPermInv, apply and unapply (i.e., apply the inverse of) an arbitrary permutation encoded as secret-shared indices $\langle\rho\rangle$ to a vector of secret-shared data $\langle x\rangle$, respectively. These protocols make use of permutations, and so, we depart from the permuting strategy taken in the 3PC setting [AKK+23]. Nevertheless, the high level approach remains the same: a random permutation unknown to all parties is applied to $\langle\rho\rangle$ to *blind* ρ, and the resulting blinded permutation is reconstructed and applied locally by each party to $\langle x\rangle$. We instantiate this by having each party, in sequence, perform a one-sided permutation (as a gate in the arithmetic permutation circuit model) on $\langle\rho\rangle$ using a preprocessed random permutation π_b to produce $\langle\pi_1 \circ \pi_0 \circ \rho\rangle$. If there exists an honest party that selects π_b during preprocessing at random, then the resulting blinding permutation $\pi_1 \circ \pi_0$ is uniformly random and hides ρ. The full protocols are given in Figures 25 and 26. Note that ApplyPerm makes use of inverse permutation gates for π_1 and π_0 to remove the blinding. We also provide an alternate construction of ApplyPerm that avoids using inverse permutation gates in Appendix F but it incurs double the permutation overhead.

All together, building these protocols as arithmetic permutation circuits allows arguing security by reasoning only about the correctness of the circuit (e.g., that it correctly implements sorting) and the simulatability of the reconstruct gates (e.g., the reconstructed blinded permutation appears random to the adversary). It separates the need to argue about the simulatability of the implementation of the arithmetic and permutation gates.

F Alternate Circuit for Arbitrary Permutation Application

Parties can avoid using inverse permutation shuffles for Sort as in ApplyPerm (Figure 25) by using our alternate circuit construction ApplyPermAlt. Instead, parties need to preprocess eight forward permutation shuffle tuples (four for π_0 and four for π_1), as opposed to four total in ApplyPerm (two forward and two inverse). The description for ApplyPermAlt is provided in Figure 27.

Protocol $\mathsf{PermNetSTPrep}[\mathbb{F}, \mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}}]$

$\mathsf{P}_b.\mathsf{Main}(\mathsf{in}_{\pi_b}) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Main}(\mathsf{in}_{\pi_{1-b}})$:

