UM-PSO: A Unified Multi-Party Framework for Private Set Operations against a Dishonest Majority

Yaxi Yang*, Xiaojian Liang†, Weizhan Jing‡, Ye Dong§✉, Xiangfu Song*, Fangyuan Sun¶, Pu Duan†, Tianwei Zhang*
*Nanyang Technological University, †Ant International, Ant Group, ‡Zhongguancun Laboratory,
§National University of Singapore, ¶Qingdao University
Email: {yxyangjnu, im.liangxj, jingweizhan97}@gmail.com, dongye@nus.edu.sg,
{bintasong, sakgofish}@gmail.com, p.duan@ant-intl.com, tianwei.zhang@ntu.edu.sg

***Abstract*—Private Set Operations (PSO) enable mutually untrusted parties to securely compute arbitrary functions (e.g., union, intersection, and cardinality) over their private input sets. These operations have wide applications in many real-world scenarios. Existing PSO protocols fall short of practical deployment for several reasons. (1) *Function-specific*. Real-world privacy-preserving applications often require multiple set operations within the same task, while existing solutions typically address individual functionalities (e.g., intersection or union) in isolation, making it difficult and costly to support diverse set operations in a unified and efficient manner. (2) *Lacking malicious security*. As PSO is commonly employed in highly sensitive applications, it is often necessary to provide strong security guarantees against malicious adversaries. Unfortunately, most existing works only achieve semi-honest security, which limits their practical applicability. (3) *Restricted settings*. The majority of existing works focus exclusively on the two-party setting. How to extend them securely and efficiently to the multi-party setting while tolerating a malicious majority remains unclear. To date, designing a maliciously secure multi-party PSO (mPSO) framework that efficiently supports diverse set operations remains an open challenge.**

This paper presents the *first* maliciously secure mPSO framework, named UM-PSO, that supports a broad range of set operations with practical efficiency. At the core of our framework is a function-independent preprocessing phase that prepares a reusable pool of secret-shared items, which can then be leveraged to securely compute diverse set functionalities in the online phase. To achieve malicious security efficiently, we design verification mechanisms on top of SPDZ-based authenticated secret sharing, along with tailored techniques and optimizations to further improve practical performance. We implement our protocols and report concrete performance results. For a representative setting with 5 parties and a total of 2^{12} 128-bit items, our framework achieves an online running time of 0.627 seconds and incurs 3.35 MB of communication. Compared to the baselines, our framework achieves up to $51\times$ speedup and up to a $76\times$ reduction in communication.

I. Introduction

Secure multi-party computation (MPC) enables multiple mutually untrusted parties to jointly compute over their private datasets while revealing nothing beyond what is implied by the prescribed output. Within this broad area, multi-party Private Set Operations (mPSO) constitute a fundamental line of research: given inputs from n $(n > 2)$ parties, the goal is to compute arbitrary set functionalities over inputs without revealing any additional information, while revealing only the prescribed result to the parties [1–3].

mPSO encompasses a broad family of functionalities over multiple input sets. The first category is multi-party Private Set Intersection (mPSI) [1–8], which outputs items common to all parties. Beyond the basic mPSI functionality, several widely studied variants exist. One variant reveals only the intersection size, known as mPSI with cardinality (mPSI-CA) [2–4, 8]. Another variant evaluates an arbitrary function f over the intersection set without revealing the intersection items themselves (circuit-mPSI) [3, 4]. A further extension is to identify items that appear in at least d of n input sets (quorum PSI) [6, 9]. In parallel to mPSI, another line of work studies multi-party Private Set Union (mPSU) [1–3, 10–12], which outputs the union of all parties' items. Similarly, it has two common variants: mPSU with cardinality (mPSU-CA) [2, 3] and circuit-mPSU [2, 3].

A. Towards Unified mPSO

Most existing multi-party set computation protocols [1, 3, 5, 10–12] are designed for a single functionality (e.g., mPSI or mPSU), or at best a narrow subset of the broader mPSO family, and cannot be readily extended to other functionalities. To compute the next set functionality, parties need to re-input their private datasets and re-execute the entire protocol from scratch.

This naive strategy severely hampers the generality and reusability of current mPSO approaches in practice, and may cause privacy concerns. Some notable real-world examples have been highlighted in previous works [13, 14]: in database applications, particularly in medical data collaboration, different clinics may need to identify common patients or construct the union of their patient records, and then perform downstream analyses on the resulting data, such as computing average test values or aggregate statistics on certain types of genetic mutations. Such workflows may require repeatedly invoking circuit-PSI/PSU on the underlying databases. Another real-world example is Apple's 2021 proposal [15]. It

✉ Corresponding author

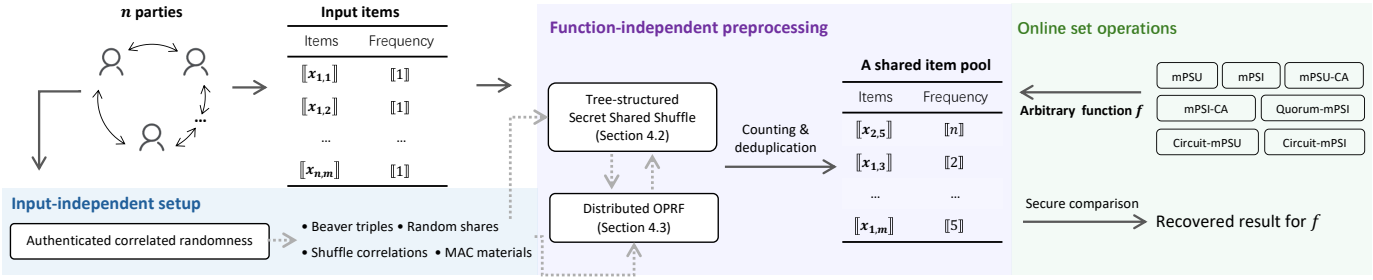

Fig. 1. System architecture of UM-PSO. $[\![\cdot]\!]$ represents the share of an item. Parties first submit item–payload pairs, where payload represents the frequency of each item. The function-independent preprocessing phase performs a tree-structured secret-shared shuffle and distributed OPRF evaluation to produce a deduplicated item pool annotated with item multiplicities. This pool can be reused across different functionalities. For a chosen functionality f, parties run lightweight secure comparisons over the pool to obtain the desired result, avoiding the need to rerun the entire protocol.

proposed using a PSI-based protocol to detect child sexual abuse material (CSAM) in iCloud Photos, but the project sparked widespread debate and was ultimately discontinued. In the proposed system, Apple needed to run (variants of) PSI operations to compare a database of known CSAM images against users' content stored in iCloud Photos. A critical concern raised by privacy advocates was that Apple could supply different input sets across users when executing set operations multiple times, potentially enabling the inference of additional information about legitimate user data.

Driven by practical needs, it is highly desirable to have a unified mPSO framework that supports a broad range of set functionalities without requiring parties to re-input their datasets or rerun the entire protocol. Such a framework can naturally enable continuous and iterative mPSI/mPSU-style queries [16]. Moreover, without requiring users to resubmit their input sets, the framework can ensure that the same inputs are used across different set operations, thereby mitigating the risk of information leakage caused by malicious parties supplying different inputs across executions.

Most mPSI/mPSU/mPSO protocols [3, 5, 10–12] follow the prevailing "align-program-obtain" paradigm to achieve better efficiency: parties use hashing techniques to *align* items into different bins, and then *program* items into pseudorandom values. Then they securely compare these values to derive a set of boolean indicators specifying whether each item x belongs to a given set. Finally, these boolean values are fed into Oblivious Transfer (OT) to *obtain* either the items absent from a party's own input set (mPSU) or only the items common to all parties (mPSI).

Nevertheless, protocols based on this paradigm suffer from two main limitations when integrated into a unified framework. i) *Distinct protocol designs hinder unification.* The *program* and *obtain* phases are typically designed in a function-specific manner, which is not reusable across different set functionalities. In mPSU, parties must (implicitly) obtain all items missing from their own input sets to reconstruct the global union. By contrast, mPSI returns only items that appear in all parties' sets, and therefore a party does not learn items held by other parties but absent from its own input set. These fundamental differences induce distinct computational structures, leading to separate, specialized technical designs. ii) *Phase inconsistency challenges the extension to malicious majority security.* Ensuring the consistency of intermediate results across the *program* and *obtain* phases is non-trivial. This issue is known as *the consistency problem* [17, 18]. This makes it difficult to extend existing protocols to the malicious security setting. A trivial combination with a commitment protocol to verify the consistency of intermediate results across different phases would incur significant overhead [19]. To date, most maliciously secure mPSI protocols [8, 20, 21] impose stronger assumptions (e.g., a trusted party or non-colluding parties) or are tailored to specific functionalities, making them infeasible to support other operations in the mPSO family. Moreover, maliciously secure mPSU protocols remain largely unexplored in the literature [22]. This naturally raises the following question:

> *Can we break out of the conventional design paradigm and construct a unified mPSO framework built on general MPC techniques (e.g., secret sharing or garbled circuits) that is function-independent and supports the full spectrum of operations in the mPSO family?*

A straightforward approach would be to build a unified mPSO framework on general MPC techniques, which would naturally inherit their strong security guarantees [23, 24], such as malicious security against up to $n-1$ corrupted parties, without relying on honest majority assumptions. However, this immediately raises a fundamental challenge: how can we unify mPSI and mPSU, the two logically and technically distinct problems with fundamentally different protocol structures, within a single framework?

B. Our Idea in a Nutshell

We observe that the union consists of items that appear in at least one input set, while the intersection consists of items that appear in all n input sets. Thus, we reformulate mPSO as a counting problem. We propose a novel design that securely computes the frequency of each item in the union of all input sets, which serves as a reusable pool during a *function-independent preprocessing* phase. Then, parties can

TABLE I

The comparison of various multi-party private set operation schemes. n denotes the number of parties. t represents the number of corrupted parties. Each party has m elements with ℓ-bit length, and M is the domain of the elements. λ and κ are statistical and computational security parameters, respectively.

Primitives	Protocols	Comm. cost/query		Comp. cost/query		Functionality	Corruption Threshold	Security Model
		Leader	Client	Leader	Client			
HE	KS05 [1]	$O(ntm(log\|M\|)\lambda)$		$O(ntm^2\lambda)$		PSO, ER†	$t<n$	semi-honest
	HV17 [25]	$O((n^2+nmlogm)\kappa)$	$O((n+mlogm)\kappa)$	$O(m^2\kappa)$		mPSI	$t<n$	malicious
GC	CDGO21 [6]	$O(nm(\kappa+\lambda+logm))$	$O(m(\kappa+\lambda+logm))$	$O(nm\kappa)$	$O(m\kappa)$	quorum PSI	$t<n/2$	semi-honest
	SCY24 [7]	$O(nm(logm)\ell)$		$O(nm(logm)\ell)$		mPSI	$t<n$	semi-honest
OT	KMPR17 [5]	$O(nm(\lambda+\kappa+\log\ m))$	$O(tm(\lambda+\kappa+logm))$	$O(nm\kappa)$	$O(tm\kappa)$	mPSI	$t<n$	semi-honest
	NTY21 [20]	$O(mn(\lambda+\kappa+logm))$	$O(m(\lambda+\kappa+logm))$	$O(mn)$	$O(tm)$	mPSI	$t<n$	malicious‡
	LG23[10]	$O((\ell+n)nm)$	$O(\ell nm)$	$O(\ell n^2m)$	$O(\ell nm)$	mPSU	$t<n^*$	semi-honest
	GLWL24 [8]	$O((n+m)\kappa)$	$O(m\kappa)$	$O(nm\kappa)$	$O(m\kappa)$	mPSI, mPSI-CA	$t<n-1^*$	malicious‡
	DZBC25 [11]	$O(n^2m(\ell+\lambda+logn+logm))$		$O(n^2m)$		mPSU	$t<n-1$	semi-honest
	GNBT25 [12]	$O(\lambda+n^2m)$	$O(\lambda+nm)$	$O(n^2m)$	$O(nm)$	mPSU	$t<n$	semi-honest
	DCZB25 [3]	$O(n^2m(\ell+\lambda+logn+logm))$	$O(nm(\ell+\lambda+logn+logm))$	$O(n^2m)$	$O(nm)$	arbitrary functions	$t<n$	semi-honest
SS	BA12 [2]	$O(\lambda^2nmlogn+\lambda n^2+n^3)$		$O(\lambda nmlogn+n^2+n^3)$		mPSO, mPSO-CA, ER	$t<n$	semi-honest
	Ours	$O(n^2mlog(n))$		$O(nmlog(n))$		arbitrary functions	$t<n$	malicious majority

* There exists a designated pivot party that is assumed not to collude with any other party.
† Element Reduction (ER) with parameter d realizes the same functionality as quorum PSI with threshold d.
‡ The malicious security of this protocol relies on an OKVS construction instantiated in the random oracle model.

securely compare the frequency of each item with 1 or n according to the specific functionality in the online phase.

The core idea of our framework is shown in Fig. 1. We assume a set of n parties, where each party P_i holds an input set $\{x_{i,1},\cdots,x_{i,m}\}$. Each party associates a frequency tag $ct_{x_{i,j}} \leftarrow \text{Count}(x_{i,j})$ with every item $x_{i,j}(i\in[n],j\in[m])$, initialized to 1. In the preprocessing phase, parties securely compute the frequency of $x_{i,j}$ and store it as ct_x. To deduplicate input items and realize frequency computation, we combine a tree-structured secret-shared shuffle (tSSS) procedure with a distributed oblivious PRF (DOPRF) protocol. At the end of this phase, each distinct item appears only once, while retaining the frequency information in $ct_{x_{i,j}}$. In the phase of online set operations, parties only need to securely compare $ct_{x_{i,j}}$ and a function-dependent threshold thr.

- mPSI: $thr=n$, item in the intersection appears in all n input sets.
- mPSU: $thr=1$, item in the union appears at least once.
- Quorum PSI: $thr=d$ $(1<d<n)$, d is the quorum threshold.
- Circuit-mPSI/circuit-mPSU: parties do not directly recover the items; instead, they forward the corresponding secret-shared items to the subsequent evaluation of a circuit f.

Consequently, the preprocessing phase is executed once and can be reused across multiple online queries: different set functionalities are realized by changing only thr (or the downstream circuit), without rerunning the full protocol.

C. Contribution

We make the following contributions in this paper.

