

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

FLOSS: Fast Linear Online Secret-Shared Shuffling

Ian Chang and Sela Navot, *University of Washington;* Alex Ozdemir, *Max Planck Institute for Security & Privacy, Georgia Tech;* Nirvan Tyagi, *University of Washington*

https://www.usenix.org/conference/usenixsecurity26/presentation/chang-ian

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

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.

1 Introduction

Randomly permuting private data is a core building block in constructing privacy-preserving protocols for analytics [5, 7, 9, 34, 50, 53], advertising [18], communication [3, 27, 49], whistleblowing [45, 48], random-access memory [28, 30, 61], contact discovery [37, 39, 43, 51], analysis of social graphs [6, 15, 44], 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 [60], is represented as an arithmetic circuit. Parties use an appropriate MPC protocol to evaluate the arithmetic circuit over the data shares [12]. 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. [20] 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* [11], 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. [20] 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 [32] and advertising [55]. 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 [1] 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 [23], there is growing interest in supporting more complex statistics like heavy hitters [7] and attribution [18]. Our work is directly applicable to enabling such statistics (that rely on shuffling for efficiency) in these standards.

Prior work has attempted lifting shuffle tuples to the malicious setting, but achieving proper security has proved to be

Protocol	Offline costs		Online costs	
MPC-Enc+Shuf [52]	$O(\lambda n)\,\mathbb{E}$	–	$O(\lambda n)\,\mathbb{E}$	–
Linear-Only LHE [26]	$O(n)$	–	$O(\lambda n)\,\mathbb{L}$	–
SimpleNet [60]	$O(n \log n)$	532s	$O(n \log n)$	491s
PermNet [52]	$O(n \log n)$	2436s	$O(n \log n)$	407s
OPM [59]	$O(nT \log \frac{n}{T})$	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 $T = 2^4$.

challenging. Initial proposed approaches to construct "authenticated" shuffle tuples [27, 47] 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. [59]. Song et al. [59] 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. [59]. 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 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 [7, 29, 38] that is itself a

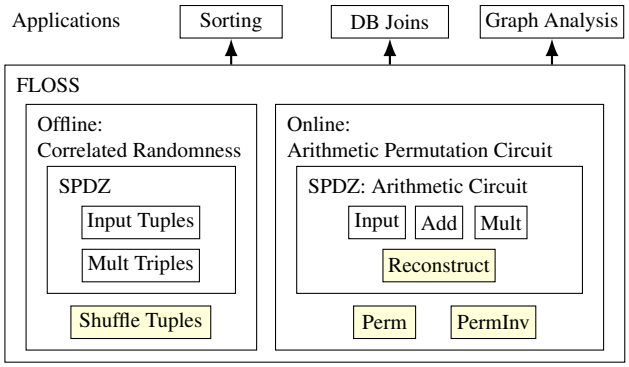


Figure 1: Overview of the FLOSS architecture for malicious-secure 2PC of arithmetic permutation circuits. Applications are built on top of the arithmetic permutation circuit abstraction. FLOSS uses an online/offline architecture in which correlated randomness is generated in an input-independent offline phase and consumed to compute gates in the online phase. FLOSS builds on the SPDZ [25] 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.

building block for many analytics applications [5, 7, 9, 18]. We adapt the popular radix-sort-and-shuffle protocol [7, 13, 14, 21] 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 full version of the paper [19] for more details on the secret-sharing scheme employed by SPDZ [25]. 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 [16]. 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 [33], 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 [22]. 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 [24]). 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 [25] 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 [25]. 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. [20, 56] who proposed a novel preprocessing technique for generating correlated randomness referred to as *shuffle tuples*. Shuffle tuples encode

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

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.

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 $\vec{a} \in \mathbb{F}^n$ selected by P_1 and an arbitrary permutation π held by P_0. The shuffle tuple consists of fresh shares of $\vec{\tilde{a}} = \pi(\vec{a})$, such that P_0 holds π and $[\![\vec{\tilde{a}}]\!]_0$ and P_1 holds $\vec{a}$ and $[\![\vec{\tilde{a}}]\!]_1$.