1. Parse in_π:
   - $[(\mathsf{id}_{\pi,b,i}, n_{\pi,b,i}, n_{\pi,inv,b,i}, [\kappa_{i,j} \in \mathbb{F}]_j^{n_{\pi,b,i}}, [\underline{\kappa}_{i,j} \in \mathbb{F}]_j^{n_{\pi,inv,b,i}}, \pi_{b,i} \in \mathsf{Perm}(m_{b,i}))]_i^{n_{\pi,b}} \leftarrow \mathsf{in}_{\pi_b}$
   - $[(\mathsf{id}_{\pi,1-b,i}, n_{\pi,inv,1-b,i}, [\vec{a}_{i,j}, \vec{b}_{i,j} \in \mathbb{F}^{m_{1-b,i}}]_j^{n_{\pi,1-b,i}}, [\underline{\vec{a}}_{i,j}, \underline{\vec{b}}_{i,j} \in \mathbb{F}^{m_{1-b,i}}]_j^{n_{\pi,inv,1-b,i}})]_i^{n_{\pi,1-b}} \leftarrow \mathsf{in}_{\pi_{1-b}}$
2. Generate multiplication triples and input tuples via SPDZ preprocessing [KOS16, KPR18]:
   - $p_m, p_{inp,b}, p_{inp,1-b} \leftarrow 0$
   - $\forall (n,\ m) \in \{(n_{\pi,b,i}, m_b),\ (n_{\pi,inv,b,i}, m_b),\ (n_{\pi,1-b,i}, m_{1-b}),\ (n_{\pi,inv,1-b,i}, m_{1-b})\}$,
     (a) $p_m \leftarrow p_m + n\big(m\log_2 m + m + 1\big)$
     (b) $p_{inp,b} \leftarrow p_{inp,b} + n\big(16m\log_2 m + 14m + 4\big) + 2m\log_2 m - 2m + 4$
     (c) $p_{inp,1-b} \leftarrow p_{inp,b}$
   - P_0 and P_1 invoke $\mathsf{F}_{\mathsf{SPDZPrep}}(n_{inp,b} = p_{inp,b},\ n_{inp,1-b} = p_{inp,1-b},\ n_m = p_m)$ for each P_b to receive (see Fig. 12):
     (a) $\big[\big(r_{i,b}, \langle r_{i,b}\rangle_b = (\llbracket r_{i,b}\rrbracket_b, \llbracket \gamma r_{i,b}\rrbracket_b)\big)\big]_i^{n_{inp,b}}$
     (b) $\big[\big(\langle r_{i,1-b}\rangle_b = (\llbracket r_{i,1-b}\rrbracket_b, \llbracket \gamma r_{i,1-b}\rrbracket_b)\big)\big]_i^{n_{inp,1-b}}$
     (c) $[(\langle a_i\rangle_b, \langle b_i\rangle_b, \langle c_i\rangle_b)]_i^{n_m}$
     (d) $\llbracket\gamma\rrbracket_b$
3. Compute and authenticate shuffle tuples for each $b' \in \{b,\ 1-b\}$:
   - $\forall i \in [n_{\pi,b'}]$, $\mathsf{P}_{b'}$ and $\mathsf{P}_{1-b'}$ invoke $\mathsf{P}_{b'}.\mathsf{ShufTupGen}([\pi_i]_j^{n_{\pi,b',i}}) \leftrightarrow \mathsf{P}_{1-b'}.\mathsf{ShufTupGen}(\big[\vec{a}_{i,j}, \vec{b}_{i,j}\big]_j^{n_{\pi,b',i}})$ to receive permuted shares of $\vec{a}_{i,j}$ and $\vec{b}_{i,j}$ (Fig. 20):
     (a) $\left[\left(\langle\vec{\tilde{a}}_{i,j}\rangle_{b'}, \langle\vec{\tilde{b}}_{i,j}\rangle_{b'}\right)\right]_j^{n_{\pi,b',i}}$
     (b) $\left[\left(\langle\vec{\tilde{a}}_{i,j}\rangle_{1-b'}, \langle\vec{\tilde{b}}_{i,j}\rangle_{1-b'}\right)\right]_j^{n_{\pi,b',i}}$
   - $\forall i \in [n_{\pi,b'}]$, $\mathsf{P}_{b'}$ and $\mathsf{P}_{1-b'}$ invoke $\mathsf{P}_{b'}.\mathsf{ApplyKappaPrep}(\left[\left(\langle\vec{\tilde{a}}_{i,j}\rangle_{b'}, \langle\vec{\tilde{b}}_{i,j}\rangle_{b'}, \kappa_{i,j}\right)\right]_j^{n_{\pi,b',i}}) \leftrightarrow \mathsf{P}_{1-b'}.\mathsf{ApplyKappaPrep}(\left[\left(\langle\vec{\tilde{a}}_{i,j}\rangle_{1-b'}, \langle\vec{\tilde{b}}_{i,j}\rangle_{1-b'}\right)\right]_j^{n_{\pi,b',i}})$ to receive κ-authenticated shares of $\vec{\tilde{a}}_{i,j}$ and $\vec{\tilde{b}}_{i,j}$ (Fig. 22):
     (a) $T_{shf,b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i, \kappa_{i,j}, \langle\vec{\tilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\tilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$
     (b) $T_{shf,1-b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{a}_{i,j}, \vec{b}_{i,j}, \langle\vec{\tilde{a}}_{i,j}\rangle_{1-b'}^{(\kappa)}, \langle\vec{\tilde{b}}_{i,j}\rangle_{1-b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$
4. Compute and authenticate inverse shuffle tuples for each $b' \in \{b,\ 1-b\}$:
   - $\forall i \in [n_{\pi,b'}]$, $\mathsf{P}_{b'}$ and $\mathsf{P}_{1-b'}$ invoke $\mathsf{P}_{b'}.\mathsf{InvShufTupGen}(\big[[\langle c_i\rangle]_i^k\big]_j^{n_{\pi,inv,b',i}}) \leftrightarrow \mathsf{P}_{1-b'}.\mathsf{InvShufTupGen}(\big[\underline{\vec{a}}_{i,j}, \underline{\vec{b}}_{i,j}\big]_j^{n_{\pi,inv,b',i}})$ to receive permuted shares of $\underline{\vec{a}}_{i,j}$ and $\underline{\vec{b}}_{i,j}$ (Fig. 21):
     (a) $\left[\left(\langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{b'}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{b'}\right)\right]_j^{n_{\pi,inv,b',i}}$
     (b) $\left[\left(\langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{1-b'}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{1-b'}\right)\right]_j^{n_{\pi,inv,b',i}}$
   - $\forall i \in [n_{\pi,b'}]$, $\mathsf{P}_{b'}$ and $\mathsf{P}_{1-b'}$ invoke $\mathsf{P}_{b'}.\mathsf{ApplyKappaPrep}(\left[\left(\langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{b'}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{b'}, \underline{\kappa}_{i,j}\right)\right]_j^{n_{\pi,inv,b',i}}) \leftrightarrow \mathsf{P}_{1-b'}.\mathsf{ApplyKappaPrep}(\left[\left(\langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{1-b'}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{1-b'}\right)\right]_j^{n_{\pi,inv,b',i}})$ to receive $\underline{\kappa}$-authenticated shares of $\vec{\underline{\tilde{a}}}_{i,j}$ and $\vec{\underline{\tilde{b}}}_{i,j}$:
     (a) $T_{invshf,b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i^{-1}, \underline{\kappa}_{i,j}, \langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{b'}^{(\underline{\kappa})}\right)\right]_j^{n_{\pi,inv,b',i}}$
     (b) $T_{invshf,1-b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\underline{\vec{a}}_{i,j}, \underline{\vec{b}}_{i,j}, \langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{1-b'}^{(\underline{\kappa})}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{1-b'}^{(\underline{\kappa})}\right)\right]_j^{n_{\pi,inv,b',i}}$
5. Return
   - $(T_{shf,0,0}, T_{shf,0,1}, T_{invshf,0,0}, T_{invshf,0,1})$ to P_0.
   - $(T_{shf,1,0}, T_{shf,1,1}, T_{invshf,1,0}, T_{invshf,1,1})$ to P_1.

Figure 19: Protocol for preprocessing of authenticated shuffle tuples consumed in FLOSS in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid.