- **A Unified Design for mPSO.** We propose the first maliciously secure unified mPSO framework, UM-PSO, that supports a broad range of set functionalities, including mPSI/mPSU, quorum PSI, circuit-mPSI/circuit-mPSU. We reduce the set union and intersection problem to a counting problem. Combined with two function-independent subprotocols, tSSS and DOPRF, UM-PSO can securely count the frequency of each item in the union of all input sets, which serves as a reusable pool for securely computing different set functionalities in the online phase.
- **Reusable Function-independent Preprocessing.** Under UM-PSO, parties can launch multiple set-related computations after a *one-time function-independent preprocessing* phase. In the online phase, parties only need to perform a lightweight secure comparison to compute different functionalities over the same input sets.
- **Integration with MPC Frameworks.** All subprotocols in our mPSO framework are designed on top of a secret-sharing-based MPC framework. As a result, our construction integrates readily with other MPC-based protocols, broadening the practical applicability of mPSO [2].
- **Malicious-Majority Security.** UM-PSO is built on the SPDZ framework [23, 24], which employs information-theoretic message authentication codes (MACs) to ensure the integrity of all intermediate computations. It ensures the integrity of authenticated intermediate values and detects inconsistent deviations at any protocol step. Consequently, it securely realizes the corresponding functionality with abort against any static adversary corrupting up to $n-1$ parties.
- **Evaluation and Efficiency.** We compare UM-PSO against a baseline using trivial counting: all parties obliviously shuffle and sort all items, then perform pairwise comparisons to compute item frequencies. UM-PSO achieves significantly better efficiency and communication costs compared to this baseline, validating its practical viability. For 5 parties with 2^{12} total 128-bit items, UM-PSO achieves a set evaluation time of 0.627 seconds and 3.35 MB of communication. Compared to the baseline, UM-PSO achieves up to $51\times$ speedup and $76\times$ communication reduction.

II. Related Work

This section reviews state-of-the-art multi-party private set operation protocols, including mPSI and mPSU. Table I classifies typical protocols, and per-query cost refers to the online

cost after the one-time function-independent preprocessing; non-reusable protocols require a complete execution for each query. The technical details are described in Appendix F.

HE-based. KS05 [1] proposes the first construction of mPSO. It utilizes the polynomial representation to design a private set operation protocol based on Homomorphic Encryption (HE). This protocol requires many HE operations and is therefore inefficient in practice. Subsequently, HV17 [25] also follows the construction of polynomial evaluations and achieves a linear communication complexity in the size of input sets in the semi-honest setting.

GC-based. Huang et al. [4] propose a pure Garbled Circuit (GC) based PSI protocol. It takes two locally sorted sets and performs a sort-compare-shuffle GC to get an intersection set. Senate [26] and SCY24 [7] extend this solution to a multiparty setting. GC-based solutions rely on customized circuit constructions, which restrict their generality and lead to large circuits.

OT-based. Oblivious Transfer (OT) is the most widely used technique for multi-party private set operation protocols. A number of works [5, 8, 20, 27] propose multiparty PSI protocols. Specifically, KMPR17 [5] adheres to the *align-program-obtain* workflow. NTY21 [20] follows the idea and extends mPSI to be maliciously secure. GLWL24 [8] also utilizes *zero-sharing* based on Vector Oblivious Linear Evaluation (VOLE) and Oblivious Key-Value Store (OKVS). This protocol adopts the idea of *reducing mPSI to two-party PSI* and assumes two non-colluding parties: a leader and a pivot. To extend the protocol to malicious settings, it utilizes the random oracle, allowing the simulator to extract the adversary's inputs. It addresses only one class of malicious behavior: sending incorrect PRF values to the leader.

Besides mPSI, researchers have also investigated mPSU protocols [10, 11], which pose technical challenges distinct from those of mPSI. LG23 [10] takes a practical step toward multi-party PSU. While secure in the semi-honest setting, it relies on a trusted leader that is assumed not to collude with any other party. Then, the mPSO framework DCZB25 [3] adopts the design of DZBC25 [11] and instantiates the *align–program–obtain* pattern with a *zero-sharing* primitive. Due to its reliance on the alignment procedure and OT-based building blocks, this approach must redo the alignment and OT for every item, which compromises its generality and makes extension to the malicious security difficult.

SS-based. BA12 [2] redesigns the workflows for PSO, PSO-CA, and Element Reduction using only linear secret sharing (SS), instantiated through sorting, equality tests, and comparisons. After sorting, equal elements become adjacent: PSI is obtained by identifying adjacent duplicates, whereas PSU is obtained by eliminating them. This generic SS-based approach yields universally composable protocols that can serve as building blocks in larger secure computations. However, the reliance on purely SS-based sorting incurs substantial computational overhead, which hinders its practical deployment. Since no public implementation is available, we construct a baseline following their approach for comparison.

TABLE II
Notations used throughout this paper.

Notation	Description
P_i	Party P_i performing secure computation
κ	Computational security parameter; $\kappa = 128$
$\mathbf{x}_i$	Party P_i's input set of m distinct items
$\mathbf{x}_i[j] = x_{i,j}$	The j-th item $x_{i,j}$ in $\mathbf{x}_i$ with ℓ-bit length
$\mathbf{X}$	The tuple of input sets $\{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$
$\mathrm{Count}(x)$	The frequency of the item x
$[\![x]\!]$, $\langle x \rangle$, $\{\!\{x\}\!\}$	Arithmetic, Authenticated, Group sharings of $x \in \mathbb{F}_p$
$[m]$	The set $\{1, \cdots, m\}$

III. Preliminaries

In this section, we first describe the notations used throughout the paper as shown in Table II, followed by a brief introduction to the building blocks and the functionality definitions employed in our mPSO framework.

A. Security Model

Our protocols consider the multi-party setting and follow the malicious security definition [28]. We allow the adversary to corrupt multiple parties and combine their views to learn more information. The definition of malicious security is as follows.

Definition 1 (Multi-party Malicious Security)**.** *Assume that $\mathcal{F} : (\{0,1\}^*)^n \to (\{0,1\}^*)^n$ is a deterministic functionality with n inputs. In the ideal world, a trusted entity receives inputs from the parties, evaluates the ideal functionality $\mathcal{F}$ with defined leakage $\mathcal{L}$, and sends the output to the parties. Assume that an n-party protocol Π computes $\mathcal{F}$ in the real world. The view of i-th party during an execution of Π with input x_i is denoted as $\mathrm{View}_{\Pi,i}(x_i)$. An adversary $\mathcal{A}$ controlling corrupted parties $\mathcal{C}$ runs the protocol with honest parties $\mathcal{H}$. A simulator $\mathcal{S}$ interacts with $\mathcal{F}$ in the ideal world. We say that a protocol Π securely computes $\mathcal{F}$ against malicious adversaries if for every PPT adversary $\mathcal{A}$ there exists a PPT simulator $\mathcal{S}$ such that*

$$Real_{\Pi,\mathcal{A}(z)}(1^\ell, 1^\kappa, \{x_i\}_{i\notin\mathcal{C}}) \stackrel{c}{\equiv} Ideal^{\mathcal{L}}_{\mathcal{F},\mathcal{S}(z)}(1^\ell, 1^\kappa, \{x_i\}_{i\notin\mathcal{C}}),$$

where the left-hand side denotes the joint output of honest parties and $\mathcal{A}$, z is the auxiliary input of $\mathcal{A}$, and the right-hand side denotes the outputs of honest parties and $\mathcal{S}$.

The definition requires the joint view of the corrupted coalition to be simulatable from its effective inputs, the prescribed outputs, and the explicitly allowed leakage.

B. UM-PSO Ideal Functionality

We define the ideal functionality of UM-PSO in Fig. 2, which encompasses multiple protocols as follows.

As is standard in the private set operations literature, we assume that each party's input $\mathbf{x}_i$ is a set of m distinct items. Specifically, the reusable function-independent preprocessing phase reveals information about duplicate elimination independently of the set operations selected in the subsequent

Functionality $\mathcal{F}_{\mathsf{UM-PSO}}$

Parameters: the number of parties n, the size of each party's set m, and the quorum threshold $d \in [n]$.

The UM-PSO functionality contains the following commands. On receiving $\mathbf{x}_i = \{x_{i,1}, \cdots, x_{i,m}\}$ from each party P_i,

- **Prepare:** Upon receiving $(\mathsf{Prepare}, \mathbf{x}_i)$ from each party P_i, store $(\mathbf{x}_1, \cdots, \mathbf{x}_n)$ and for every $x \in \bigcup_{i=1}^n \mathbf{x}_i$, compute $ct_x = |\{i \in [n] : x \in \mathbf{x}_i\}|$ and release the leakage $\mathcal{L}_D$.
- **Operate:** Upon receiving $(\mathsf{Operate}, \mathsf{op}, d, f)$ from all parties, evaluate the requested set operation over the stored state:
  1. mPSU: output $\bigcup_{i=1}^n \mathbf{x}_i$ to all parties.
  2. mPSI: output $\bigcap_{i=1}^n \mathbf{x}_i$ and $|\bigcup_{i=1}^n \mathbf{x}_i|$ to all parties.
  3. circuit-mPSI: given an arbitrary function f, output $f(\bigcap_{i=1}^n \mathbf{x}_i)$ and $|\bigcup_{i=1}^n \mathbf{x}_i|$ to all parties.
  4. circuit-mPSU: given an arbitrary function f, output $f(\bigcup_{i=1}^n \mathbf{x}_i)$ and $|\bigcup_{i=1}^n \mathbf{x}_i|$ to all parties.
  5. quorum PSI: output $Qr = \{x : ct_x \geq d, x \in \bigcup_{i=1}^n \mathbf{x}_i\}$ and $|\bigcup_{i=1}^n \mathbf{x}_i|$ to all parties.

The same stored state may be used for multiple queries. If any party changes its input set, the functionality discards the state and requires a new Prepare invocation.

Fig. 2. The multi-party private set operation functionality. For simplicity, we integrate the cardinality computation (mPSU-CA/mPSI-CA) into the circuit-mPSO, as mPSO-CA can be viewed as a function f on set results, with the goal of computing the cardinality $|\bigcup_{i=1}^n \mathbf{x}_i|$ or $|\bigcap_{i=1}^n \mathbf{x}_i|$.

online phase. Let $\mathcal{U} = \bigcup_{i=1}^n \mathbf{x}_i$. The preprocessing phase begins with nm input items. We then use two subprotocols, DOPRF and tSSS, to privately deduplicate the same items, aggregate their frequency payloads, and replace exactly one of the two identical items with a dummy item. Since every redundant occurrence is eliminated exactly once, if D denotes the total number of successful duplicate eliminations, then $D = nm - |\mathcal{U}|$. Thus, the preprocessing transcript reveals the cardinality of the union. For mPSU, $|\mathcal{U}|$ is already implied by the prescribed union output and therefore does not constitute additional output leakage. For mPSI, circuit-mPSI, circuit-mPSU, and quorum PSI, however, the prescribed result does not in general determine $|\mathcal{U}|$. Accordingly, the functionality in Fig. 2 explicitly includes $|\mathcal{U}|$ in the outputs of these operations.

More precisely, the leakage originates from the equality relations exposed by the DOPRF comparisons during randomized deduplication. At this point, we describe these comparisons abstractly as a sequence of randomized deduplication, and the concrete formalization of this leakage $\mathcal{L}_D$ is defined in Section IV-D.

C. Authenticated Secret Sharing Functionality

Following the arithmetic-black-box formalization [29], we model the authenticated secret-sharing basic operations, secret-shared shuffle, and secure comparison through a single stateful ideal functionality $\mathcal{F}_{\mathsf{AuSS}}$ as shown in Fig. 3. The functionality maintains an internal typed value store indexed by fresh public identifiers. Each item value is referenced by an identifier id,

Functionality $\mathcal{F}_{\mathsf{AuSS}}$

Parameters: the number of parties n; a prime p; a cyclic group $\mathbb{G}$ of order p; a generator $g \in \mathbb{G}$; a tuple $\mathsf{Val}[\mathsf{id}] = (\mathsf{type}, x)$, where x represents a stored value, id is a unique identifier, and $\mathsf{type} \in \{\mathsf{arithmetic}, \mathsf{group}\}$; arithmetic values belong to $\mathbb{F}_p$, while group values belong to $\mathbb{G}$.

The multi-party authenticated secret sharing functionality contains the following operations.

- **Input:** On input $(\mathsf{Input}, P_i, \mathsf{id}_x, x)$ from P_i and $(\mathsf{Input}, P_i, \mathsf{id}_x)$ from all other parties, where id_x is fresh and $x \in \mathbb{F}_p$, store $(\mathsf{arithmetic}, \mathsf{id}_x, x)$.
- **Rand:** On input $(\mathsf{Rand}, \mathsf{id}_r)$ from all parties, sample $r \xleftarrow{\$} \mathbb{F}_p^*$ uniformly at random and store $(\mathsf{arithmetic}, \mathsf{id}_r, r)$.
- **Linear Combine:** On input $(\mathsf{LinComb}, \mathsf{id}, (\mathsf{id}_j)_{j=1}^m, c, (c_j)_{j=1}^m)$ from all parties, retrieve $(\mathsf{id}_1, x_1), \ldots, (\mathsf{id}_m, x_m)$, compute $y = c + \sum_{j=1}^m c_j x_j \pmod p$ and store $(\mathsf{arithmetic}, \mathsf{id}, y)$.
- **Multiply:** On input $(\mathsf{Mult}, \mathsf{id}_z, \mathsf{id}_x, \mathsf{id}_y)$ from all parties, retrieve arithmetic values (id_x, x) and (id_y, y), compute $z = xy \bmod p$, and store $(\mathsf{arithmetic}, \mathsf{id}_z, z)$.
- **Convert:** On input $(\mathsf{Convert}, \mathsf{id}_{\tilde{x}}, \mathsf{id}_x)$ from all parties, retrieve the arithmetic value x, compute $\tilde{x} = g^x$, and store $(\mathsf{group}, \mathsf{id}_{\tilde{x}}, \tilde{x})$.
- **Shuffle:** On input $(\mathsf{Shuffle}, (\mathbf{id}_j)_{j=1}^m, (\mathbf{id}'_j)_{j=1}^m)$ from all parties, where each $\mathbf{id}_j = (\mathsf{id}_{j,1}, \cdots, \mathsf{id}_{j,w})$ is a record containing the same public number of typed identifiers and all output identifiers are fresh, sample $\pi \leftarrow S_m$ uniformly at random. For every $j \in [m]$ and $q \in [w]$, store $\mathsf{Val}[\mathsf{id}'_{j,q}] \leftarrow \mathsf{Val}[\mathsf{id}_{\pi(j),q}]$.
- **Compare:** On input $(\mathsf{Compare}, (\mathsf{id}_j^{\mathsf{cnt}})_{j=1}^m, (\mathsf{id}_{b,j}^{\mathsf{bit}})_{j=1}^m, d)$ from all parties, where $d \in [n]$ is a public threshold, retrieve $x_1, \ldots, x_m$. For every $j \in [m]$, compute $b_j = \begin{cases} 1, & \mathsf{Val}[\mathsf{id}_j^{\mathsf{cnt}}] \geq d, \\ 0, & \mathsf{Val}[\mathsf{id}_j^{\mathsf{cnt}}] < d, \end{cases}$ and store $(\mathsf{arithmetic}, \mathsf{id}_{b,j}^{\mathsf{bit}}, b_j)$.
- **Evaluate:** On input $(\mathsf{Evaluate}, (\mathsf{id}_{x,j})_{j=1}^L, (\mathsf{id}_{y,k})_{k=1}^s, C)$ from all parties, retrieve the referenced values, compute $(y_1, \ldots, y_s) = C(x_1, \ldots, x_L)$, and store the outputs under the fresh identifiers $\mathsf{id}_{y,1}, \ldots, \mathsf{id}_{y,s}$.
- **OpenCheck:** On input $(\mathsf{OpenCheck}, \mathsf{type}, \mathsf{id})$ from all honest parties, retrieve $Val[\mathsf{id}] = (\mathsf{type}, y)$ and send y to the adversary. If the adversary responds with Deliver, output y to all parties; otherwise output $\perp$ to all parties and terminate.