Constructing the shuffle tuple still requires permuting secret shares of $\vec{a}$ 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 $\vec{a}$ to blind their shares, $\vec{c} \leftarrow [\![\vec{x}]\!]_1 - \vec{a}$, and sends blinded shares $\vec{c}$ to P_0. Party P_1 simply sets their new shares for $[\![\pi(\vec{x})]\!]_1 \leftarrow [\![\vec{\tilde{a}}]\!]_1$. Party P_0 applies permutation π to set their new shares as

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

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

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

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 [25]. 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, $\vec{a}$ and $\vec{b}$. Party P_0 holds $(\pi, [\![\vec{\tilde{a}}]\!]_0, [\![\vec{\tilde{b}}]\!]_0)$ and P_1 holds $(\vec{a}, \vec{b}, [\![\vec{\tilde{a}}]\!]_1, [\![\vec{\tilde{b}}]\!]_1)$. To shuffle $\langle\vec{x}\rangle = ([\![\vec{x}]\!], [\![\gamma\vec{x}]\!])$, parties undergo the same procedure as above, using $\vec{a}$ to shuffle the 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 [27, 47] 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. [59] 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. [59] propose a new approach to address this selec-

[1] The general attack strategy had also been previously described by Araki et al. [6].

tive 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. [59] 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. [59] employ the honest-but-curious oblivious puncturable matrix (OPM) preprocessing algorithm of Chase et al. [20] 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. [59]. 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 [25] 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 $\kappa \leftarrow\!\!\$\ \mathbb{F}$. 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 $([\![\vec{x}]\!], [\![\gamma\vec{x}]\!])$. The authenticated shares $[\![\gamma\vec{x}]\!]$ are handled separately. As before, P_1 sends blinded shares

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

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 $([\![\vec{v}]\!]_1, [\![\gamma\vec{v}]\!]_1) \leftarrow \langle\vec{v}\rangle_1$:
$\vec{c} \leftarrow [\![\vec{v}]\!]_1 - \vec{a}$ $\quad$ $\vec{d} \leftarrow [\![\gamma\vec{v}]\!]_1 - \vec{b}$;
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.

$\vec{c} \leftarrow [\![\vec{x}]\!]_1 - \vec{a}$ 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. [59]. 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 [25] (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 $\langle\vec{a}\rangle \leftarrow (\llbracket\vec{a}\rrbracket, \llbracket\gamma\vec{a}\rrbracket)$. 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 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 $\llbracket\vec{\tilde{a}}\rrbracket$ and $\llbracket\kappa\vec{\tilde{a}}\rrbracket$ 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. [52] and Dittmer et al. [26] both propose linear cost protocols that rely on encryption schemes with homomorphic properties. Mohassel et al. [52] use MPC to encode and encrypt secret data into El Gamal elliptic curve ciphertexts [57] and then have the permuting party verifiably shuffle the ciphertexts using a proof of knowledge [10, 35, 36, 54]. Dittmer et al. [26] 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 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 [60] 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. [51, 52] introduce a permutation-encoding technique that avoids running the permutation network as a $(\log n)$-depth circuit. Song et al. [59], 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 [52] 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 [59] 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 [25]) 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 9) 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 [8, 18, 50] and histogram computation [5, 7]. We demonstrate how to express the efficient radix-sort-and-shuffle protocol of Asharov et al. [7] 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

[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.

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. [7] propose a neat approach to instantiating this subprotocol using random permutations that we observe can be preprocessed (see the full version of the paper [19]). 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 the full version of the paper [19]). 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. [52] permutation network approach (or any circuit-description permutation network) can be easily adapted to compute correlated inverse shuffle tuples 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 the full version of the paper [19], 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) [24] model), where the interactive functionality interface exposes black-box access to compute arithmetic "gates". Wire values in the circuit are represented by identifier *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 the full version of the paper [19]), 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 9).

- $\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$: 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 [25] 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 [25], 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. [52] (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 11.