$\mathsf{P}_b.\mathsf{ShufTupGen}([\pi]_j^n) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{ShufTupGen}(\left[\vec{a}_j, \vec{b}_j\right]_j^n)$:

1. Prepare the permutation network circuit, which is independent of the shuffle-tuple vectors:
   - P_b locally parses π into topologically-ordered control bits $(c_1, \cdots, c_k) \in \{0,1\}^k$ for switching network of input length $m = |\pi|$ [BD02].
   - P_b creates authenticated input shares of its control bits: $\forall i \in [k], \langle c_i \rangle \leftarrow \mathsf{P}_b.\mathsf{Input}(\langle c_i \rangle_b, c_j) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{RcvInput}(\langle c_i \rangle_{1-b})$
   - $\forall i \in [k]$, P_b and P_{1-b} compute and reconstruct $\langle c_i \rangle \cdot (1 - \langle c_i \rangle)$. If $\langle c_i \rangle \cdot (1 - \langle c_i \rangle) \neq 0$, output `abort`.
   - $\forall j \in [n]$, P_b and P_{1-b} generate data and authenticated blinds for each wire in a and b's switching network: for $e \in E$, $\langle r_{a,e,j} \rangle, \langle l_{a,e,j} \rangle, \langle r_{b,e,j} \rangle, \langle l_{b,e,j} \rangle \leftarrow \mathsf{P}_b.\mathsf{Coin}(4) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Coin}(4)$
   - $\forall j \in [n]$, P_b and P_{1-b} generate one-time authentication key for a and b: $\langle \iota_{a,j} \rangle, \langle \iota_{b,j} \rangle \leftarrow \mathsf{P}_b.\mathsf{Coin}(2) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Coin}(2)$
   - $\forall j \in [n]$, P_b and P_{1-b} compute the following, then reconstruct to obtain $s_{a,in_0,j}$, $s_{a,in_1,j}$, $s_{b,in_0,j}$, and $s_{b,in_1,j}$:
     (a) $\langle s_{a,in_0,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{a,out_0,j} \rangle - \langle r_{a,in_0,j} \rangle) + \langle c_i \rangle \cdot (\langle r_{a,out_1,j} \rangle - \langle r_{a,in_0,j} \rangle)$
     (b) $\langle s_{a,in_1,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{a,out_1,j} \rangle - \langle r_{a,in_1,j} \rangle) + \langle c_i \rangle \cdot (\langle r_{a,out_0,j} \rangle - \langle r_{a,in_1,j} \rangle)$
     (c) $\langle s_{b,in_0,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{b,out_0,j} \rangle - \langle r_{b,in_0,j} \rangle) + \langle c_i \rangle \cdot (\langle r_{b,out_1,j} \rangle - \langle r_{b,in_0,j} \rangle)$
     (d) $\langle s_{b,in_1,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle r_{b,out_1,j} \rangle - \langle r_{b,in_1,j} \rangle) + \langle c_i \rangle \cdot (\langle r_{b,out_0,j} \rangle - \langle r_{b,in_1,j} \rangle)$
   - P_b and P_{1-b} compute the following, then reconstruct to obtain $\Delta_{a,in_0,j}$, $\Delta_{a,in_1,j}$, $\Delta_{b,in_0,j}$, and $\Delta_{b,out_1,j}$:
     (a) $\langle \Delta_{a,in_0,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{a,out_0,j} \rangle - \langle l_{a,in_0,j} \rangle) + \langle c_i \rangle \cdot (\langle l_{a,out_1,j} \rangle - \langle l_{a,in_0,j} \rangle) + s_{a,in_0,j} \cdot \langle \iota_{a,j} \rangle$
     (b) $\langle \Delta_{a,in_1,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{a,out_1,j} \rangle - \langle l_{a,in_1,j} \rangle) + \langle c_i \rangle \cdot (\langle l_{a,out_0,j} \rangle - \langle l_{a,in_1,j} \rangle) + s_{a,in_1,j} \cdot \langle \iota_{a,j} \rangle$
     (c) $\langle \Delta_{b,in_0,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{b,out_0,j} \rangle - \langle l_{b,in_0,j} \rangle) + \langle c_i \rangle \cdot (\langle l_{b,out_1,j} \rangle - \langle l_{b,in_0,j} \rangle) + s_{b,in_0,j} \cdot \langle \iota_{b,j} \rangle$
     (d) $\langle \Delta_{b,out_1,j} \rangle \leftarrow (1 - \langle c_i \rangle) \cdot (\langle l_{b,out_1,j} \rangle - \langle l_{b,in_1,j} \rangle) + \langle c_i \rangle \cdot (\langle l_{b,out_0,j} \rangle - \langle l_{b,in_1,j} \rangle) + s_{b,in_1,j} \cdot \langle \iota_{b,j} \rangle$
2. Prepare inputs for the permutation network circuit on the authenticated input shares:
   - $\forall j \in [n]$, P_{1-b} creates authenticated input shares of its randomized vectors $(\vec{a}_j, \vec{b}_j)$:
     (a) $(\langle \vec{a}_j \rangle, \langle \vec{b}_j \rangle) \leftarrow \mathsf{P}_{1-b}.\mathsf{Input}((\langle \vec{a}_j \rangle_{1-b}, \langle \vec{b}_j \rangle_{1-b}),\ (\vec{a}_j, \vec{b}_j)) \leftrightarrow \mathsf{P}_b.\mathsf{RcvInput}((\langle \vec{a}_j \rangle_b, \langle \vec{b}_j \rangle_b))$
   - $\forall j \in [n]$, P_b and P_{1-b} prepare wire values for each paired input index and input wire index $(e \in [m], i_e)$ for the switching network, and then reconstruct to obtain $t_{a,i_e,j}$ and $t_{b,i_e,j}$:
     (a) $\langle t_{a,i_e,j} \rangle \leftarrow \langle \vec{a}_{e,j} \rangle + \langle r_{a,i_e,j} \rangle$
     (b) $\langle t_{b,i_e,j} \rangle \leftarrow \langle \vec{b}_{e,j} \rangle + \langle r_{b,i_e,j} \rangle$
   - $\forall j \in [n]$, P_b and P_{1-b} prepare authenticated wire values for each input wire index i_e for the switching network, and then reconstruct to obtain $\tau_{a,i_e,j}$ and $\tau_{b,i_e,j}$:
     (a) $\langle \tau_{a,i_e,j} \rangle \leftarrow \langle l_{a,i_e,j} \rangle + t_{a,i_e,j} \cdot \langle \iota_{a,j} \rangle$
     (b) $\langle \tau_{b,i_e,j} \rangle \leftarrow \langle l_{b,i_e,j} \rangle + t_{b,i_e,j} \cdot \langle \iota_{b,j} \rangle$
3. P_b iterates through all the switches topologically indexed for each $i \in [k]$ in the network to evaluate the output wires:
   - $in_0, in_1 \leftarrow \mathsf{In}(v_i)$
   - $out_0, out_1 \leftarrow \mathsf{Out}(v_i)$
   - $t_{a,out_{c_i},j} \leftarrow s_{a,in_0,j} + t_{a,in_0,j}$
   - $t_{a,out_{1-c_i},j} \leftarrow s_{a,in_1,j} + t_{a,in_1,j}$
   - $\tau_{b,out_{c_i},j} \leftarrow \Delta_{b,in_0,j} + \tau_{b,in_0,j}$
   - $\tau_{b,out_{1-c_i},j} \leftarrow \Delta_{b,in_1,j} + \tau_{b,in_1,j}$
4. For each paired output index and output wire index $(e \in [m], o_e)$, P_b and P_{1-b} perform authentication check with t and τ:
   - P_b sends output wire values $t_{a,o_e,j}$, $t_{b,o_e,j}$ and authenticated output wire values $\tau_{a,o_e,j}$, $\tau_{b,o_e,j}$ to P_{1-b}.
   - P_b and P_{1-b} compute the following, then reconstruct to obtain $\gamma_{a,e,j}$ and $\gamma_{b,e,j}$:
     (a) $\langle \gamma_{a,e,j} \rangle \leftarrow \langle l_{a,o_e,j} \rangle + t_{a,o_e,j} \cdot \langle \iota_{a,j} \rangle$
     (b) $\langle \gamma_{b,e,j} \rangle \leftarrow \langle l_{b,o_e,j} \rangle + t_{b,o_e,j} \cdot \langle \iota_{b,j} \rangle$
   - P_b and P_{1-b} compute $\gamma_{a,e,j} - \tau_{a,o_e,j}$ and $\gamma_{b,e,j} - \tau_{b,o_e,j}$, If either are $\neq 0$, output `abort`.
5. P_b and P_{1-b} combine shares from output wires indices o_e to return authenticated shuffle tuples:
   - $[\langle \vec{\tilde{a}}_j \rangle_b, \langle \vec{\tilde{b}}_j \rangle_b]_j^n \leftarrow [[t_{a,o_e,j} - \langle r_{a,o_e,j} \rangle_b]_{o_e}, [[t_{b,o_e,j} - \langle r_{b,o_e,j} \rangle_b]]_{o_e}]_j^n$
   - $[\langle \vec{\tilde{a}}_j \rangle_{1-b}, \langle \vec{\tilde{b}}_j \rangle_{1-b}]_j^n \leftarrow [[-\langle r_{a,o_e,j} \rangle_{1-b}]_{o_e}, [-\langle r_{b,o_e,j} \rangle_{1-b}]_{o_e}]_j^n$