Fig. 3. The ideal functionality for the authenticated secret shared operations.

while its associated frequency counter is stored as a separate arithmetic value under an identifier $\mathsf{id}^{\mathsf{cnt}}$. In the ideal world, a stored identifier refers directly to an underlying value. In the real protocol, an identifier is represented by an SPDZ authenticated sharing, while a group identifier is represented by the corresponding secret-shared group element.

Instantiations. In line with the notation used in the SPDZ secret-sharing schemes [24, 30], we use $[\![x]\!]$ to denote an arithmetic Linear Secret Sharing (LSS) of $x \in \mathbb{F}$ shared among n parties. When using the superscript notation, $[\![x]\!] \in \mathbb{F}_p$ represents the secret share of x mod p. A party P_i holds a random share $[\![x]\!]_i$ $(1 \leq i \leq n)$. For simplicity, when the index is omitted, $[\![x]\!]$ refers to the shares of the same secret

value held by all parties.

LSS provides perfect security against up to $n-1$ corrupted parties [31]. It employs Information-Theoretic Message Authentication Codes (IT-MACs) to achieve malicious security. Specifically, each party holds a uniformly random additive share $[\![\xi]\!]$ for a secret global MAC key $\xi \xleftarrow{\$} \mathbb{F}$. For each element share $[\![x]\!]$, the corresponding MAC share is denoted by $[\![\gamma(x)]\!]$, where $\gamma(x) = \xi \cdot x$. Accordingly, we write $\langle x \rangle = ([\![x]\!], [\![\gamma(x)]\!]) \xleftarrow{\$} \mathbb{F}^2$ to denote the Authenticated Secret Share (AuSS) of x. Every operation on a value share $[\![x]\!]$ will also be performed on its MAC share $[\![\gamma(x)]\!]$.

Linear operations of $\langle\cdot\rangle$-sharing. We formalize the ideal functionality of AuSS $\mathcal{F}_{\mathsf{AuSS}}$ in Fig. 3. Technically, the functionality $\mathcal{F}_{\mathsf{AuSS}}$ can be achieved by the following procedures:

- LinComb: $\langle a \cdot x \rangle \leftarrow a \cdot \langle x \rangle$. P_i is given a shared value $\langle x \rangle_i$ and a public value a, and locally computes $a \cdot \langle x \rangle$ and gets $\langle a \cdot x \rangle$;
- Convert: $\langle\!|x|\!\rangle \leftarrow \mathrm{Convert}(\langle x \rangle)$. P_i locally computes $g^{\langle x \rangle_i}$ over $\mathbb{G}$. So parties can convert $\langle x \rangle$ to $\langle\!|x|\!\rangle = \langle g^x \rangle$ [17].
- Rand: $\langle r \rangle \leftarrow \mathrm{Rand}()$. P_i locally chooses a random value as $r_i \xleftarrow{\$} \mathbb{F}_p$ and shares it with the other parties, so parties can get $\langle r \rangle = \sum_{i=1}^{n} \langle r_i \rangle$;
- Multiply: $\langle x \cdot y \rangle \leftarrow \langle x \rangle \cdot \langle y \rangle$. Parties first pre-shares *a Beaver Triple* $(\langle a \rangle, \langle b \rangle, \langle c \rangle)$ with $a \cdot b = c$. Then they compute $\langle e \rangle \leftarrow \langle x \rangle - \langle a \rangle$ and $\langle f \rangle \leftarrow \langle y \rangle - \langle b \rangle$, and open e and f. So they can compute $\langle x \cdot y \rangle = \langle c \rangle + f \cdot \langle a \rangle + e \cdot \langle b \rangle + e \cdot f$;

Following [30], to obtain an authenticated sharing of an input x held by party P_i, i.e., $\langle x \rangle \leftarrow \mathsf{Input}(x)$, the parties first invoke $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Rand}()$ to generate a random authenticated sharing $\langle r \rangle$, and open r only to P_i. P_i computes and broadcasts $\epsilon = x - r$, after which all parties locally compute $\langle x \rangle \leftarrow \langle r \rangle + \epsilon$.

Opening $\langle\cdot\rangle$ & $\langle\!|\cdot|\!\rangle$-sharings and checking the MAC. P_i publishes its share $\langle x \rangle_i$, so that each party can recover a value by computing $x' \leftarrow \sum_{i=1}^{n} \langle x \rangle_i$. Then, they can leverage the embedded MAC to detect any errors or malicious behavior by computing $[\![d]\!] \leftarrow [\![\gamma(x)]\!] - [\![\xi]\!] \cdot x'$. The parties open the share of d and check whether $d = 0$. If not, they abort. Similarly, to open $\langle\!|\cdot|\!\rangle$-sharing, they first recover $g^{x'} = \prod_{i=1}^{n} ([\![g^x]\!]_i)$ and then P_i computes $[\![g^d]\!]_i \leftarrow (g^{x'})^{[\![\xi]\!]_i} / \langle\!|\gamma(x)|\!\rangle$. The parties open $[\![g^d]\!]_i$ and check whether $\prod_{i=1}^{t} ([\![g^d]\!]_i) = 1$.

Among the basic arithmetic operations, linear combinations are local, whereas authenticated inputs, multiplications, and openings require interaction.

Multi-party Secret Shared Shuffle. The goal of the Shuffle command is to allow n parties $P_1, \cdots, P_n$ to jointly permute shares of a set $\mathbf{x} = \{x_1, \cdots, x_m\}$ using a permutation π known by none of parties [32, 33]. The permutation function π is a bijective function: $[m] \mapsto [m]$, denoted as $\mathbf{S}_m$. Then, for the set $\mathbf{x}$, we can obtain the following.

$$\mathbf{x}' = \pi(\{x_1, \cdots, x_m\}) = (x_{\pi(1)}, \cdots, x_{\pi(m)}). \tag{1}$$

In addition, the functionality defines an optional opening procedure, which can be invoked to reconstruct the shuffled values when required by the specific protocol. The construction of an mSSS protocol is proposed in [34], with details shown in Appendix B.

Secure Comparison. The Compare command enables comparison operations over secret shared inputs while ensuring outputs remain secret shared. We consider a secure comparison scenario, where a set of secret shared values is compared against a threshold. The outputs are secret sharings of 0 or 1, representing whether each item is not less than the threshold. We adopt the secure comparison protocol from [23, 35]. Its detailed construction, as well as the corresponding computation and communication costs, is provided in Appendix D.

D. Updatable Pseudo-random Function

Dodis-Yampolskiy PRF [36]. DY-PRF is defined as

$$\mathcal{F}_{\mathsf{DY}}.(mk, x) = g^{\frac{1}{mk+x}},$$

where g denotes a generator of a cyclic group $\mathbb{G}$ of order p, and $mk \xleftarrow{\$} \mathbb{F}_p^*$. A key feature of DY-PRF is its support for updates via key rotation, which enables previously computed PRF values to be re-randomized under a new key [17, 37]. The formal definition of an updatable DY-PRF is given below.

Definition 2 (Updatable DY-PRF)**.** *An updatable DY-PRF with a master key space $\mathcal{K}$, a domain $\mathcal{X}$, and a range $\mathcal{Y}$, is a pseudorandom function F_{DY} with an additional update key space $\mathcal{K}_u$. Three probabilistic polynomial-time (PPT) algorithms* $(\mathrm{KeyGen}, \mathrm{Eval}, \mathrm{Update})$ *are included in F_{DY}:*

- $\mathcal{F}_{\mathsf{DY}}.\mathrm{KeyGen}(1^\kappa, n)$. *This is a key generation algorithm. It takes the security parameter κ as input and outputs a master key $mk \xleftarrow{\$} \mathcal{K}$, an update key $uk \xleftarrow{\$} \mathcal{K}_u$.*
- $\mathcal{F}_{\mathsf{DY}}.\mathrm{Eval}(mk, x)$. *Given a master key mk and a message x from the input message space $\mathcal{M}$, this algorithm outputs a PRF value $\tilde{x} = g^{\frac{1}{mk+x}}$.*
- $\mathcal{F}_{\mathsf{DY}}.\mathrm{Update}(\tilde{x}, uk)$. *An update algorithm takes a PRF value $\tilde{x}$ and an update key uk as inputs. It outputs a randomized PRF value $\tilde{x}_{uk}$ updated using uk. The updated PRF value is denoted as $\tilde{x}_{uk} = g^{\frac{1}{mk+x} \cdot uk}$.*

The function $\mathcal{F}_{\mathsf{DY}}.\mathrm{Update}$ can be performed multiple times with different update keys [37]. Pseudorandomness is guaranteed by the Decisional q-Diffie-Hellman Inversion Assumption (q-DDHI) [17, 36].

IV. Unified Private Set Operation Framework

We first give an overview of UM-PSO, then introduce the essential sub-protocols used in $\mathcal{F}_{\mathsf{UM-PSO}}$, including the tree-structured shuffle procedure (tSSS) and a Distributed Oblivious Pseudorandom Function, $\mathcal{F}_{\mathsf{DOPRF}}$. We also illustrate how these sub-protocols can be seamlessly integrated with SPDZ to form UM-PSO suitable for any set-related computations.

A. Technical Overview

We provide a technical overview of $\mathcal{F}_{\mathsf{UM-PSO}}$. Our protocol is divided into three phases. First, an input-independent setup phase generates correlated randomness (CR), including authenticated multiplication triples, edaBits, and shuffle materials. Second, a function-independent preprocessing phase

authenticates the parties' fixed input sets and transforms them into a reusable secret shared *unified pool*, in which the occurrences of each item across all input sets are counted in a privacy-preserving manner and stored as a payload associated with that item. These payload values are maintained in the secret-shared form among the parties, while all duplicated items are removed in parallel. This phase is input-dependent but independent of the subsequently selected set operation. Third, an online set operations phase evaluates a selected set functionality over the prepared pool. The parties then securely compare each item's payload against a threshold (e.g., 1 for mPSU, n for mPSI, or thr for quorum PSI).

Revisiting Malicious Two-party circuit-PSI [17]. To motivate our design, we begin by revisiting a malicious two-party circuit-PSI protocol [17], and then outline the technical challenges and our overall roadmap for constructing a function-independent mPSO protocol. Consider a scenario with two input sets from two parties, namely $\mathbf{x}_1 = \{x_{1,1}, \cdots, x_{1,m}\}$ and $\mathbf{x}_2 = \{x_{2,1}, \cdots, x_{2,m}\}$, held by P_1 and P_2, respectively. Two parties follow the *shuffle-randomize-compare* paradigm in three steps:

(1) Both parties invoke a two-party secret-shared shuffle protocol to *shuffle* their input items, obtaining shuffled shares $\langle \pi(\mathbf{x}_1) \rangle$ and $\langle \pi(\mathbf{x}_2) \rangle$.
(2) Both parties run an SPDZ-compatible two-party OPRF protocol on the secret-shared inputs $\langle \pi(\mathbf{x}_1) \rangle$ and $\langle \pi(\mathbf{x}_2) \rangle$ to compute the corresponding *randomized* values, obtaining PRF outputs $\mathcal{F}_{\mathsf{DY}}.\mathsf{Eval}(mk, \mathbf{x}_1)$ and $\mathcal{F}_{\mathsf{DY}}.\mathsf{Eval}(mk, \mathbf{x}_2)$.
(3) The PRF outputs are then opened to both parties, who *compare* them to derive $f(\mathbf{x}_1 \cap \mathbf{x}_2)$.

Challenges of extending to multi-party setting. Following the *shuffle–randomize–compare* paradigm in [17], a straightforward extension to the multi-party setting is to compute PRF values for all input items and then compare the results. In detail, we consider parties P_i for $i \in [n]$, where each party holds an input set $\mathbf{x}_i = x_{i,1}, \cdots, x_{i,m}$. Parties first call $\mathcal{F}_{\mathsf{mSSS}}$ to shuffle all items to obtain secret shared shuffled inputs $\langle \pi(\mathbf{x}[1:n]) \rangle = \langle x_{\pi(1,1)} \rangle, \cdots, \langle x_{\pi(n,m)} \rangle$. Next, parties take shared items $\langle \pi(\mathbf{x}[1:n]) \rangle$ together with a shared key $\langle mk \rangle$ as inputs to a multiparty distributed OPRF protocol $\mathcal{F}_{\mathsf{DOPRF}}$. At the end of this protocol, all parties obtain PRF outputs for each item, namely $\mathcal{F}_{\mathsf{DY}}.\mathsf{Eval}(mk, \pi(\mathbf{x}[1:n]))$. Parties then compare these PRF values and count their multiplicities in order to determine which items in $\langle \pi(\mathbf{x}[1:n]) \rangle$ belong to the intersection or union, and finally recover the results. This approach immediately raises a major concern: all PRF values are computed under the same key. As a result, identical input items are mapped to identical PRF outputs for comparison, which inevitably reveals their frequencies.

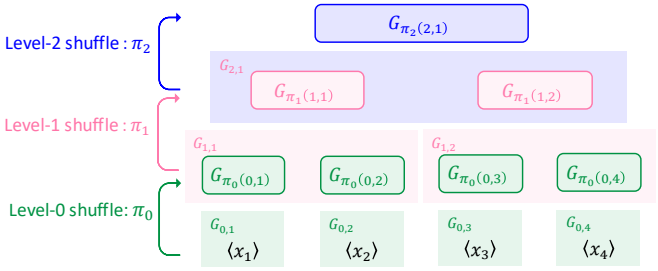

Fig. 4. A toy example of Π_{tSSS} for 4 items. All input items are secret shared between parties.

Limitation: Frequency leakage. Parties can obtain PRF values of all input items under the same key, so the frequency distribution of inputs will be revealed, although they are unable to determine which exact item corresponds to each frequency.

Therefore, we need to break linkability so that parties cannot associate PRF values derived from the same input item across different comparisons.

Our Construction. To solve the aforementioned issues, we propose a tree-structured secret-shared shuffle (tSSS) protocol that organizes item merging in a hierarchical manner. This structure enables us to extend two-party PSO protocols to the multi-party setting with acceptable efficiency, while preventing frequency leakage.