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. [52] construction for malicious-secure permutation networks which itself builds on the SPDZ malicious-secure arithmetic circuit computation protocol [25], 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 perform 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 the full version of the paper [19]. We provide the following corollary from the security proof of the original permutation network protocol of Mohassel et al. [52], on which our construction is based:

Corollary 1 (Informal)**.** *The protocol* $\mathsf{PermNetSTPrep}[\mathbb{F}]$ *(Figure 12) UC-realizes the ideal functionality* $\mathsf{F}_{\mathsf{ShufTuplePrep}}$ *(Figure 11) 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 [25] 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 to Appendix B. The complete security proof is in the full version of the paper [19]. Our main result is captured by the following theorem:

Theorem 2 (Informal)**.** *The protocol* $\mathsf{FLOSS}[\mathbb{F}]$ *(Figure 10) UC-realizes the ideal functionality* $\mathsf{F}_{\mathsf{ArithPermCirc}}[\mathbb{F}]$ *(Figure 9) 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 the full version of the paper [19].

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 $\$0.93/\text{hr}$, 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 [25] for arithmetic circuits (see full version of the paper [19]). Our implementation of arithmetic permutation circuits uses relevant SPDZ gates/gadgets like addition and multiplication and extends to include permutation gates (see Figure 10). 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 [2].

To generate SPDZ multiplication triples and input tuples, we use the MP-SPDZ library [2] implementations of MASCOT [41] and LowGear [42]. 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 11 and full version of the paper [19]), 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. [52]). We select the Waksman network [60] 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.[3]

Shuffling baselines. We compare to three baselines for

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

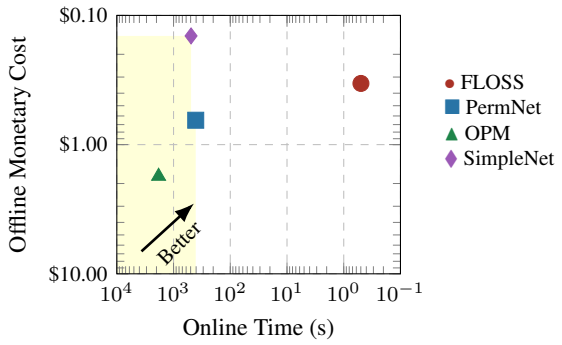


Figure 4: Cost summary of protocols for shuffling 2^{20} 128-bit field entries on two machines (25Gbps, single-digit ms latency); OPM uses $T = 2^4$. FLOSS has $\geq 800\times$ faster online time than prior protocols, and $2\times$ cheaper offline monetary cost than prior protocols outside of SimpleNet.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

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. [52]. 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. [20] honest-but-curious shuffle tuple construction proposed by Song et al. [59]. 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 [59] 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. 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 [7], Shuffle-then-Sort [38], and a sorting network [40].

	Shuffle			
Sorting algorithm	FLOSS	OPM	PermNet	SimpleNet
Radix-Sort-and-Shuffle [7]	✓		✓	✓
Shuffle-then-Sort [38]	✓	✓	✓	✓

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

Both Radix-Sort-and-Shuffle and Shuffle-then-Sort rely on a shuffling subroutine. We describe the Radix-Sort-and-Shuffle algorithms in the full version of the paper [19] as an arithmetic permutation circuit. Similarly, Shuffle-then-Sort can also be described 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 [40] implemented in MP-SPDZ [2] 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 [4].