Figure 20: Subprotocol for generating forward authenticated shuffle tuples in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid.

$\mathsf{P}_b.\mathsf{InvShufTupGen}([[\langle c_i\rangle]_i^k]_j^n) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{InvShufTupGen}(\left[\vec{\mathrm{a}}_j, \vec{b}_j\right]_j^n)$:

1. Prepare the permutation network circuit, which is independent of the shuffle-tuple vectors:
   - P_b and P_{1-b} reuse their control bit shares $[\langle c_i\rangle]_i^k$ from $\mathsf{ShufTupGen}$.
   - $\forall i \in [k]$, P_b and P_{1-b} compute and reconstruct $\langle c_i\rangle \cdot (1 - \langle c_i\rangle)$. If $\langle c_i\rangle \cdot (1 - \langle c_i\rangle) \neq 0$, output `abort`.
   - $\forall j \in [n]$, P_b and P_{1-b} generate data and authenticated blinds for each wire in a and b's switching network: for $e \in E$, $\langle r_{a,e,j}\rangle, \langle l_{a,e,j}\rangle, \langle r_{b,e,j}\rangle, \langle l_{b,e,j}\rangle \leftarrow \mathsf{P}_b.\mathsf{Coin}(4) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Coin}(4)$
   - $\forall j \in [n]$, P_b and P_{1-b} generate one-time authentication key for a and b: $\langle \iota_{a,j}\rangle, \langle \iota_{b,j}\rangle \leftarrow \mathsf{P}_b.\mathsf{Coin}(2) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Coin}(2)$
   - $\forall j \in [n]$, P_b and P_{1-b} compute the following, then reconstruct to obtain $s_{a,out_0,j}$, $s_{a,out_1,j}$, $s_{b,out_0,j}$, and $s_{b,out_1,j}$:
     (a) $\langle s_{a,out_0,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle r_{a,in_0,j}\rangle - \langle r_{a,out_0,j}\rangle) + \langle c_i\rangle \cdot (\langle r_{a,in_1,j}\rangle - \langle r_{a,out_0,j}\rangle)$
     (b) $\langle s_{a,out_1,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle r_{a,in_1,j}\rangle - \langle r_{a,out_1,j}\rangle) + \langle c_i\rangle \cdot (\langle r_{a,in_0,j}\rangle - \langle r_{a,out_1,j}\rangle)$
     (c) $\langle s_{b,out_0,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle r_{b,in_0,j}\rangle - \langle r_{b,out_0,j}\rangle) + \langle c_i\rangle \cdot (\langle r_{b,in_1,j}\rangle - \langle r_{b,out_0,j}\rangle)$
     (d) $\langle s_{b,out_1,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle r_{b,in_1,j}\rangle - \langle r_{b,out_1,j}\rangle) + \langle c_i\rangle \cdot (\langle r_{b,in_0,j}\rangle - \langle r_{b,out_1,j}\rangle)$
   - P_b and P_{1-b} compute the following, then reconstruct to obtain $\Delta_{a,out_0,j}$, $\Delta_{a,out_1,j}$, $\Delta_{b,out_0,j}$, and $\Delta_{b,out_1,j}$:
     (a) $\langle \Delta_{a,out_0,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle l_{a,in_0,j}\rangle - \langle l_{a,out_0,j}\rangle) + \langle c_i\rangle \cdot (\langle l_{a,in_1,j}\rangle - \langle l_{a,out_0,j}\rangle) + s_{a,out_0,j} \cdot \langle \iota_{a,j}\rangle$
     (b) $\langle \Delta_{a,out_1,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle l_{a,in_1,j}\rangle - \langle l_{a,out_1,j}\rangle) + \langle c_i\rangle \cdot (\langle l_{a,in_0,j}\rangle - \langle l_{a,out_1,j}\rangle) + s_{a,out_1,j} \cdot \langle \iota_{a,j}\rangle$
     (c) $\langle \Delta_{b,out_0,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle l_{b,in_0,j}\rangle - \langle l_{b,out_0,j}\rangle) + \langle c_i\rangle \cdot (\langle l_{b,in_1,j}\rangle - \langle l_{b,out_0,j}\rangle) + s_{b,out_0,j} \cdot \langle \iota_{b,j}\rangle$