A toy example of tSSS is illustrated in Fig. 4. The parties secret-share four input items in total as $\langle \mathbf{x} \rangle = \{\langle x_1 \rangle, \cdots, \langle x_4 \rangle\}$. In Level-0, treat each shared item as a singleton group, i.e., $G_{0,1}, G_{0,2}, G_{0,3}, G_{0,4}$. Parties will invoke $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Shuffle}()$ to shuffle all input items in groups using a permutation π_0. Next, in Level-1, the shuffled items are paired: every two neighboring items are grouped, forming two groups in total, $G_{1,1}, G_{1,2}$. These groups are then shuffled using a fresh permutation π_1, yielding $G_{\pi_1(1,1)}, G_{\pi_1(1,2)}$. $G_{\pi_1(1,1)}$ consists of two secret-shared items concatenated as $\{\langle x_{\pi_1\pi_0(1)} \rangle || \langle x_{\pi_1\pi_0(2)} \rangle\}$, while $G_{\pi_1(1,2)}$ consists of $\{\langle x_{\pi_1\pi_0(3)} \rangle || \langle x_{\pi_1\pi_0(4)} \rangle\}$. In Level 2, the two groups $G_{1,1}$ and $G_{1,2}$ are merged into a single group $G_{2,1}$. Since only one group exists, the permutation π_2 reduces to the identity permutation, and the positions of elements remain unchanged. In summary, in each level, every two neighboring groups from the previous layer are merged into one group, and the resulting groups are shuffled using a fresh permutation. Throughout the entire procedure, none of the parties learns any of the applied permutations.

The next step is to identify and deduplicate identical items within each pair of neighboring groups. To this end, we propose a distributed oblivious PRF (DOPRF) protocol, which evaluates a pseudorandom function on secret shared items using a key jointly held by all parties. Specifically, within each pair of neighboring groups, all items are evaluated under the same DOPRF key. During this process, each item is associated with a payload value, $\langle ct_{x_1} \rangle \leftarrow \mathsf{Count}(\langle x_1 \rangle)$, which

records the multiplicity of item x_1 across all input sets and is initialized to 1. In different levels of tSSS, parties will do the following:

- Level-0 shuffle: Parties evaluate DOPRF under key $mk_{1,1}$ for items in $G_{\pi_0(0,1)}, G_{\pi_0(0,2)}$, and under key $mk_{1,2}$ for items in $G_{\pi_0(0,3)}, G_{\pi_0(0,4)}$. They then compare PRFs across neighboring groups. If $x_1 = x_2$, the payload shares of x_1 and x_2 are added, and $\langle x_2 \rangle$ is replaced by a random sharing $\langle r \rangle$ with $r \xleftarrow{\$} \mathbb{F}_q$ and $\mathsf{Count}(r) = 0$; otherwise, both items are kept.
- Level-1 shuffle: Parties evaluate DOPRF using the key $mk_{2,1}$ for the items in $G_{1,1}, G_{1,2}$. They then compare the PRF values between $G_{\pi_1(1,1)}$ and $G_{\pi_1(1,2)}$. As at Level-0, any identical items are replaced with a fresh random sharing.
- Level-2 shuffle: After deduplication at the previous levels, the shared items in $G_{2,1}$ constitute the resulting *unified pool*.

Once the unified pool has been produced in the preprocessing phase, parties proceed to the online phase, where they agree on a threshold thr ($1 \leq thr \leq n$) determined by the specific set operation, and subsequently compare the shared payload value of each item against thr.

Procedure tSSS

Input: Let $N = mn$. P_i $(1 \leq i \leq n)$ holds shared input sets $\mathbf{x}_i = \{x_{i,1}, ..., x_{i,m}\}$.
Output: P_i learns the shares in $\tilde{G}_{\log_2(N),1}$.

- **for** $h = 0, 1, \ldots, \log_2(N)$ **do**
  - **if** $h = 0$ **then**
    - parties call $\mathcal{F}_{\mathsf{AuSS}}$.Shuffle with $\langle x_{1,1} \rangle, \cdots, \langle x_{n,m} \rangle$ and π_0 to obtain $\langle x_{\pi_0(1,1)} \rangle, \cdots, \langle x_{\pi_0(n,m)} \rangle$;
    - $\langle x_{\pi_0(1,1)} \rangle, \cdots, \langle x_{\pi_0(n,m)} \rangle$ are assigned to groups $G_{0,1}, \cdots, G_{0,N}$, respectively; ▷ Level-0 shuffle
  - **else for** $j = 1, 3, \cdots, (\frac{N}{2^{h-1}} - 1)$ **do**:
    - merge items in $G_{h-1,j}$ and $G_{h-1,j+1}$ to $G_{h,\frac{j+1}{2}}$;
    - parties call $\mathcal{F}_{\mathsf{AuSS}}$.Shuffle with $G_{h,1}, \cdots, G_{h,\frac{N}{2^h}}$ by π_h to get $\tilde{G}_{h,1}, \cdots, \tilde{G}_{h,\frac{N}{2^h}}$; ▷ Level-$h$ shuffle

Fig. 5. The tree-structured secret shared shuffle procedure.

B. Tree-structured Secret Shared Shuffle

This section formally introduces the tree-structured multi-party secret-shared shuffle procedure, denoted as tSSS, which serves as the main building block of our UM-PSO. The core idea of tSSS is to secret-share, shuffle, and recursively group input items across multiple layers of a tree structure, thereby breaking the *linkability* of items when counting their frequencies.

In general, we consider n input sets $\mathbf{X} = \{\mathbf{x}_1, \cdots, \mathbf{x}_n\}$, and each set has m items, i.e., $\mathbf{x}_1 = \{x_{1,1}, \cdots x_{1,m}\}$. We assume mn is a power of 2. The tree has depth $\log_2(mn)$, and at each level, parties merge every pair of neighboring groups into a single group and secret-share the resulting merged groups. The detailed procedure is provided in Fig. 5. The complexity of mSSS is determined by the total number of items that need to be shuffled, rather than the size of each item. Therefore, in level-h, the communication cost is $O(\frac{n^2 m}{2^h} \log m)$.

Functionality $\mathcal{F}_{\mathsf{DOPRF}}$

Public parameters: the number of parties n, a field $\mathbb{F}_p$, a cyclic group $\mathbb{G}$ of order p, and a generator g of $\mathbb{G}$.
Private parameter: a master key mk, an update key uk.

- **Eval**: On input $(\mathsf{Eval}, \mathsf{id}_{mk}, \mathsf{id}_e, \mathsf{id}_x, \mathsf{id}_{\tilde{x}})$ from parties, retrieve $mk, x \in \mathbb{F}_p$. If $mk + x = 0 \bmod p$, abort; otherwise compute $e = (mk+x)^{-1}$, $\tilde{x} = g^e$, and store $\mathsf{Val}[\mathsf{id}_e] = (\mathsf{arithmetic}, e)$, $\mathsf{Val}[\mathsf{id}_{\tilde{x}}] = (\mathsf{group}, \tilde{x})$.
- **Update**: On input $(\mathsf{Update}, \mathsf{id}_{uk}, \mathsf{id}_e, \mathsf{id}_{e'}, \mathsf{id}_{\tilde{x_u}})$ from all n parties, retrieve uk, e, compute $e' = e \cdot uk$ and $\tilde{x_u} = g^{e'}$, and store them under $\mathsf{id}_{e'}$ and $\mathsf{id}_{\tilde{x_u}}$, respectively.

Fig. 6. Ideal functionality of DOPRF. The functionality extends $\mathcal{F}_{\mathsf{AuSS}}$ and inherits its typed value store, identifier namespace, output interface, validation rules, and abort semantics.

C. Distributed Oblivious Pseudorandom Function

An oblivious pseudorandom function (OPRF) allows a party holding a private input x and another party holding a key k for a PRF F to evaluate F_{mk} on x obliviously [38, 39]. Building on the PRF foundation mentioned in the last section, we construct a Distributed Oblivious Pseudorandom Function (DOPRF). For n parties, $P_1, \cdots, P_n$ will share a master PRF key mk and an input value x. When jointly evaluating the PRF on x, no party can learn mk. The ideal functionality for DOPRF is shown in Fig. 6. Update operates on valid internal state $(\mathsf{id}_e, \mathsf{id}_{\tilde{x}})$ produced by a previous Eval/Update. Depending on the calling protocol, the functionality may invoke Eval alone, invoke Update on an existing valid PRF value, or Eval followed by one or more invocations of Update.

Below, we provide a detailed explanation of how to implement the DOPRF protocol based on the DY-PRF. The DY-based DOPRF is extended from a 2PC-based DOPRF protocol [17]. We consider the setting in which n parties jointly compute a PRF value on input x. Specifically, a key feature of the DY-PRF-based DOPRF is the update of the PRF, as shown in Fig. 7. After this process, this protocol will reveal the PRF value of the input x to the specified parties. Depending on the calling protocol, parties may open the value after Eval, or retain the internal exponent state and immediately execute one or more Update operations before opening only the final group value. Then, if parties intend to update the value g^e, they can choose an updating key uk for this operation as in steps 6 and 7. After step 7, parties can obtain $g^e = g^{\frac{uk}{mk+x}}$.

1) Efficiency Analysis: The aforementioned Π_{DOPRF} has low round complexity. In particular, parties only need to perform one secret-shared multiplication followed by two opening operations to compute a PRF value. In step 2, parties will compute $\langle d \rangle$ with a Beaver triple. The two opening operations are to open d and g^e. Excluding the final reconstruction of the prescribed PRF output, Eval requires two communication

rounds: one for the authenticated multiplication and one for opening d.

Protocol Π_{DOPRF}

Parameters: AuSS sharing $\langle mk \rangle$ for the master key mk, AuSS sharing of the update key $\langle uk \rangle$.

Input: Given $\langle x \rangle$, $\langle mk \rangle$, and $\langle uk \rangle$, each party proceeds as follows:

1. Call $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Rand}()$ to generate $\langle r \rangle$, where $r \xleftarrow{\$} \mathbb{F}_p^*$;
2. Compute $\langle d \rangle \leftarrow \langle r \rangle \cdot (\langle mk \rangle + \langle x \rangle)$;
3. Open $d \leftarrow \mathcal{F}_{\mathsf{AuSS}}.\mathsf{OpenCheck}(\langle d \rangle)$;
4. Compute $\langle e \rangle \leftarrow d^{-1} \cdot \langle r \rangle$;

Eval:

5. Call $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Convert}(\langle e \rangle)$ to obtain $\langle\!\langle e \rangle\!\rangle$;

Update:

6. Compute $\langle e \rangle \leftarrow \langle e \rangle \cdot \langle uk \rangle$;
7. Call $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Convert}(\langle e \rangle)$ to obtain $\langle\!\langle e \rangle\!\rangle$;

Output: Each party obtains $g^e \leftarrow \mathcal{F}_{\mathsf{AuSS}}.\mathsf{OpenCheck}(\langle\!\langle e \rangle\!\rangle)$.

Fig. 7. The DY-based DOPRF protocol.

2) Security Analysis: For the security of Π_{DOPRF}, we require that the PRF results look random whenever the input item x is chosen arbitrarily. We claim that our DOPRF protocol Π_{DOPRF} is secure against a malicious adversary in the n-party setting, as formulated by the following theorem.

Theorem 1. *In the $\mathcal{F}_{\mathsf{AuSS}}$-hybrid model, the protocol Π_{DOPRF} implements $\mathcal{F}_{\mathsf{DOPRF}}$ correctly and securely against any malicious adversary that statically corrupts up to $n-1$ parties.*

Proof. We construct an ideal world simulator $\mathcal{S}_{\mathrm{DOPRF}}$ that can simulate the view of the adversary. The details of the proof are given in Appendix G. □

D. Overall Workflow of UM-PSO

We present an overview of how our multi-party private set operation (mPSO) framework UM-PSO is constructed using the aforementioned subprotocols. UM-PSO separates the computation into three phases: input-independent setup, function-independent preprocessing, and online set operations (Fig. 8). The first phase generates authenticated correlated randomness, whereas the second phase processes the parties' fixed inputs into a reusable pool independently of the subsequently selected set operation.

1) Function-independent Preprocessing: We assume that n parties input n sets $\mathbf{x}_1, \cdots, \mathbf{x}_n$, and each set has m items, i.e., $\mathbf{x}_i = \{x_{i,1}, \cdots, x_{i,m}\}$. For the case where $N = mn$ is not a power of two, we pad the input with random items whose frequency tags are set to 0 until the input size reaches the next power of two. In the preprocessing phase, parties privately count the frequency of each item. Then all parties can get shares of the frequency. They can use the shared frequency to compare with a threshold in the online phase. In Step 1 of Fig. 8, parties will first treat each item as a singleton group, and call $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Shuffle}$ with π_0 to shuffle all nm groups. Then, in Step 2, when $h = 1$, parties will first compute PRF values for items in all groups, and items in neighboring groups use the same PRF key, i.e.,

$$\underbrace{\tilde{G}_{0,1}, \tilde{G}_{0,2}}_{mk_{1,1}}, \underbrace{\tilde{G}_{0,3}, \tilde{G}_{0,4}}_{mk_{1,2}}, \ldots, \underbrace{\tilde{G}_{0,nm-1}, \tilde{G}_{0,nm}}_{mk_{1,\frac{nm}{2}}}.$$

During Step $2(b)$, parties locally match the opened DOPRF values between the two child groups. Since each child group contains no duplicate valid items, each PRF value has at most one match in the opposite group.

Consequently, each neighboring pair of groups is associated with a common PRF key, whereas distinct pairs use independent keys. This separation guarantees that PRF comparisons are restricted to items within the same pair, eliminating unintended linkability across different pairs. So parties can directly compare the PRF values in the neighboring groups; if the two groups contain equal PRF values, then parties find the corresponding sharings, add the corresponding payload sharings, and replace one of them with a sharing of a random dummy value $r_\Delta \in \mathbb{R}$. Let $\mathbb{R} \subseteq \mathbb{Z}_p$ be a random domain such that $\mathbb{R} \cap \{x_{1,1}, \ldots, x_{n,m}\} = \emptyset$. Then, parties will combine the neighboring groups into a new group: $G_{1,1} \leftarrow \tilde{G}_{0,1} \| \tilde{G}_{0,2}$, $G_{1,2} \leftarrow \tilde{G}_{0,3} \| \tilde{G}_{0,4}$, $\cdots$, $G_{1,\frac{nm}{2}} \leftarrow \tilde{G}_{0,nm-1} \| \tilde{G}_{0,nm}$. Next, parties shuffle the generated groups using the permutation π_1, which is the Level-1 shuffle described in Section IV-B. For each level $h \in [1, \log_2(nm)]$, parties recursively execute Steps $2(a)$ to (c). In the final level, when $h = \log_2(nm)$, all items are merged into a single final group $\tilde{G}_{\log(nm),1}$.