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

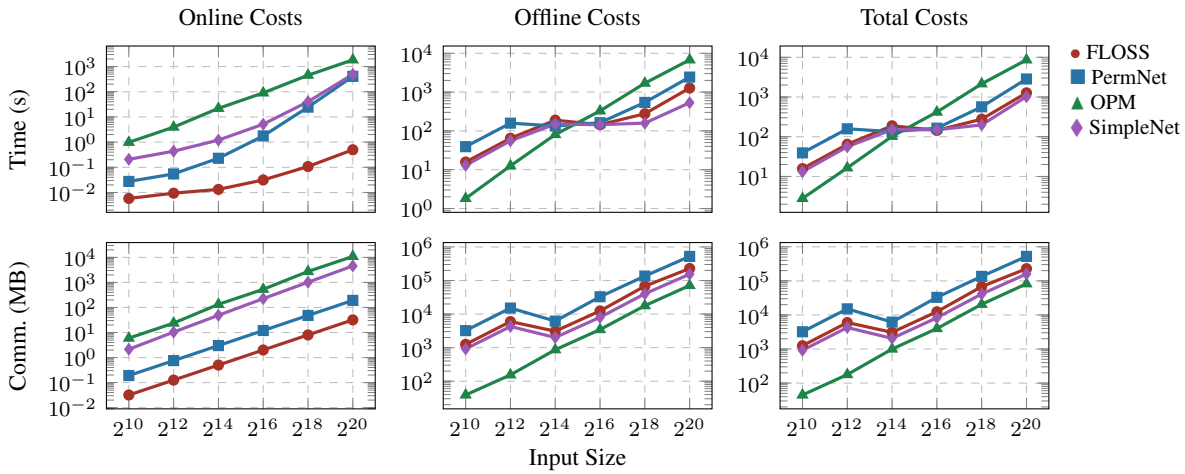


Figure 5: Computation time and communication costs for shuffling 128-bit field entries on two machines (25Gbps, single-digit ms latency) varying from 2^{10} to 2^{20} broken down by online and offline phases; OPM uses $T = 2^4$. FLOSS has lower online costs than all baselines.

FLOSS 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 [4, 46]. 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.

	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 $\approx 2\times$ less SPDZ preprocessing than PermNet.

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 sum-

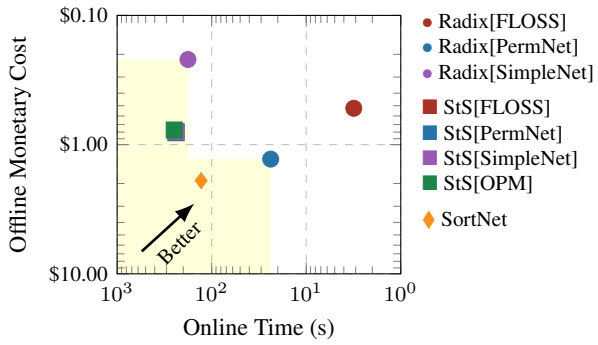


Figure 6: Cost summary of protocols for sorting 2^{13} 32-bit decomposed values on two machines (25Gbps, single-digit ms latency); OPM uses $T = 2^4$. Radix-Sort-and-Shuffle instantiated with FLOSS has $\geq 5\times$ faster online time and $1.5\times$ cheaper offline monetary cost than prior protocols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

marizes 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 $O(\upsilon \cdot n)$, while all other evaluated online shuffles incur $O(\upsilon \cdot n \log n)$ 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 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 Additional Related Work

Gao et al. [31] and MOSAC [58] are concurrent closely related works. They construct authenticated shuffle tuples in the malicious-secure MPC setting using similar high-level strategies. The Gao et al. shuffle [31], as observed by [58], is vulnerable to a permutation-guessing selective failure attack. However, they provide techniques to support more than two parties which while insecure may be recovered in future work. MOSAC [58] incurs the use of an additional polynomial-based permutation check that FLOSS avoids. As such, FLOSS results in a more concretely efficient shuffle.

Lastly, ORQ [9] is a recent system targeting MPC private analytics implementing protocols for database queries and complex joins. ORQ similarly observes using correlated inverse shuffle tuples as an optimization. However, ORQ makes use of shuffle tuples in the honest-but-curious dishonest majority setting, whereas we work in the malicious 2PC dishonest majority setting. The protocols of ORQ may be built on the arithmetic permutation circuit abstraction, so FLOSS could lift their work to the malicious-secure 2PC setting.