     (d) $\langle \Delta_{b,out_1,j}\rangle \leftarrow (1 - \langle c_i\rangle) \cdot (\langle l_{b,in_1,j}\rangle - \langle l_{b,out_1,j}\rangle) + \langle c_i\rangle \cdot (\langle l_{b,in_0,j}\rangle - \langle l_{b,out_1,j}\rangle) + s_{b,out_1,j} \cdot \langle \iota_{b,j}\rangle$
2. Prepare inputs for the permutation network circuit on the authenticated input shares:
   - $\forall j \in [n]$, P_{1-b} creates authenticated input shares of its randomized vectors $(\vec{\mathrm{a}}_j, \vec{b}_j)$:
     (a) $(\langle\vec{\mathrm{a}}_j\rangle, \langle\vec{b}_j\rangle) \leftarrow \mathsf{P}_{1-b}.\mathsf{Input}((\langle\vec{\mathrm{a}}_j\rangle_{1-b}, \langle\vec{b}_j\rangle_{1-b}),\ (\vec{\mathrm{a}}_j, \vec{b}_j)) \leftrightarrow \mathsf{P}_b.\mathsf{RcvInput}((\langle\vec{\mathrm{a}}_j\rangle_b, \langle\vec{b}_j\rangle_b))$
   - $\forall j \in [n]$, P_b and P_{1-b} prepare wire values for each paired output index and output wire index $(e \in [m], o_e)$ for the switching network, and then reconstruct to obtain $t_{a,i_e,j}$ and $t_{b,i_e,j}$:
     (a) $\langle t_{a,o_e,j}\rangle \leftarrow \langle\vec{\mathrm{a}}_{e,j}\rangle + \langle r_{a,o_e,j}\rangle$
     (b) $\langle t_{b,o_e,j}\rangle \leftarrow \langle\vec{b}_{e,j}\rangle + \langle r_{b,o_e,j}\rangle$
   - $\forall j \in [n]$, P_b and P_{1-b} prepare authenticated wire values for each output wire index o_e for the switching network, and then reconstruct to obtain $\tau_{a,i_e,j}$ and $\tau_{b,i_e,j}$:
     (a) $\langle \tau_{a,o_e,j}\rangle \leftarrow \langle l_{a,o_e,j}\rangle + t_{a,o_e,j} \cdot \langle \iota_{a,j}\rangle$
     (b) $\langle \tau_{b,o_e,j}\rangle \leftarrow \langle l_{b,o_e,j}\rangle + t_{b,o_e,j} \cdot \langle \iota_{b,j}\rangle$
3. P_b iterates through all the switches reverse-topologically indexed for each $i \in [k]$ in the network to evaluate the input wires:
   - $in_0, in_1 \leftarrow \mathsf{In}(v_i)$
   - $out_0, out_1 \leftarrow \mathsf{Out}(v_i)$
   - $t_{a,in_{c_i},j} \leftarrow s_{a,out_0,j} + t_{a,out_0,j}$
   - $t_{a,in_{1-c_i},j} \leftarrow s_{a,out_1,j} + t_{a,out_1,j}$
   - $\tau_{b,in_{c_i},j} \leftarrow \Delta_{b,out_0,j} + \tau_{b,in_0,j}$
   - $\tau_{b,in_{1-c_i},j} \leftarrow \Delta_{b,out_1,j} + \tau_{b,in_1,j}$
4. For each paired input index and input wire index $(e \in [m], i_e)$, P_b and P_{1-b} perform authentication check with t and τ:
   - P_b sends input wire values $t_{a,i_e,j}$, $t_{b,i_e,j}$ and authenticated input wire values $\tau_{a,i_e,j}$, $\tau_{b,i_e,j}$ to P_{1-b}.
   - P_b and P_{1-b} compute the following, then reconstruct to obtain $\gamma_{a,e,j}$ and $\gamma_{b,e,j}$:
     (a) $\langle \gamma_{a,e,j}\rangle \leftarrow \langle l_{a,i_e,j}\rangle + t_{a,i_e,j} \cdot \langle \iota_{a,j}\rangle$
     (b) $\langle \gamma_{b,e,j}\rangle \leftarrow \langle l_{b,i_e,j}\rangle + t_{b,i_e,j} \cdot \langle \iota_{b,j}\rangle$
   - P_b and P_{1-b} compute $\gamma_{a,e,j} - \tau_{a,i_e,j}$ and $\gamma_{b,e,j} - \tau_{b,i_e,j}$, If either are $\neq 0$, output `abort`.
5. P_b and P_{1-b} combine shares from input wires indices i_e to return authenticated inverse shuffle tuples:
   - $[\langle\underline{\vec{\mathrm{a}}}_j\rangle_b, \langle\underline{\vec{b}}_j\rangle_b]_j^n \leftarrow [[t_{a,i_e,j} - \langle r_{a,i_e,j}\rangle_b]_{i_e},\ [t_{b,i_e,j} - \langle r_{b,i_e,j}\rangle_b]_{i_e}]_j^n$
   - $[\langle\underline{\vec{\mathrm{a}}}_j\rangle_{1-b}, \langle\underline{\vec{b}}_j\rangle_{1-b}]_j^n \leftarrow [[-\langle r_{a,i_e,j}\rangle_{1-b}]_{i_e},\ [-\langle r_{b,i_e,j}\rangle_{1-b}]_{i_e}]_j^n$