During this phase, the protocol guarantees that each group contains no duplicate items, and identical items are deduplicated prior to group merging. Each item carries an associated payload value that counts its total number of occurrences in all input sets.

Concrete Optimizations. In Step 2(a), when we choose Π_{DOPRF} to realize $\mathcal{F}_{\mathsf{DOPRF}}$, we can leverage its updatable property to optimize the preprocessing phase. Technically, in the level-h shuffle, parties need to compute the PRF value using the same PRF keys for shares in the neighboring groups for comparison. Then, in the next level-$(h+1)$ shuffle, parties will first shuffle all PRF values and corresponding PRF keys, so that they can invoke $\mathrm{F}_{\mathrm{DY}}.\mathsf{Update}$ to directly update the PRF values in the last level. Therefore, we give a thorough description of the optimized Π_{mPSO} in Appendix J.

2) Online Set Operations: Parties can choose different operation types as shown in Step 4. According to the selected operation, parties compare the payload values of items in the final group $\tilde{G}_{\log(nm),1}$ with a threshold thr. For mPSU, the goal is to select all items that appear at least once across the input sets, i.e., $thr = 1$. For mPSI, parties select the items whose $\mathsf{Count}()$ values are equal to the number of parties, i.e., $thr = n$. If an item x is in the intersection set, then parties invoke $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Compare}$ to compute $n \leq \mathsf{Count}(x)$. While the maximum number of occurrences of x across all input sets is n, $n \leq \mathsf{Count}(x)$ is equivalent to $n = \mathsf{Count}(x)$. For quorum PSI, the threshold is set to a predefined value, meaning that

Protocol $\Pi_{\text{UM-PSO}}$

Parameters: Party P_i ($i \in [1,n]$) has an input set $\mathbf{x}_i = \{x_{i,1}, \cdots, x_{i,m}\}$, where $\mathbf{x}_i \in \mathbb{Z}_p^m$; the length of each item in $\mathbf{x}_i$ is ℓ; sets of PRF key shares $\{\langle mk_{h,1}\rangle, \cdots, \langle mk_{h,\frac{nm}{2^h}}\rangle\}$, where $h \in [1,H]$, $H = \log_2(nm)$;

Input-independent setup: The parties generate sufficient fresh authenticated correlated randomness for the subsequent executions. Every correlation is consumed at most once.

Function-independent preprocessing: Given inputs $\mathbf{x}_1, \cdots, \mathbf{x}_n$, the parties proceed as follows:

1. Level-0 shuffle: For every $i \in [n]$ and $j \in [m]$, P_i invokes $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Input}(P_i, \mathsf{id}_{0,i,j}, x_{i,j})$. Parties initialize the corresponding frequency counter as $\mathsf{Val}[\mathsf{id}^{\mathsf{cnt}}_{0,i,j}] = (\mathsf{arithmetic}, 1)$.
   a) Parties partition these shares into nm groups $\vec{G_0} = \{G_{0,1}, \cdots, G_{0,nm}\}$, with a single share per group; that is, $G_{0,(i-1)m+j}.\mathrm{val} = \{x_{i,j}\}$ for $i \in [1,n]$ and $j \in [1,m]$.
   b) Parties then invoke $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Shuffle}$, which samples a hidden uniformly random permutation π_0. Then parties obtain the shuffled shared set $\tilde{G}_{0,(i-1)m+j}.\mathrm{val} = \{x_{\pi_0(i,j)}\}$.
2. For each level $h = 1, \ldots, H$, the parties do the following:
   a) For each $u \in [1, \frac{nm}{2^h}]$, the parties invoke $\mathcal{F}_{\mathsf{DOPRF}}$ with key $\langle mk_{h,u}\rangle$ to compute PRF values for the shares in groups $\tilde{G}_{h-1,2u-1}$ and $\tilde{G}_{h-1,2u}$, yielding $\tilde{G}_{h-1,2u-1}.\mathrm{PRF}$ and $\tilde{G}_{h-1,2u}.\mathrm{PRF}$.
   b) For each $u \in [1, \frac{nm}{2^h}]$ and $w_0, w_1 \in [1, 2^{h-1}]$, the parties locally compare the opened DOPRF values between two child groups, i.e., $\tilde{G}_{h-1,2u-1}[w_0].\mathrm{PRF}$ and $\tilde{G}_{h-1,2u}[w_1].\mathrm{PRF}$. If the two PRF values are equal, the parties aggregate the corresponding payload shares by setting
   $$\mathsf{Count}(\tilde{G}_{h-1,2u-1}[w_0].\mathrm{val}) \leftarrow \mathcal{F}_{\mathsf{AuSS}}.\mathsf{LinComb}(\mathsf{Count}(\tilde{G}_{h-1,2u-1}[w_0].\mathrm{val}), \mathsf{Count}(\tilde{G}_{h-1,2u}[w_1].\mathrm{val})),$$
   and replace $\tilde{G}_{h-1,2u}[w_1].\mathrm{val}$ with a fresh random share $\langle r_\Delta\rangle \xleftarrow{\$} \mathbb{R}$, where $\mathsf{Count}(r_\Delta) = 0$.
   c) Level-h shuffle: For each $u \in [1, \frac{nm}{2^h}]$, the parties merge groups $\tilde{G}_{h-1,2u-1}$ and $\tilde{G}_{h-1,2u}$ into a new group $G_{h,u}$, and then invoke $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Shuffle}$ on $\vec{G_h} = \{G_{h,1}, \cdots, G_{h,\frac{nm}{2^h}}\}$ with permutation π_h, obtaining the shuffled shared set $\{\tilde{G}_{h,1}, \cdots, \tilde{G}_{h,\frac{nm}{2^h}}\}$.
3. The parties obtain the root group $\tilde{G}_{H,1}$ and initialize an empty set $\vec{Res}$.

Online set operations:

4. The parties choose an operation type:
   $$\mathsf{op} \in \{\text{mPSU, mPSI, quorum PSI, circuit-mPSI, circuit-mPSU}\}.$$
5. The parties set the threshold thr according to op:
   - for mPSU and circuit-mPSU, $thr = 1$;
   - for mPSI and circuit-mPSI, $thr = n$;
   - for quorum PSI, $1 < thr < n$.
6. For each $u \in [1, nm]$, the parties invoke $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Compare}$ to compare $\mathsf{Count}(\tilde{G}_{\log_2(nm),1}[u].\mathrm{val})$ with thr, and store the resulting shares in $\vec{Res}$.
7. If $\mathsf{op} \in \{\text{mPSU, mPSI, quorum PSI}\}$, the parties reconstruct the result by invoking $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{OpenCheck}(\vec{Res})$. For every position j with $b_j = 1$, they open the corresponding item identifier and output the resulting items in their shuffled order.
8. If $\mathsf{op} \in \{\text{circuit-mPSI, circuit-mPSU}\}$, the parties invoke the MPC functionality $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Evaluate}$ to evaluate a function f on the private items in the final unified pool $\tilde{G}_{H,1}[j].val$, for which $b_j = 1$.

Fig. 8. The mPSO protocols in UM-PSO.

items appearing at least thr times in input sets are selected. In contrast, when computing circuit-mPSU or circuit-mPSI, parties do not directly open the shares. Instead, the shares are provided as inputs to a subsequent secure computation that evaluates the target function.

3) Complexity Analysis: Assume that an item x appears $\mathsf{Count}(x) \in [1,n]$ times across the $N = nm$ items. We derive the computation and communication costs of the preprocessing and online phases of UM-PSO as follows. Unless otherwise specified, all computational costs are measured per party, and communication costs are measured between all parties.

- **Function-independent preprocessing.** For a total $N = nm$ items, we assume N is a power of 2, and $H = \log_2 N$. At level h, tSSS shuffle $\frac{N}{2^h}$ groups, and each group contains 2^h records, hence having size $O(2^h\ell)$ bits. According to $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{mSSS}$ construction, the Level-h shuffle incurs $O(\ell n^2 N)$ communication costs. Summing over all H levels gives $O(\ell n^2 N \log N)$ communication for tSSS.
  At every level, DOPRF Eval/Update is performed on N records. Hence, $O(N \log N)$ DOPRF operations are performed in total, incurring $O(\kappa n^2 N \log N)$ aggregate communication. Consequently, the overall communication complexity of the function-independent preprocessing phase is $O((\ell+\kappa)n^2 N \log N)$. Considering ℓ and κ as fixed parameters, this becomes $O(n^2 N \log N)$. For computation, each party performs $O(nN)$ local work for the multiparty shuffle at each level, while DOPRF

Eval/Update and local matching require $O(N)$ work per level. Thus, the overall per-party computational complexity is $O(nN \log N)$.

- **Online Set Operations.** The online phase compares the frequency counter of each of N items with a public threshold. Since $Count(x) \in [0, n]$, each counter can be represented using $b_{cnt} = \lceil \log_2(n+1) \rceil$ bits. Using $\mathcal{F}_{\mathsf{AuSS}}$.Compare described in Appendix D, one comparison require $O(b_{cnt} \log b_{cnt})$ online communication per party and $O(b_{cnt})$ local computation. Therefore, performing N comparisons incurs $O(nNb_{cnt} \log b_{cnt})$ aggregate communication and $O(Nb_{cnt})$ computation per party. These costs exclude the final reconstruction of the prescribed output.

4) Security Analysis: During preprocessing, we can observe that parties need to replace duplicate occurrences with random dummy values, which induces randomized deduplication leakage, defined as $\mathcal{L}_D$. We present the formalization of $\mathcal{L}_D$ as follows.

Definition 3 (Randomized Deduplication Leakage)**.** *Let $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$ denote the parties' input sets and $\rho = (\pi_0, \ldots, \pi_H)$ denote the randomness used by the level-wise $\mathcal{F}_{\mathsf{AuSS}}$.Shuffle. At level h, let $\{(G^L_{h,u}, G^R_{h,u})\}_{u \in [q_h]}$ be the pairs of groups formed after the shuffle.*

For every paired group u, define $E^{\mathbf{X},\rho}_{h,u}$ as the anonymous matching matrix, where $E^{\mathbf{X},\rho}_{h,u}[a, b] = 1$ if and only if the items at positions a and b are selected by the deduplication procedure as an equal pair, causing exactly one of them to be removed. Otherwise, $E^{\mathbf{X},\rho}_{h,u}[a, b] = 0$. Then, the Randomized Deduplication Leakage is defined as

$$\mathcal{L}_D(\mathbf{X}; \rho) := \left(\left(E^{\mathbf{X},\rho}_{h,u} \right)_{u \in [q_h]} \right)_{h=0}^{H}.$$

We write $\mathcal{L}_D(\mathbf{X})$ for the random variable induced by sampling ρ according to the shuffle randomness used in the real protocol.

Here is a toy example to illustrate this leakage. Consider four parties holding a total of eight input items. In the first case, after $\mathcal{F}_{\mathsf{AuSS}}$.Shuffle, the Level-1 items are paired as $(x_1, x_1), (x_1, x_2), (x_3, x_4), (x_5, x_6)$. In this case, one deduplication occurs at Level-1 because of the pair (x_1, x_1). After the neighboring groups are merged, the remaining copy of x_1 may be compared with the remaining x_1 at Level-2, resulting in another deduplication. Hence, one deduplication occurs at Level-1 and another at Level-2. As a second case, suppose that the Level-1 pairs are $(x_1, x_1), (x_2, x_2), (x_3, x_4), (x_5, x_6)$. Here, two deduplications occur directly at Level-1, and no further duplicate remains to be removed at the subsequent levels.

Therefore, both cases contain exactly two successful deduplications and thus imply the same union cardinality, while their level-wise deduplication patterns are different. This motivates $\mathcal{L}_D$, which captures the randomized deduplication pattern revealed throughout tSSS execution rather than merely the total number of removed duplicates. In general, the total number of removed duplicates is a deterministic function of this leakage: $N - |\bigcup_i \mathbf{x}_i| = \sum_{h,u} \|E_{h,u}\|_1$. Since the groups are randomly shuffled and the opened DOPRF values are pseudorandom, the adversary cannot associate an observed match with a particular input item.

Next, we give the security proof of UM-PSO.

Theorem 2. *In the $(\mathcal{F}_{\mathsf{AuSS}}, \mathcal{F}_{\mathsf{DOPRF}})$-hybrid model, the protocol $\Pi_{\mathsf{UM-PSO}}$ implements $\mathcal{F}_{\mathsf{UM-PSO}}$ correctly and securely against a malicious adversary who can control at most $t < n$ parties.*

Proof. We construct a simulator $\mathcal{S}_{\mathsf{UM-PSO}}$ using only the corrupted parties' effective inputs, the prescribed output, and the randomized deduplication leakage $\mathcal{L}_D$. The shuffle process anonymizes honest inputs, while the pseudorandomness of DOPRF allows all opened PRF values to be simulated as random labels, preserving only the equality relations specified by $\mathcal{L}_D$. Hence, the simulated and real views are computationally indistinguishable. The complete proof is given in Appendix G. □

V. Implementation and Evaluation

We implement Π_{mPSO} in C++, using MP-SPDZ[1] and YACL[2] libraries, which provide secret shared operations and cryptographic interfaces. We run most experiments on a desktop PC equipped with a 12th Gen Intel Core i9-12900K and 125 GB of memory, running Ubuntu 20.04 LTS. We emulate different network settings using the Linux `tc` command. The network settings follow previous mPSO works. In the LAN setting, the bandwidth is 1 Gbps with 0.1 ms RTT. In the WAN setting, the bandwidths are 400 Mbps with 20 ms RTT latency, and 100 Mbps with 40 ms RTT latency.

We provide a detailed performance breakdown of our UM-PSO framework, separated as the input-independent setup (CR), the function-independent preprocessing phase (Pre.), and the online set operations (Online.). In the preprocessing phase, we separately evaluate the performance of $\mathcal{F}_{\mathsf{DOPRF}}$ and $\mathcal{F}_{\mathsf{mSSS}}$ under different network settings. Note that all SPDZ-based protocols require the generation of Correlated Randomness (CR) in the offline phase, which is used for Beaver triples, secure comparison, and other secure computations. CR generation is typically more expensive than the main protocol execution. Following prior works [17, 40], we exclude the CR generation time from the evaluation of the main protocols (Sec. V-A and Sec. V-B), while reporting this cost specifically in Appendix I. In the online computation phase, we illustrate the performance of the secure comparison protocol Π_{Compare} (Sec. V-A).