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

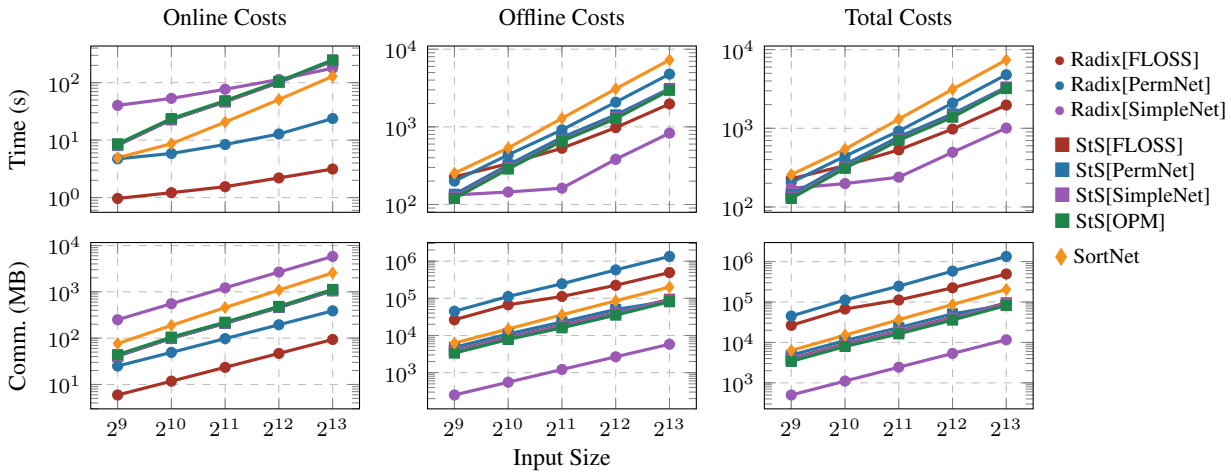


Figure 7: Computation time and communication costs for sorting 32-bit decomposed values on two machines (25Gbps, single-digit ms latency) varying from 2^9 to 2^{13} broken down by online and offline phases; OPM uses $T = 2^4$. FLOSS when combined with the Radix-Sort-and-Shuffle algorithm has lower online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

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.[4] 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 [59].
- Scripts to run and plot the aforementioned experiments.

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

References

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

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

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

[4] Amazon Web Services. Specifications for Amazon EC2 compute optimized instances, 2026. URL: https://docs.aws.amazon.com/ec2/latest/instancetypes/co.html.

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

[6] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure graph analysis at scale. In *CCS*, 2021.

[7] 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.

[8] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal.

Secret-shared joins with multiplicity from aggregation trees. In *CCS*, 2022.

[9] 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.

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

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

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

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

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

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

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

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

[18] 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.

[19] Ian Chang, Sela Navot, Alex Ozdemir, and Nirvan Tyagi. FLOSS: Fast linear online secret-shared shuffling. Cryptology ePrint Archive, Paper 2026/672, 2026. URL: https://eprint.iacr.org/2026/672.

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

[21] 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.

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

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

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

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

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

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

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

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

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

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

[32] 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. URL: https://datatracker.ietf.org/doc/draft-ietf-ppm-dap/14/.

[33] Shafi Goldwasser and Yehuda Lindell. Secure multiparty computation without agreement. *J. Cryptol.*, 2005.

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

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

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

[37] 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.

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

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

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

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

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

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

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

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

[46] 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. URL: https://browser.geekbench.com/v5/cpu/14437445.

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

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

[49] 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.

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

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

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

[53] 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.

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

[55] 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. URL: https://www.w3.org/TR/privacy-preserving-attribution/.

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

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

[58] Xiangfu Song, Xiaojian Liang, Ye Dong, Jianli Bai, Pu Duan, Changyu Dong, Tianwei Zhang, and Ee-Chien Chang. Secret-shared shuffle from authenticated correlations. *IACR ePrint Archive*, Paper 2026/566, 2026.

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

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

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

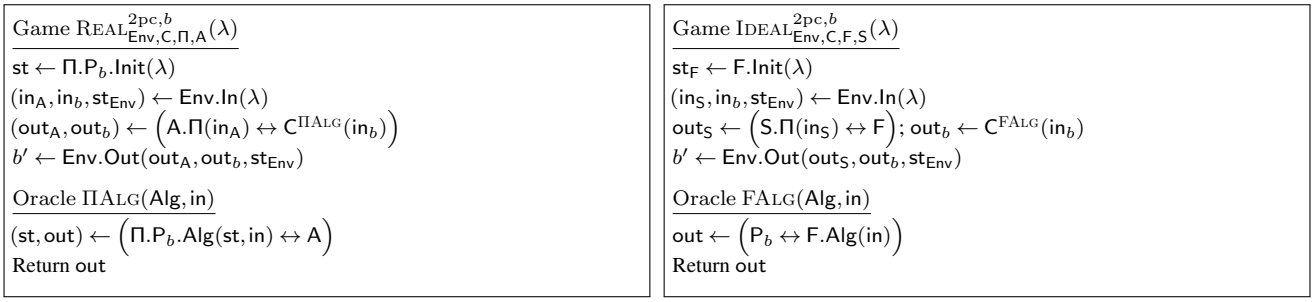


Figure 8: Two-party UC security definition.