Figure 21: Subprotocol for generating inverse authenticated shuffle tuples in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid.

$\mathsf{P}_b.\mathsf{ApplyKappaPrep}(\left[\langle\vec{\widetilde{a}}_j\rangle_b,\ \langle\vec{\widetilde{b}}_j\rangle_b,\ \kappa_j\right]_j^n) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{ApplyKappaPrep}(\left[\langle\vec{\widetilde{a}}_j\rangle_{1-b},\ \langle\vec{\widetilde{b}}_j\rangle_{1-b}\right]_j^n)$:

1. P_b creates authenticated input share of its κ value: $\forall j \in [n],\ \langle\kappa_j\rangle \leftarrow \mathsf{P}_b.\mathsf{Input}(\langle\kappa_j\rangle_b,\ \kappa_j) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{RcvInput}(\langle\kappa_j\rangle_{1-b})$
2. P_b and P_{1-b} compute:
   - $\forall j \in [n],\ \langle\kappa_j \cdot \vec{\widetilde{a}}_j\rangle \leftarrow \langle\vec{\widetilde{a}}_j\rangle \cdot \langle\kappa_j\rangle$
   - $\forall j \in [n],\ \langle\kappa_j \cdot \vec{\widetilde{b}}_j\rangle \leftarrow \langle\vec{\widetilde{b}}_j\rangle \cdot \langle\kappa_j\rangle$
3. P_b and P_{1-b} each perform blind authentication check with respect to γ on computed outputted shares:
   - $\langle\alpha\rangle \leftarrow \mathsf{P}_b.\mathsf{Coin}(1) \leftrightarrow \mathsf{P}_{1-b}.\mathsf{Coin}(1)$
   - Invoke $\mathsf{BlindAuth}([\![\gamma]\!], \langle\alpha\rangle, [\langle\vec{\widetilde{a}}_j\rangle || \langle\vec{\widetilde{b}}_j\rangle || \langle\kappa_j \cdot \vec{\widetilde{a}}_j\rangle || \langle\kappa_j \cdot \vec{\widetilde{b}}_j\rangle]_j^n)$
   - Retrieve deferred multiplications $[(z_i, [\![\gamma]\!]z_i)]_i$ and invoke $\mathsf{Auth}([\![\gamma]\!], [(z_i, [\![\gamma]\!]z_i)]_i)$
4. P_b and P_{1-b} output $[\langle\vec{\widetilde{a}}_j\rangle^{(\kappa_j)}]_j^n \leftarrow [(\langle\vec{\widetilde{a}}_j\rangle, \langle\kappa_j \cdot \vec{\widetilde{a}}_j\rangle)]_j^n$ and $[\langle\vec{\widetilde{b}}_j\rangle^{(\kappa_j)}]_j^n \leftarrow [(\langle\vec{\widetilde{b}}_j\rangle, \langle\kappa_j \cdot \vec{\widetilde{b}}_j\rangle)]_j^n$

Figure 22: Subprotocol for applying and auth-checking κ-authenticated shares to shuffle tuples in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid.