Baselines. To evaluate UM-PSO against existing protocols, we first implement a baseline that extends BA12 [2] by directly applying the SPDZ framework to perform radix sort and secure shuffling, yielding a straightforward maliciously secure mPSO

[1] https://github.com/data61/MP-SPDZ
[2] https://github.com/secretflow/yacl

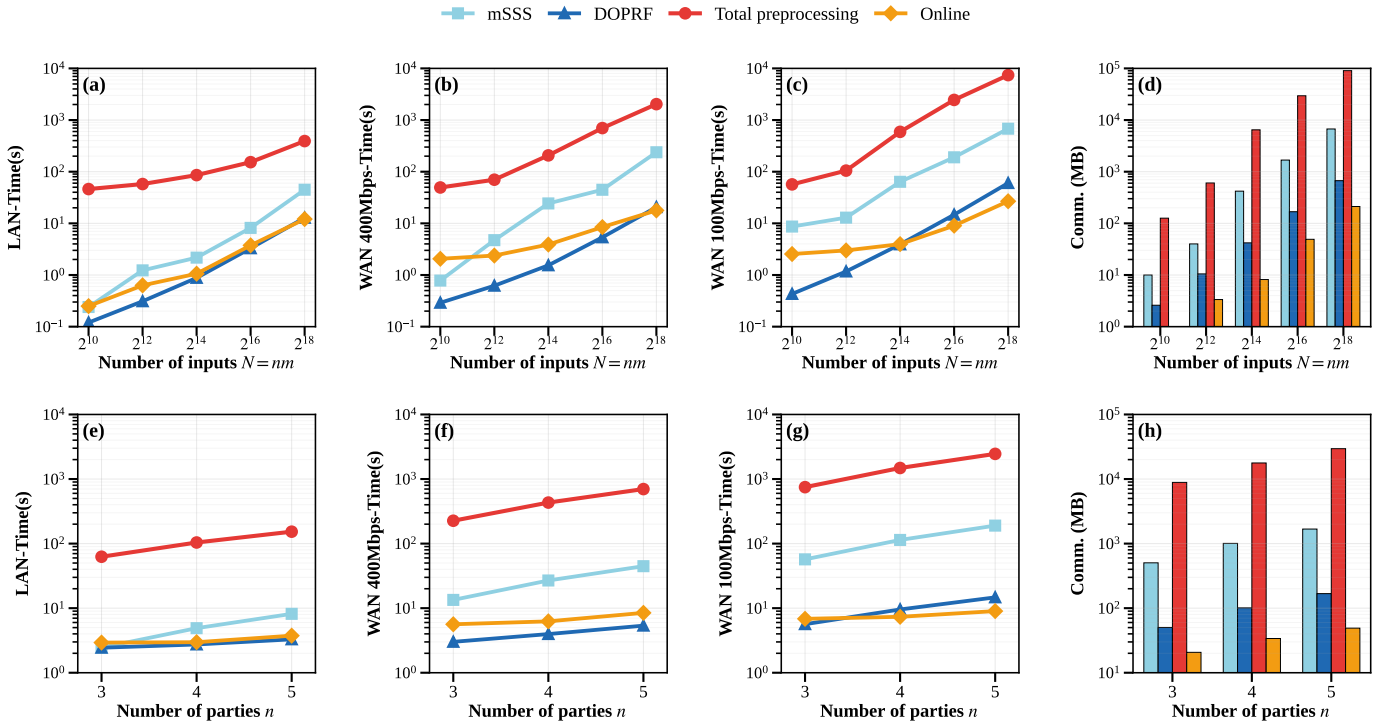

Fig. 9. **Scalability of UM-PSO.** Panels (a)–(d) evaluate input-size scalability with $n = 5$ parties and total input size $N = nm \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$, while panels (e)–(h) evaluate party scalability with a fixed total input size $N = 2^{16}$ and $n \in \{3, 4, 5\}$. Panels (a)–(c) and (e)–(g) report running time under LAN, WAN 400Mbps, and WAN 100Mbps, respectively, whereas panels (d) and (h) report communication cost, which is independent of the network setting. The curves report the costs of Π_{mSSS}, Π_{DOPRF}, the overall function-independent preprocessing phase, and the online computation phase, while communication costs are shown as grouped bars. All y-axes use logarithmic scales.

realization. Specifically, following the procedure of [4], each party locally shuffles its input items and then jointly invokes an oblivious sorting protocol (Radix sort) [41] to sort the combined items from all parties; neighboring sorted items are then compared to deduplicate entries and compute the desired set functions. Finally, the parties securely shuffle [42] the results to hide the order of the output, enabling the protocol to return the union or intersection in a random order. In addition, we compare against a state-of-the-art maliciously secure PSI protocol [20] and the state-of-the-art mPSU protocol [3]. Their implementations are available at https://github.com/asu-crypto/mPSI and https://github.com/real-world-cryptography/MPSO.

A. *Performance of* $\Pi_{\mathsf{UM-PSO}}$

We evaluate the scalability of UM-PSO with respect to both the total input size and the number of participating parties. Fig. 9 summarizes the running time and communication cost of the main components, including Π_{mSSS}, Π_{DOPRF}, the overall function-independent preprocessing phase, and the online set-operation phase. Specifically, the thread is set to 1 by default. The permutation method used in mSSS is formed by small permutations with T_d dimensions, where $T_d = 2^{10}$ in our experiments. Detailed numerical results under different network settings are illustrated in Appendix H.

Scalability with input size. Figs. 9(a)–(d) evaluate the effect of increasing the total input size $N = nm$ while fixing the number of parties to $n = 5$. Overall, the costs of all components increase smoothly with the input size. In particular, the costs of Π_{DOPRF} and the online comparison phase increase with the number of processed records, while the tree-structured Π_{mSSS} procedure remains the dominant component of the preprocessing phase, especially under WAN settings. Nevertheless, increasing the input size does not introduce a corresponding increase in the number of communication rounds, since the depth of the tree structure grows only logarithmically with N. As a result, UM-PSO can efficiently scale to relatively large input sets.

Scalability with the number of parties. Figs. 9(e)–(h) fix the total input size to $N = 2^{16}$ and vary the number of participating parties from 3 to 5. Compared with increasing the input size, increasing the number of parties results in a more pronounced growth in both running time and communication. This effect is particularly visible for Π_{mSSS}, since the underlying malicious mSSS requires interactions among an increasing number of parties. The DOPRF and online comparison phases exhibit the same trend, although their absolute costs remain substantially smaller than the overall preprocessing cost.

Overall, the results show that the performance of UM-PSO is more sensitive to the number of participating parties than to the total input size. This observation is consistent with our complexity analysis: the preprocessing communication has a quadratic dependence on the number of parties, while its dependence on the number of input records is linear up to a

TABLE III
End-to-end comparison with the generic malicious SS-based baseline. UM-PSO includes function-independent preprocessing and one online operation. Runtime is reported in seconds and communication in MB. T/O denotes timeout.

Network	n	Method	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
LAN	3	**Baseline**	44.2	127	363	1040	2982
		UM-PSO	11.5	16.1	28.2	65.5	205
		Speedup	3.84×	7.89×	12.87×	15.88×	14.55×
	4	**Baseline**	63.2	132	384	1408	5503
		UM-PSO	24.3	32.0	53.2	107	296
		Speedup	2.60×	4.12×	7.22×	13.16×	18.59×
	5	**Baseline**	91	221	703	2466	8849
		UM-PSO	46.4	58.3	86.9	157	404
		Speedup	1.96×	3.79×	8.09×	15.71×	21.90×
WAN 400Mbps	3	**Baseline**	760	1115	3023	9500	T/O
		UM-PSO	15.1	23.7	68.5	232	701
		Speedup	50.33×	47.05×	44.13×	40.95×	–
	4	**Baseline**	844	1331	4100	13500	T/O
		UM-PSO	28.7	42.8	130	438	1249
		Speedup	29.41×	31.10×	31.54×	30.82×	–
	5	**Baseline**	1100	1981	5971	T/O	T/O
		UM-PSO	51.6	72.4	211	707	2049
		Speedup	21.32×	27.36×	28.30×	–	–
WAN 100Mbps	3	**Baseline**	879	1536	5441	14510	T/O
		UM-PSO	17.1	33.9	182	756	2314
		Speedup	51.52×	45.32×	29.88×	19.20×	–
	4	**Baseline**	895	1868	6891	T/O	T/O
		UM-PSO	33.1	63.0	358	1490	4363
		Speedup	27.06×	29.64×	19.23×	–	–
	5	**Baseline**	1273	2873	13136	T/O	T/O
		UM-PSO	59.4	108	595	2462	7438
		Speedup	21.45×	26.60×	22.08×	–	–
Overhead	n	**Method**	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
Comm.	3	**Baseline**	2901	11855	55915	259463	T/O
		UM-PSO	38.2	182	1942	8889	27268
		Reduction	76.04×	64.99×	28.79×	29.19×	–
	4	**Baseline**	3000	14630	69043	320426	T/O
		UM-PSO	76.2	365	3886	17772	52537
		Reduction	39.38×	40.05×	17.77×	18.03×	–
	5	**Baseline**	8054	39289	185472	860656	T/O
		UM-PSO	127	608	6475	29613	90751
		Reduction	63.50×	64.58×	28.64×	29.06×	–

logarithmic factor. Consequently, UM-PSO is particularly suitable for cross-silo applications involving a moderate number of organizations, each holding a relatively large private dataset.

Online Set Operations. Once the function-independent preprocessing phase has been completed, different set functionalities can be evaluated over the same reusable pool through lightweight secure comparisons. As shown in Table VI, the online cost remains substantially lower than the preprocessing cost across all evaluated settings. This separation is particularly beneficial when multiple set operations are performed over the same fixed input sets, since the expensive function-independent preprocessing phase is executed only once and can be reused across subsequent queries.

Input-independent Setup As in SPDZ-style MPC, UM-PSO requires input-independent correlated randomness (CR), including Beaver triples, edaBits, and shuffle correlations. Following prior works [40] and standard benchmarking practice for preprocessing-based MPC, we report CR generation separately from the input-dependent execution costs, shown in Appendix I.

TABLE IV
Running time (in seconds) under LAN and communication cost (in MB) of NTY21 [20], DCZB25 [3], and our online phase with $n = 4$ parties.

Protocol	Functions	Reusable	N	Times	Comm.
NTY21 [20] malicious†	mPSI	No	2^{10}	0.07	0.49
			2^{14}	0.13	6.94
			2^{18}	1.87	119
DCZB25 [3] semi-honest	mPSO	No	2^{10}	0.07	0.25
			2^{14}	0.10	3.62
			2^{18}	1.03	59.5
UM-PSO malicious majority	mPSO	Yes	2^{10}	0.23	0.58
			2^{14}	0.80	5.65
			2^{18}	11.8	146

† Specialized mPSI construction under a client–server–pivot architecture; see Appendix F for its corruption assumptions.

B. Comparison with Baseline Protocols

Table III presents the comparison results between UM-PSO and the baseline protocol that extends BA12 [2]. The reported results of UM-PSO include both the preprocessing phase and the online phase to ensure a fair comparison. From the results, UM-PSO achieves significantly better performance, with up to a $51.5\times$ speedup and up to a $76\times$ reduction in communication, over the generic malicious baseline. The advantage becomes especially pronounced in high-latency WAN settings, since the most time-consuming component of the baseline is the oblivious sorting of nm 128-bit items. Based on radix sort, this sorting requires $\ell = 128$ rounds of shuffling, resulting in a large fixed round complexity whose cost is amplified by network latency. Consequently, the baseline becomes particularly latency-bound in WAN settings. As the input size increases, however, communication cost and hence bandwidth cost become increasingly dominant, reducing the relative runtime gap between the two approaches.

Table IV shows the comparison results between UM-PSO and the maliciously secure mPSI protocol NTY21 [20]. Note that the malicious setting of NTY21 differs from the standard malicious-majority MPC framework. Its security relies on OKVS-based techniques instantiated in the random oracle model and requires that the pivot and the servers are not all corrupted. Therefore, the running time and communication cost of their multiparty PSI protocol under the malicious setting are nearly identical to those of their semi-honest mPSI protocol, due to the specific malicious model they employ. Compared with NTY21, our online phase incurs roughly $3.3\times$ to $8.6\times$ higher running time under LAN, while keeping communication in a comparable range ($0.51\times$ to $1.37\times$).

In addition, we compare UM-PSO with DCZB25 [3], the state-of-the-art mPSO protocol in the semi-honest model, as reported in Table IV. We adopt this semi-honest baseline because, to the best of our knowledge, no practical maliciously secure mPSU protocol is currently available for comparison. Compared with DCZB25, UM-PSO incurs on average about $8.7\times$ higher running time and about $2.3\times$ higher communication cost.

VI. Conclusion

In this paper, we present UM-PSO, a general framework for the mPSO problem under the malicious majority model. As a unified framework, UM-PSO is capable of securely computing a variety of set operations as well as general functions over private sets. Nevertheless, the efficiency of the preprocessing phase in UM-PSO remains a bottleneck to its practical deployment, and we leave further optimization as future work.

References

[1] L. Kissner and D. Song, "Privacy-preserving set operations," in *Annual International Cryptology Conference*. Springer, 2005, pp. 241–257.

[2] M. Blanton and E. Aguiar, "Private and oblivious set and multiset operations," in *Proceedings of the 7th ACM symposium on information, computer and communications security*, 2012, pp. 40–41.

[3] M. Dong, Y. Chen, C. Zhang, Y. Bai, and Y. Cao, "Multi-party private set operations from predicative zero-sharing," in *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security*, 2025, pp. 4079–4093.

[4] Y. Huang, D. Evans, and J. Katz, "Private set intersection: Are garbled circuits better than custom protocols?" in *NDSS*, 2012.

[5] V. Kolesnikov, N. Matania, B. Pinkas, M. Rosulek, and N. Trieu, "Practical multi-party private set intersection from symmetric-key techniques," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 1257–1272.

[6] N. Chandran, N. Dasgupta, D. Gupta, S. L. B. Obbattu, S. Sekar, and A. Shah, "Efficient linear multiparty psi and extensions to circuit/quorum psi," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1182–1204.

[7] J. Su, Z. Chen, and X. Yang, "Multi-party private set intersection: A circuit-based protocol with jaccard similarity for secure and efficient anomaly detection in network traffic," in *Proceedings of the 2024 3rd International Conference on Cryptography, Network Security and Communication Technology*, 2024, pp. 361–366.

[8] Y. Gao, Y. Luo, L. Wang, X. Liu, L. Qi, W. Wang, and M. Zhou, "Efficient scalable multi-party private set intersection (-variants) from bicentric zero-sharing," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4137–4151.

[9] X. Yang, L. Cai, Y. Wang, K. Yin, L. Sun, and J. Hu, "Efficient unbalanced quorum psi from homomorphic encryption," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1003–1016.

[10] X. Liu and Y. Gao, "Scalable multi-party private set union from multi-query secret-shared private membership test," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2023, pp. 237–271.

[11] M. Dong, C. Zhang, Y. Bai, and Y. Chen, "Efficient {Multi-Party} private set union without {Non-Collusion} assumptions," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 2005–2024.