A Security Definition

We use Universal Composability (UC) [16] to prove the security of all our protocols. In particular, we adapt the Simplified UC framework (SUC) of Canetti et al. [17]. 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:*

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

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*:*

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

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

B FLOSS Protocol for Arithmetic Permutation Circuits

The ideal functionality $\mathsf{F}_{\mathsf{ArithPermCirc}}$ for arithmetic permutation circuit computation is given in Figure 9, the FLOSS protocol realizing this functionality is given in Figure 10, the shuffle tuple preprocessing functionality $\mathsf{F}_{\mathsf{ShufTuplePrep}}$ and protocol PermNetSTPrep are given in Figures 11 and 12. The helper subprotocols (ShufTupGen, InvShufTupGen, ApplyKappaPrep) for PermNetSTPrep are provided in the full version of the paper [19].

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 $(\mathtt{prep}, n_{i,0}, n_{i,1}, n_m, \mathsf{in}_\pi)$ 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 $\mathsf{op} = \mathtt{prep}$.
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 $\mathtt{abort}$ to all parties.
5. Else,
   - $ictr_b \leftarrow n_{i,b}$ and $mctr \leftarrow n_m$, $prep \leftarrow 1$.
   - $\forall i \in [n_{\pi,b'}]\ T_{\pi,b'}[\mathsf{id}_{\pi,b',i}] \leftarrow (n_{\pi,b',i}, n_{\pi,inv,b',i}, \pi_{b',i}, m_{b',i})$
6. Send $(\mathtt{prep})$ to all parties.

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

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{F}_{\mathsf{ArithCirc}}[\mathbb{F}]$ (see full version of the paper [19]).

$\mathsf{Permute}([\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ 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 $(\mathtt{perm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ 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 $\mathsf{op} = \mathtt{perm}$.
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
   - $(\mathtt{perm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ to P_b.
   - $(\mathtt{perm}, [\mathsf{id}'_i]_i^m, [\widetilde{\mathsf{id}}'_i]_i^m, \mathsf{id}'_\pi, b'')$ 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 $(\mathtt{invperm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ 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 $\mathsf{op} = \mathtt{invperm}$.
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
   - $(\mathtt{invperm}, [\mathsf{id}_i]_i^m, [\widetilde{\mathsf{id}}_i]_i^m, \mathsf{id}_\pi, b')$ to P_b.
   - $(\mathtt{invperm}, [\mathsf{id}'_i]_i^m, [\widetilde{\mathsf{id}}'_i]_i^m, \mathsf{id}'_\pi, b'')$ to P_{1-b}.

Figure 9: 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 full version of the paper [19]):
   - $L_{inp,b'} \leftarrow \left[\left(r_{i,b'}, \langle r_{i,b'}\rangle_{b'}\right)\right]_i^{n_{inp,b'}}$
   - $L_{inp,1-b'} \leftarrow \left[\left(\langle r_{i,1-b'}\rangle_{b'}\right)\right]_i^{n_{inp,1-b'}}$
   - $L_{mul} \leftarrow [(\langle a_i\rangle_{b'}, \langle b_i\rangle_{b'}, \langle c_i\rangle_{b'})]_i^{n_m}$
   - $[\![\gamma]\!]_{b'}$
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. 11):
   - $T_{shf,b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i, \kappa_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$
   - $T_{shf,1-b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{a}_{i,j}, \vec{b}_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,1-b',i}}$
   - $T_{invshf,b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i^{-1}, \underline{\kappa}_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$
   - $T_{invshf,1-b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{\underline{a}}_{i,j}, \vec{\underline{b}}_{i,j}, \langle\vec{\widetilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\widetilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,1-b',i}}$
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 full version of the paper [19]).

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

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see full version of the paper [19]).

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

Same as $\mathsf{SPDZ}[\mathbb{F}]$ (see full version of the paper [19]).