$\mathsf{P}_0.\mathsf{Sort}(\left[[\langle x_{i,j}\rangle_0]_j^{\upsilon}\right]_i^n \in \mathbb{F}^{\upsilon\times n}) \leftrightarrow \mathsf{P}_1.\mathsf{Sort}(\left[[\langle x_{i,j}\rangle_1]_j^{\upsilon}\right]_i^n \in \mathbb{F}^{\upsilon\times n})$

Input: Shared vector of LSB to MSB bit decomposition of $\langle x\rangle$.

Output: Sorting permutation $\langle\sigma\rangle$ for x.

1. $\langle\rho^{(1)}\rangle \leftarrow \mathsf{BitSortInv}([\langle x_{1,i}\rangle]_i^n)$
2. $\langle\sigma^{(1)}\rangle \leftarrow \mathsf{ApplyPermInv}(\langle\rho^{(1)}\rangle, [n])$
3. For $j = 2$ to υ:
   - $\langle x'^{(j)}\rangle \leftarrow \mathsf{ApplyPerm}(\langle\sigma^{(j-1)}\rangle, [\langle x_{j,i}\rangle]_i^n)$
   - $\langle\rho^{(j)}\rangle \leftarrow \mathsf{BitSortInv}(\langle x'^{(j)}\rangle)$
   - $\langle\sigma^{(j)}\rangle \leftarrow \mathsf{ApplyPermInv}(\langle\rho^{(j)}\rangle, \langle\sigma^{(j-1)}\rangle)$
4. Output $\langle\sigma^{(\upsilon)}\rangle$

Figure 23: An arithmetic permutation circuit for radix-sorting secret shares of a bit-decomposed vector $\vec{x}$. Each element of $\vec{x}$ is represented using υ bits.

$\mathsf{P}_0.\mathsf{BitSortInv}(\langle\vec{b}\rangle_0 \in \mathbb{F}^n) \leftrightarrow \mathsf{P}_1.\mathsf{BitSortInv}(\langle\vec{b}\rangle_1 \in \mathbb{F}^n)$

Input: Shares of a vector of bits $\langle\vec{b}\rangle$.

Output: Inverse of stable sorting permutation $\langle\rho\rangle$ for the bits $\vec{b}$.

1. $\langle t\rangle \leftarrow \mathsf{Input}(\mathsf{P}_0,\ 0)$
2. Define bit indicators: $\langle f_0\rangle, \langle f_1\rangle \leftarrow \vec{1}_n - \langle\vec{b}\rangle, \langle\vec{b}\rangle$
3. Initialize prefix sum vectors: $\langle s_0\rangle, \langle s_1\rangle \leftarrow \vec{0}_n, \vec{0}_n$
4. For $j = 1$ to n:
   - $\langle t\rangle \leftarrow \langle t\rangle + \langle f_{0,j}\rangle$
   - $\langle s_{0,j}\rangle \leftarrow \langle t\rangle$
5. For $j = 1$ to n:
   - $\langle t\rangle \leftarrow \langle t\rangle + \langle f_{1,j}\rangle$
   - $\langle s_{1,j}\rangle \leftarrow \langle t\rangle$
6. $\langle\rho^{-1}\rangle \leftarrow \langle\vec{b}\rangle \cdot (\langle s_1\rangle - \langle s_0\rangle) + \langle s_0\rangle$
7. Output $\langle\rho^{-1}\rangle$

Figure 24: An arithmetic permutation circuit for producing the inverse sorting permutation (e.g., the destination encoding) for secret shares of a vector of bits.

$\mathsf{P}_0.\mathsf{ApplyPerm}(\langle\rho\rangle_0 \in \mathbb{F}^n, \langle x\rangle_0 \in \mathbb{F}^n)$
$\leftrightarrow \mathsf{P}_1.\mathsf{ApplyPerm}(\langle\rho\rangle_1 \in \mathbb{F}^n, \langle x\rangle_1 \in \mathbb{F}^n)$

Input: Shares of permutation $\rho \in \mathbb{S}_n$ as $\langle\rho([n])\rangle$, shares of input $\langle x\rangle$.
Output: Shares of permutation applied to inputs $\langle\rho(x)\rangle$.
Assume: id_{π_b} identifies a random permutation $\pi_b \leftarrow\!\!\$\, \mathbb{S}_n$ for P_b.