[12] J. Gao, S. Nguyen, M. Blanton, and N. Trieu, "Pulse: Parallel private set union for large-scale entities," in *Proceedings of the 2025 ACM SIGSAC Conference on Com-*

puter and Communications Security, 2025, pp. 1784–1798.

[13] Y. Lindell and B. Pinkas, "Privacy preserving data mining," in *Annual international cryptology conference*. Springer, 2000, pp. 36–54.

[14] A. Goel, P. Miao, P. V. L. Pham, and S. Singh, "Pics: Private intersection over committed (and reusable) sets," *Cryptology ePrint Archive*, 2025, to be appear at USENIX Security 2026.

[15] A. Bhowmick, D. Boneh, S. Myers, K. Talwar, and K. Tarbe. (2021) The apple psi system. [Online]. Available: https://www.apple.com/child-safety/pdf/Apple_PSI_System_Security_Protocol_and_Analysis.pdf

[16] A. Bonifati and R. Tommasini, "An overview of continuous querying in (modern) data systems," in *Companion of the 2024 International Conference on Management of Data*, 2024, pp. 605–612.

[17] Y. Yang, X. Liang, X. Song, Y. Dong, L. Huang, H. Ren, C. Dong, and J. Zhou, "Maliciously secure circuit private set intersection via spdz-compatible oblivious prf," *Proceedings on Privacy Enhancing Technologies*, 2025.

[18] S. Pu, J. Gao, and N. Trieu, "Malicious private set union with two-sided output," *Cryptology ePrint Archive*, 2026.

[19] P. Miao, S. Patel, M. Raykova, K. Seth, and M. Yung, "Two-sided malicious security for private intersection-sum with cardinality," in *Annual International Cryptology Conference*. Springer, 2020, pp. 3–33.

[20] O. Nevo, N. Trieu, and A. Yanai, "Simple, fast malicious multiparty private set intersection," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 1151–1165.

[21] A. Ben-Efraim, O. Nissenbaum, E. Omri, and A. Paskin-Cherniavsky, "Psimple: Practical multiparty maliciously-secure private set intersection," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022, pp. 1098–1112.

[22] Y. Gao, X. Zheng, and C. Hu, "A multi-party private set union protocol against malicious adversary," in *International Conference on Innovative Computing*. Springer, 2024, pp. 159–167.

[23] M. Keller, "Mp-spdz: A versatile framework for multi-party computation," in *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*, 2020, pp. 1575–1590.

[24] M. Keller, V. Pastro, and D. Rotaru, "Overdrive: Making spdz great again," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2018, pp. 158–189.

[25] C. Hazay and M. Venkitasubramaniam, "Scalable multi-party private set-intersection," in *IACR international workshop on public key cryptography*. Springer, 2017, pp. 175–203.

[26] R. Poddar, S. Kalra, A. Yanai, R. Deng, R. A. Popa, and J. M. Hellerstein, "Senate: a {Maliciously-Secure}{MPC} platform for collaborative analytics," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 2129–2146.

[27] M. Wu, T. H. Yuen, and K. Y. Chan, "{O-Ring} and {K-Star}: Efficient multi-party private set intersection," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 6489–6506.

[28] Y. Lindell, "How to simulate it–a tutorial on the simulation proof technique," *Tutorials on the Foundations of Cryptography: Dedicated to Oded Goldreich*, pp. 277–346, 2017.

[29] D. Escudero, S. Ghosh, M. Keller, R. Rachuri, and P. Scholl, "Improved primitives for mpc over mixed arithmetic-binary circuits," in *Annual international cryptology conference*. Springer, 2020, pp. 823–852.

[30] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference*. Springer, 2012, pp. 643–662.

[31] A. Beimel, "Secret-sharing schemes: A survey," in *International conference on coding and cryptology*. Springer, 2011, pp. 11–46.

[32] M. Chase, E. Ghosh, and O. Poburinnaya, "Secret-shared shuffle," in *Advances in Cryptology–ASIACRYPT 2020: 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7–11, 2020, Proceedings, Part III 26*. Springer, 2020, pp. 342–372.

[33] S. Eskandarian and D. Boneh, "Clarion: Anonymous communication from multiparty shuffling protocols," *Cryptology ePrint Archive*, 2021.

[34] X. Song, D. Yin, J. Bai, C. Dong, and E.-C. Chang, "Secret-shared shuffle with malicious security," *Cryptology ePrint Archive*, 2023.

[35] E. Makri, D. Rotaru, F. Vercauteren, and S. Wagh, "Rabbit: Efficient comparison for secure multi-party computation," in *International conference on financial cryptography and data security*. Springer, 2021, pp. 249–270.

[36] Y. Dodis and A. Yampolskiy, "A verifiable random function with short proofs and keys," in *International Workshop on Public Key Cryptography*. Springer, 2005, pp. 416–431.

[37] S. Casacuberta, J. Hesse, and A. Lehmann, "Sok: Oblivious pseudorandom functions," in *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. Los Alamitos, CA, USA: IEEE Computer Society, 2022, pp. 625–646.

[38] M. J. Freedman, Y. Ishai, B. Pinkas, and O. Reingold, "Keyword search and oblivious pseudorandom functions," in *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005. Proceedings 2*. Springer, 2005, pp. 303–324.

[39] N. Tyagi, S. Celi, T. Ristenpart, N. Sullivan, S. Tessaro, and C. A. Wood, "A fast and simple partially oblivious prf, with applications," in *Annual International Conference on the Theory and Applications of Cryptographic*

Techniques. Springer, 2022, pp. 674–705.

[40] B. Yuan, S. Yang, Y. Zhang, N. Ding, D. Gu, and S.-F. Sun, "{MD-ML}: Super fast {Privacy-Preserving} machine learning for malicious security with a dishonest majority," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 2227–2244.

[41] K. Hamada, D. Ikarashi, K. Chida, and K. Takahashi, "Oblivious radix sort: An efficient sorting algorithm for practical secure multi-party computation," Cryptology ePrint Archive, Paper 2014/121, 2014. [Online]. Available: https://eprint.iacr.org/2014/121

[42] M. Keller and P. Scholl, "Efficient, oblivious data structures for mpc," in *Advances in Cryptology – ASIACRYPT 2014*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 506–525.

[43] M. Keller, E. Orsini, and P. Scholl, "Mascot: faster malicious arithmetic secure computation with oblivious transfer," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 830–842.

[44] E. Boyle, G. Couteau, N. Gilboa, Y. Ishai, L. Kohl, and P. Scholl, "Efficient pseudorandom correlation generators: Silent ot extension and more," in *Annual International Cryptology Conference*. Springer, 2019, pp. 489–518.

[45] X. Guo, H. Liu, Z. Huang, H. Cui, W. Zhang, C. Hong, X. Wang, K. Yang, and Y. Yu, "Dory: streaming pcg with small memory," *Cryptology ePrint Archive*, 2025.

[46] Y. Liu, Y. Kang, T. Zou, Y. Pu, Y. He, X. Ye, Y. Ouyang, Y.-Q. Zhang, and Q. Yang, "Vertical federated learning: Concepts, advances, and challenges," *IEEE Transactions on Knowledge and Data Engineering*, vol. 36, no. 7, pp. 3615–3634, 2024.

[47] L. Lu and N. Ding, "Multi-party private set intersection in vertical federated learning," in *2020 IEEE 19th international conference on trust, security and privacy in computing and communications (trustcom)*. IEEE, 2020, pp. 707–714.

[48] R. Jia, D. Dao, B. Wang, F. A. Hubis, N. Hynes, N. M. Gürel, B. Li, C. Zhang, D. Song, and C. J. Spanos, "Towards efficient data valuation based on the shapley value," in *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR, 2019, pp. 1167–1176.

[49] S. Zheng, Y. Cao, and M. Yoshikawa, "Secure shapley value for cross-silo federated learning," *Proceedings of the VLDB Endowment*, vol. 16, no. 7, pp. 1657–1670, 2023.

[50] H. Xia, X. Li, J. Pang, J. Liu, K. Ren, and L. Xiong, "P-shapley: Shapley values on probabilistic classifiers," *Proceedings of the VLDB Endowment*, vol. 17, no. 7, pp. 1737–1750, 2024.

[51] D. Boneh, E. Boyle, H. Corrigan-Gibbs, N. Gilboa, and Y. Ishai, "Lightweight techniques for private heavy hitters," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 762–776.

[52] Y. Zhang, Q. Ye, and H. Hu, "Federated heavy hitter analytics with local differential privacy," *Proceedings of the ACM on Management of Data*, vol. 3, no. 1, pp. 1–27, 2025.

APPENDIX

A. Generative AI Usage

We used ChatGPT only for editorial assistance, including grammar checking, spelling correction, and light language polishing. The tool was not used to generate technical ideas, proofs, algorithms, experimental results, figures, or citations. All AI-assisted text was critically reviewed and verified by the authors, who retain full responsibility for the accuracy, originality, and integrity of the submission.

Functionality $\mathcal{F}_{\mathsf{mOSS}}$

Parameters: n parties P_i $(i \in [1,n])$, $\mathcal{P} = \{P_1, \cdots, P_n\}$; a dictionary Val storing all authenticated values; m input items.

Inputs: A party P_i provides a permutation method $\pi \in \mathbf{S}$; the identifiers $\{id_j\}_{j\in[m]}$ for the values being shuffled, and the the identifiers $\{id'_j\}_{j\in[m]}$ for the shuffled values to be stored with.

Functionality: receiving $(\mathsf{mOSS}, \pi, \{id_j\}_{j\in[m]}, \{id'_j\}_{j\in[m]})$ from P_i and $(\mathsf{Shuffle}, \{id_j\}_{j\in[m]}, \{id'_j\}_{j\in[m]})$ from $\mathcal{P}\backslash\{P_i\}$:

- If $\pi \notin \mathbf{S}$, abort;
- Define a vector $\mathbf{x}$ such that $\mathbf{x}[j] = \mathsf{Val}[id_j]$ for $j \in [1,m]$;
- Compute $\mathbf{y} = \pi(\mathbf{x})$ and store $\mathsf{Val}[id'_j] \leftarrow \mathbf{y}[j]$ for $j \in [1,m]$.

Fig. 10. The functionality of mOSS.

B. Multi-party Secret Shared Shuffle [34]

A multi-party secret shared shuffle protocol with linear complexity is introduced in [33]. Following the pair-wise execution paradigm from the existing works [33, 34], we describe how to achieve a multiparty SSS protocol Π_{mSSS} based on a two-party SSS protocol under the malicious setting. For n parties, Π_{mSSS} requires n^2 invocations of pair-wise CGP shuffle [32]. It will generate $O(\kappa n^2 m \log m)$ (offline) and $O(\ell n^2 m)$ (online) communication costs.

Technically, Π_{mSSS} is constructed from multiparty one-sided secret shared shuffle functionality $\mathcal{F}_{\mathsf{mOSS}}$. We will give the ideal functionality of $\mathcal{F}_{\mathsf{mOSS}}$ as shown in Figure 10. With the n-party OSS protocol Π_{mOSS}, the construction of a n-party SSS protocol Π_{mSSS} is shown in Figure 11.

C. Functionality of edaBits

The extended doubly authenticated shared bits (edaBits) [29, 35] aims to generate shared bits living both in $\mathbb{F}_p$ and in $\mathbb{F}_{2^k}$. The functionality $\mathcal{F}_{\mathsf{edaBits}}$ is shown in Figure 12.

Functionality $\mathcal{F}_{\mathsf{mSSS}}$

Parameters: T denotes cut-and-choose bucket number and B denotes cut-and-choose bucket size;

Offline: On inputting T and B, do the following:

- For $i \in [n]$, the parties run $\Pi^{[P_i]}_{\mathsf{mOSS}}.\mathsf{Offline}(T, B)$, where P_i provides TB permutations.

Online: On inputting sharings $\langle \mathbf{x} \rangle$, do the following:

- Let $\langle \mathbf{x}^{(1)} \rangle \leftarrow \langle \mathbf{x} \rangle$;
- For $i \in [1, n]$, run $\langle \mathbf{x}^{i+1} \rangle \leftarrow \Pi^{[P_i]}_{\mathsf{mOSS}}.\mathsf{Online}(\langle \mathbf{x}^{(i)} \rangle)$;
- Output $\langle \mathbf{x}^{n} \rangle$.

Fig. 11. The functionality of mSSS.

Functionality $\mathcal{F}_{\mathsf{edaBits}}$

The edaBits functionality contains the following steps:

- **edaBits**: On input $(\mathsf{edaBits}, \ell)$ from all parties, this functionality samples a uniformly random value r with ℓ bits, where $r = \sum_{i=1}^{\ell} 2^{i-1} r[i]$, and distributes the sharings $\{\langle r[i] \rangle\}_{i=1}^{\ell}$ and $\langle r \rangle^p$ to all parties if it does not abort.

Fig. 12. The functionality of edaBits.

D. Secure Comparison

The functionality $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Compare}$ aims to compare a shared item with a constant value in the arithmetic domain, as shown in Figure 3. It utilizes the $\mathcal{F}_{\mathsf{edaBits}}$ functionality shown in Section C. Based on edaBits, the secure comparison protocol can be achieved by the functionality of Less Than Constant (LTC). We show $\mathcal{F}_{\mathsf{LST}}$ as in Figure 13. In this functionality, the Π_{LTBits} aims to compare an input shared bitwise and a public value [35].

E. Correctness

The correctness of $\Pi_{\mathsf{UM-PSO}}$ comes from two aspects:

- **Duplicate Removal.** All duplicates of x are eliminated, ensuring that x appears only once in the final group $G_{H,1}$ in Step 7 of Figure 8. At Level $h = 1$, singleton child groups are first compared and deduplicated. By induction, after completing Level h, each resulting group contains at most one valid occurrence of each item. Therefore, after Level H, each distinct valid item appears exactly once in $G_{H,1}$.
- **Multiplicity Counting.** When eliminating the duplicate x, the identical items will be compared, and each successful comparison adds their payload values. Specifically, when two identical items x and x' are detected in neighboring groups, their payloads are combined as $\mathrm{Count}(x) \leftarrow \mathrm{Count}(x) + \mathrm{Count}(x')$. x' is then replaced by a fresh random value r, where $\mathrm{Count}(r)$ is set to 0, while x keeps the accumulated payload. Therefore, the final payload equals the true multiplicity $\mathrm{Count}(x)$.

Functionality $\mathcal{F}_{\mathsf{LTC}}$

Inputs: Values x are secret shared between n parties, such that each party holds $\langle x \rangle$ and a constant value R; a shared $\mathsf{edaBit}(\langle r \rangle^p, \langle r[0] \rangle^2, \cdots, \langle r[\ell-1] \rangle^2)$, where $\langle \cdot \rangle^2$ represents boolean share, $r = \sum_{i=0}^{\ell-1} r[i] \cdot 2^i$.
Outputs: Compute the Boolean value $\langle (x < R) \rangle^2$.