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

1. Require $\mathsf{id} \in T$.
2. P_0 and P_1 run authentication check on all pending multiplication opens in L_{auth} (see full version of the paper [19]):
   (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 (see full version of the paper [19]):
   $\mathsf{P}_0.\mathsf{Auth}([\![\gamma]\!], [(v, [\![\gamma v]\!])]) \leftrightarrow \mathsf{P}_1.\mathsf{Auth}([\![\gamma]\!], [(v, [\![\gamma v]\!])])$
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 10: 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 $b' \in \{b, 1-b\}$:
   - $\forall i \in [n_{\pi,b'}],\ T_{shf,b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i, \kappa_{i,j}, \langle\vec{\tilde{a}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\tilde{b}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$
   - $\forall i \in [n_{\pi,b'}],\ T_{shf,1-b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\vec{a}_{i,j}, \vec{b}_{i,j}, \langle\vec{\tilde{a}}_{i,j}\rangle_{1-b'}^{(\kappa)}, \langle\vec{\tilde{b}}_{i,j}\rangle_{1-b'}^{(\kappa)}\right)\right]_j^{n_{\pi,b',i}}$

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

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

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

   (b) Create $\underline{\kappa}$-authenticated shares of $\vec{\underline{\tilde{a}}}_{i,j}$ and $\vec{\underline{\tilde{b}}}_{i,j}$:
   - $\langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{b'} \leftarrow\!\!\$\ (\mathbb{F}^2)^{m_{1-b,i}}$, $\langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{1-b'} \leftarrow (\vec{\underline{\tilde{a}}}_{i,j} - [\![\vec{\underline{\tilde{a}}}_{i,j}]\!]_{b'}, \underline{\kappa}_{i,j}\vec{\underline{\tilde{a}}}_{i,j} - [\![\underline{\kappa}_{i,j}\vec{\underline{\tilde{a}}}_{i,j}]\!]_{b'})$
   - $\langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{b'} \leftarrow\!\!\$\ (\mathbb{F}^2)^{m_{1-b,i}}$, $\langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{1-b'} \leftarrow (\vec{\underline{\tilde{b}}}_{i,j} - [\![\vec{\underline{\tilde{b}}}_{i,j}]\!]_{b'}, \underline{\kappa}_{i,j}\vec{\underline{\tilde{b}}}_{i,j} - [\![\underline{\kappa}_{i,j}\vec{\underline{\tilde{b}}}_{i,j}]\!]_{b'})$
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 11: Functionality of preprocessing of authenticated shuffle tuples consumed in FLOSS for fast online permutations.

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 [41, 42]:
   - $p_m, p_{inp,b}, p_{inp,1-b} \leftarrow 0$
   - $\forall (n,\, m) \in \{(n_{\pi,b,i}, m_b),\ (n_{\pi,inv,b,i}, m_b),\ (n_{\pi,1-b,i}, m_{1-b}),\ (n_{\pi,inv,1-b,i}, m_{1-b})\}$,
     (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 full version of the paper [19]):
     (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 $\vec{b}_{i,j}$ (see full version of the paper [19]):
     (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 $\vec{\tilde{b}}_{i,j}$ (see full version of the paper [19]):
     (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 $\underline{\vec{b}}_{i,j}$ (see full version of the paper [19]):
     (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) $T_{invshf,b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\pi_i^{-1}, \underline{\kappa}_{i,j}, \langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{b'}^{(\kappa)}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{b'}^{(\kappa)}\right)\right]_j^{n_{\pi,inv,b',i}}$
     (b) $T_{invshf,1-b',b'}[\mathsf{id}_{\pi,b',i}] \leftarrow \left[\left(\underline{\vec{a}}_{i,j}, \underline{\vec{b}}_{i,j}, \langle\vec{\underline{\tilde{a}}}_{i,j}\rangle_{1-b'}^{(\kappa)}, \langle\vec{\underline{\tilde{b}}}_{i,j}\rangle_{1-b'}^{(\kappa)}\right)\right]_j^{n_{\pi,inv,b',i}}$
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 12: Protocol for preprocessing of authenticated shuffle tuples consumed in FLOSS in the $(\mathsf{F}_{\mathsf{SPDZPrep}}, \mathsf{F}_{\mathsf{Comm}})$-hybrid.