1. $\langle\pi_0(\rho)\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_0}, \langle\rho\rangle)$
2. $\langle\pi_1(\pi_0(\rho))\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_1}, \langle\pi_0(\rho)\rangle)$
3. $\pi_1(\pi_0(\rho)) \leftarrow \mathsf{Reconstruct}(\langle\pi_1(\pi_0(\rho))\rangle)$
4. Parse $\pi_1(\pi_0(\rho))$ as $(\pi_1 \circ \pi_0 \circ \rho) \in \mathbb{S}_n$
5. Locally permute shares with public reconstructed permutation:
$\langle(\pi_1 \circ \pi_0 \circ \rho)(x)\rangle \leftarrow (\pi_1 \circ \pi_0 \circ \rho)(\langle x\rangle)$
6. $\langle(\pi_0 \circ \rho)(x)\rangle \leftarrow \mathsf{PermuteInv}(\mathsf{id}_{\pi_1}, \langle(\pi_1 \circ \pi_0 \circ \rho)(x)\rangle)$
7. $\langle\rho(x)\rangle \leftarrow \mathsf{PermuteInv}(\mathsf{id}_{\pi_0}, \langle(\pi_0 \circ \rho)(x)\rangle)$
8. Output $\langle\rho(x)\rangle$

Figure 25: A description of an arithmetic permutation circuit for applying an arbitrary secret-shared permutation ρ to a vector of data $\vec{x}$. This is a useful subcircuit in many protocols including sorting.

$\mathsf{P}_0.\mathsf{ApplyPermInv}(\langle\rho\rangle_0 \in \mathbb{F}^n, \langle x\rangle_0 \in \mathbb{F}^n)$
$\leftrightarrow \mathsf{P}_1.\mathsf{ApplyPermInv}(\langle\rho\rangle_1 \in \mathbb{F}^n, \langle x\rangle_1 \in \mathbb{F}^n)$

Input: shares of permutation $\langle\rho\rangle$, shares of input $\langle x\rangle$.
Output: shares of inverse permutation applied to inputs $\langle\rho^{-1}(x)\rangle$.

1. $\pi_b \leftarrow\!\!\$\, \mathsf{Perm}(n)$
2. $\mathsf{Prep}((\mathsf{id}_{\pi_b},\ n_{\pi_b} = 2,\ n_{\pi_b,\,inv} = 0,\ \pi_b))$
3. $\langle\pi_0(\rho)\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_0}, \langle\rho\rangle)$
4. $\langle\pi_1(\pi_0(\rho))\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_1}, \langle\pi_0(\rho)\rangle)$
5. $\pi_1 \circ \pi_0 \circ \rho \leftarrow \mathsf{Reconstruct}(\langle\pi_1(\pi_0(\rho))\rangle)$
6. $\rho^{-1} \circ \pi_0^{-1} \circ \pi_1^{-1} \leftarrow (\pi_1 \circ \pi_0 \circ \rho)^{-1}$
7. $\langle\pi_0(x)\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_0}, \langle x\rangle)$
8. $\langle\pi_1(\pi_0(x))\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_1}, \langle\pi_0(x)\rangle)$
9. $\langle\rho^{-1}(x)\rangle \leftarrow (\rho^{-1} \circ \pi_0^{-1} \circ \pi_1^{-1})(\langle\pi_1(\pi_0(x))\rangle)$
10. Output $\langle\rho^{-1}(x)\rangle$

Figure 26: An arithmetic permutation circuit for applying the inverse of an arbitrary secret-shared permutation ρ to a vector of data $\vec{x}$.

$\mathsf{P}_0.\mathsf{ApplyPermAlt}(\langle\rho\rangle_0 \in \mathbb{F}^n, \langle x\rangle_0 \in \mathbb{F}^n)$
$\leftrightarrow \mathsf{P}_1.\mathsf{ApplyPermAlt}(\langle\rho\rangle_1 \in \mathbb{F}^n, \langle x\rangle_1 \in \mathbb{F}^n)$

Input: Shares of permutation $\rho \in \mathbb{S}_n$ as $\langle\rho([n])\rangle$, shares of input $\langle x\rangle$.
Output: Shares of permutation applied to inputs $\langle\rho(x)\rangle$.
Assume: id_{π_b} identifies a random permutation $\pi_b \leftarrow\!\!\$\, \mathbb{S}_n$ for P_b.

1. $\langle\rho^{-1}([n])\rangle \leftarrow \mathsf{ApplyPermInv}(\langle\rho\rangle,\ [n])$
2. $\langle\pi_1 \circ \rho^{-1}\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_1}, \langle\rho^{-1}([n])\rangle)$
3. $\langle\pi_0 \circ \pi_1 \circ \rho^{-1}\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_0}, \langle\pi_1 \circ \rho^{-1}\rangle)$
4. $\pi_0 \circ \pi_1 \circ \rho^{-1} \leftarrow \mathsf{Reconstruct}(\langle\pi_0 \circ \pi_1 \circ \rho^{-1}\rangle)$
5. $\rho \circ \pi_1^{-1} \circ \pi_0^{-1} \leftarrow (\pi_0 \circ \pi_1 \circ \rho^{-1})^{-1}$
6. $\langle\pi_1(x)\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_1}, \langle x\rangle)$
7. $\langle(\pi_0 \circ \pi_1)(x)\rangle \leftarrow \mathsf{Permute}(\mathsf{id}_{\pi_0}, \langle\pi_1(x)\rangle)$
8. Locally permute shares with public reconstructed permutation:
$\langle\rho(x)\rangle \leftarrow (\rho \circ \pi_1^{-1} \circ \pi_0^{-1})(\langle(\pi_0 \circ \pi_1)(x)\rangle)$
9. Output $\langle\rho(x)\rangle$

Figure 27: Alternate arithmetic permutation circuit description for ApplyPerm that avoids using the PermuteInv gate.