Protocol:

- Parties compute the values $\langle a \rangle = \langle x + r \rangle, \langle b \rangle = \langle x + r + p - R \rangle$;
- Parties open the value a, and b can be opened locally as $b = a + p - R$;
- Parties compute the following quantities:
  - $\langle w_1 \rangle^2 = \Pi_{\mathsf{LTBits}}(a, \langle r[0] \rangle^2, \cdots, r[\ell-1]^2)$;
  - $\langle w_2 \rangle^2 = \Pi_{\mathsf{LTBits}}(b, \langle r[0] \rangle^2, \cdots, r[\ell-1]^2)$;
  - $w_3 = (b < p - R)$.
- Output $\langle w \rangle^2 = 1 - (\langle w_1 \rangle^2 - \langle w_2 \rangle^2 + \langle w \rangle_3)$.

Fig. 13. The functionality of LTC.

F. Related Work

KS05 [1] proposes one of the earliest constructions for multi-party private set operations. They utilize the polynomial representation to design a private set operation protocol based on homomorphic encryption. For each party P_i $(1 \le i \le n)$ with an input set $\mathbf{x}_i = \{x_{i,1}, \cdots, x_{i,m}\}$, each item in $\mathbf{x}_i$ plays the root of a polynomial f_i. So, the roots of the polynomial $\prod_{i=1}^{n} f_i$ are the union set $\bigcup_{i=1}^{n} \{\mathbf{x}_i\}$, and the roots of $\sum_{i=1}^{n} f_i$ denotes the intersection $\bigcap_{i=1}^{n} \{\mathbf{x}_i\}$.

KMPR17 [5] adheres to an *align-program-obtain* workflow. First, a leading party will use cuckoo hashing to map all elements in its input set, and other parties will use simple hashing to *align* the elements. Then, each party $P_{i, i \in [1,n]}$ randomly generates n zero shares for each item $x_{i,j}$ $(j \in [1, m])$ in input set $\mathbf{x}_i$, i.e., $\gamma^1_{i,j} \oplus \cdots \oplus \gamma^n_{i,j}$. Then, P_i *program* each element with corresponding shares as $(x_{i,j}, \gamma^{\mu}_{i,j})$, and then perform an Oblivious Programmable PRF (OPPRF) protocol with the other party P_u. If $x_{i,j} \in X_u$, then P_u obtains $\gamma^{\mu}_{i,j}$. For elements in the intersection set, the sum of corresponding shares is 0; this idea is known as *zero-sharing*. Finally, the leading party will announce all sums of shares and *obtain* them to zero to get the intersection set.

LG23 [10] introduces a practical dimension to the domain of multi-party PSU. The core idea of their protocol is to split $\bigcup_{i=1}^{n} \mathbf{x}_i$ into $\mathbf{x}_1 \cup (\mathbf{x}_2 \backslash \mathbf{x}_1) \cup \cdots \cup (\mathbf{x}_n \backslash (\mathbf{x}_1 \cup \cdots \cup \mathbf{x}_{n-1}))$. For each part $\mathbf{x}_n \backslash (\mathbf{x}_1 \cup \cdots \cup \mathbf{x}_{n-1})$, it is equivalent to $(\mathbf{x}_n \backslash \mathbf{x}_1) \cap \cdots \cap (\mathbf{x}_n \backslash \mathbf{x}_{n-1})$. Therefore, the authors utilize several underlying protocols, Private Membership Test (PMT) and Oblivious Transfer (OT), so that P_i can invoke those protocols with $P_1, \cdots, P_{i-1}$ to compute $(\mathbf{x}_i \backslash \mathbf{x}_1), \cdots, (\mathbf{x}_i \backslash \mathbf{x}_{i-1})$, respectively. And the results are shared between parties. Before the other parties send all shares to P_1 (as the leader) to recover the results, all parties will use a multiparty secret shared shuffle (mSSS) protocol to shuffle those shares. This step ensures that P_1 does not learn which parties the difference sets are coming from. As we can see from those steps, if P_1 colludes with

Protocol $\Pi_{\mathsf{UM-PSO}}$

Parameters: Party P_i $(i \in [1,n])$ has an input set $\mathbf{x}_i = \{x_{i,1}, \cdots, x_{i,m}\}$, where $\mathbf{x}_i \in \mathbb{Z}_p^m$, $N = nm$; the length of each item in $\mathbf{x}_i$ is ℓ; a shared master key $\langle mk \rangle$; a set of update PRF keys shares $\{\langle uk_{1,1} \rangle, \cdots, \langle uk_{1,\frac{nm}{2}} \rangle\}$; the level $h \in [1,H]$, $H = \log_2(N)$.
Input-independent setup: The parties generate sufficient fresh authenticated correlated randomness for the subsequent executions. Every correlation is consumed at most once.

Function-independent preprocessing: On inputting $\mathbf{x}_1, \cdots, \mathbf{x}_n$, parties do the following:

1. Level-0 shuffle: P_i shares input sets $\{\langle x_{1,1} \rangle, \cdots, \langle x_{n,m} \rangle\}$, and partition the shares into nm groups, denoted by $\vec{G_0} = \{G_{0,1}, \cdots, G_{0,nm}\}$, and one share in one group, i.e., $G_{0,(i-1)m+j}.\mathrm{val} = \{x_{i,j}\}$, where $i \in [1,n], j \in [1,m]$. Then parties invoke $\mathcal{F}_{\mathsf{mSSS}}$ and takes the shares in $\vec{G_0}$ and π_0 as input items, and obtain shuffled shared set $\tilde{G}_{0,(i-1)m+j}.\mathrm{val} = \{x_{\pi_0(i,j)}\}$;
2. For each round $h = 1$ to $\log_2(nm)$, do:
   a) **if** $h = 1$ **then**
   For $u \in [1, \frac{N}{2}]$, parties call $\mathcal{F}_{\mathsf{DOPRF}}.\mathsf{Eval}$ with keys mk on shared items in $\tilde{G}_{0,2u-1}$ and $\tilde{G}_{0,2u}$ to obtain valid internal states. Then they invoke $\mathcal{F}_{\mathsf{DOPRF}}.\mathsf{Update}$ on these states using the fresh update key $\langle uk_{1,u} \rangle$ for the two neighboring groups, and open the updated PRF values;
   b) **else do**: For $u \in [1, \frac{nm}{2^h}]$, parties call $\mathcal{F}_{\mathsf{DOPRF}}.\mathsf{Update}$ to randomize all PRF values in the group $\tilde{G}_{h-1,2u-1}$ with shared key sets $\{\langle uk_{1,\delta} \rangle | (2u-1)2^{h-2} + 1 \leq \delta \leq u2^{h-1}\}$, and randomize all PRF values in the group $\tilde{G}_{h-1,2u}$ with shared key sets $\{\langle uk_{1,\delta} \rangle | (u-1)2^{h-1} + 1 \leq \delta \leq (2u-1)2^{h-2}\}$;
   c) For $u \in [1, \frac{nm}{2^h}]$, $w_0, w_1 \in [1, 2^{h-1}]$, parties locally compare PRF values $\tilde{G}_{h-1,2u-1}[w_0].\mathrm{PRF}$ and $\tilde{G}_{h-1,2u}[w_1].\mathrm{PRF}$. If they are equal, then parties add corresponding payload shares, i.e., $\mathrm{Count}(\tilde{G}_{h-1,2u-1}[w_0].\mathrm{val}) \leftarrow \mathcal{F}_{\mathsf{AuSS}}.\mathrm{LinComb}(\mathrm{Count}(\tilde{G}_{h-1,2u-1}[w_0].\mathrm{val}), \mathrm{Count}(\tilde{G}_{h-1,2u}[w_1].\mathrm{val}))$, and substitute $\tilde{G}_{h-1,2u}[w_1].\mathrm{val}$ with a random share $\langle r_\Delta \rangle \xleftarrow{\$} \mathbb{R}$;
   d) Level-h shuffle: For $u \in [1, \frac{nm}{2^h}]$, parties combine the groups $\tilde{G}_{h-1,2u-1}$ and $\tilde{G}_{h-1,2u}$ as a new group $G_{h,u}$, and invoke $\mathcal{F}_{\mathsf{mSSS}}$, and takes the shares in $\vec{G_h} = \{G_{h,1}, \cdots, G_{h,\frac{nm}{2^h}}\}$ and π_h as input items, and obtain shuffled shared set $\{\tilde{G}_{h,1}, \cdots, \tilde{G}_{h,\frac{nm}{2^h}}\}$;
3. Parties obtain the root group $G_{H,1}$, and prepare an empty set $\vec{Res}$.

Online set operations:

4. Parties choose the operation type:
$$\mathsf{op} \in \{\mathrm{mPSU}, \mathrm{mPSI}, \mathrm{quorumPSI}, \mathrm{circuit\text{-}mPSI}, \mathrm{circuit\text{-}mPSU}\}.$$
5. Parties set the threshold thr as follows:
   - mPSU, circuit-mPSU: $thr = 1$;
   - mPSI, circuit-mPSI: $thr = n$;
   - quorum PSI: $1 < thr < n$.
6. Parties call $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Compare}$ to compare $\mathrm{Count}(\tilde{G}_{\log_2(nm),1}[u].\mathrm{val})$ and thr, where $u \in [1, nm]$. The resulting shares are stored in $\vec{Res}$;
7. If $\mathsf{op} \in \{\mathrm{mPSU}, \mathrm{mPSI}, \mathrm{quorum\ PSI}\}$, then parties reconstruct the shares by invoking $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{OpenCheck}(\vec{Res})$. Otherwise, if $\mathsf{op} \in \{\mathrm{circuit\text{-}mPSI}, \mathrm{circuit\text{-}mPSU}\}$, the parties invoke $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Evaluate}$ to evaluate f on the private items in $\tilde{G}_{H,1}$ for which $b_j = 1$.

Fig. 14. The optimized preprocessing phase in UM-PSO.

any P_i, e.g., P_3, for $x \in \mathbf{x}_3$, P_1 can recover the intermediary results $(\mathbf{x}_3 \backslash \mathbf{x}_1) \cap (\mathbf{x}_3 \backslash \mathbf{x}_2)$ to deduce whether $x \in \mathbf{x}_2$ or not. So, it will reveal information about P_2.

Next, DZBC25 [11] proposes an mPSU protocol without non-collusion assumptions. Their protocol follows the same idea of LG23 [10]. The difference is that the intermediary results, e.g., $(X_3 \backslash X_1) \cap (X_3 \backslash X_2)$, will be re-shared between all parties rather than only P_1, P_2, P_3. Specifically, each party P_j has computed PMT with other parties P_i $(1 \leq i < j \leq n)$ to obtain $\mathbf{y}_i = \mathbf{x}_j \backslash (\mathbf{x}_1 \cup \cdots \cup \mathbf{x}_{j-1})$. Instead of directly sending all $\mathbf{y}_i$ to P_1 to recover all results, it will be secret shared shuffled among all parties, and then sent to P_1, who will recover the union set.

DCZB25 [3] adopts the design of DZBC25 [11]. The parties invoke a predicate *zero-sharing* protocol: whenever a target element x belongs to (or does not belong to) a set $\mathbf{y}$, the protocol distributes shares of zero to all parties. To realize mPSI or mPSU, the parties first express the operation as a canonical predicate formula; any target item x that satisfies the formula yields shares summing to 0. For mPSI, the parties seek items x such that $x \in \mathbf{x}_1 \wedge \cdots \wedge x \in \mathbf{x}_n$, which is typically realized via OPRF. For mPSU, by contrast, the parties seek items satisfying $x \in \mathbf{x}_1 \vee (x \notin \mathbf{x}_1 \wedge x \in \mathbf{x}_2) \vee \cdots \vee (x \notin \mathbf{x}_1 \wedge x \in \mathbf{x}_n)$, which is commonly realized through Private Membership Testing (PMT) and OT [3, 10].

As we can conclude from the related works, most of them are based on OT to push the efficiency to its limit under the semi-honest model. While this model has a large number

of applications, some existing OT-based protocols cannot be securely used as building blocks in larger protocols, as they are not composable [2]. That is, revealing the results of prior private set computations to any party breaks composability and prevents further secure computations. Other protocols are designed to secret-share the results among parties before subsequent computations, but they are difficult to extend to the malicious setting.

G. Security Analysis

In this section, we will give a formal security proof of $\Pi_{\mathsf{UM-PSO}}$. We will first give proofs of sub-protocols Π_{DOPRF}. The security proof of Theorem 1.

Theorem 3. *In the $(\mathcal{F}_{\mathsf{AuSS}})$-hybrid model, the protocol Π_{DOPRF} implements $\mathcal{F}_{\mathsf{DOPRF}}$ correctly and securely against a malicious adversary who can control at most $t \leq n-1$ parties.*

Proof. Let $\mathcal{A}$ denote the adversary, $\mathcal{C}$ denote the set of corrupted parties, and $\mathcal{H}$ denote the set of honest parties. We construct a simulator $\mathcal{S}_{\mathsf{DOPRF}}$ in the ideal world.

Upon receiving a Eval invocation on $(\mathsf{id}_{mk}, \mathsf{id}_x, \mathsf{id}_y)$, $\mathcal{S}_{\mathsf{DOPRF}}$ forwards the invocation to $\mathcal{F}_{\mathsf{DOPRF}}$. Since all arithmetic values remain inside $\mathcal{F}_{\mathsf{AuSS}}$, the only input-independent values opened during Eval are d and the final PRF result. To simulate the first opening, $\mathcal{S}_{\mathsf{DOPRF}}$ samples $\hat{d} \xleftarrow{\$} \mathbb{F}_p^*$, and sends it to $\mathcal{A}$. $\hat{d}$ is identically distributed to the real value because for every $mk + x \neq 0$, $mk + x$ is multiplied by a random value $r \xleftarrow{\$} \mathbb{F}_p^*$: $r(mk + x)$.

For the second opening, $\mathcal{S}_{\mathsf{DOPRF}}$ invokes the ideal $\mathcal{F}_{\mathsf{DOPRF}}$ on the corresponding authenticated identifiers and obtains the prescribed group output $y = g^{1/(mk+x)}$. Then, $\mathcal{S}_{\mathsf{DOPRF}}$ forward the group output y to $\mathcal{A}$. If $\mathcal{A}$ completes the OpenCheck consistently, $\mathcal{S}_{\mathsf{DOPRF}}$ sends Deliver to $\mathcal{F}_{\mathsf{DOPRF}}$, and $\mathcal{F}_{\mathsf{DOPRF}}$ delivers y to $\mathcal{H}$; otherwise, $\mathcal{S}_{\mathsf{DOPRF}}$ forwards Abort, and $\mathcal{H}$ receive $\perp$.

The Update case is also handled in the same way: all intermediate exponents remain hidden, and $\mathcal{S}_{\mathsf{DOPRF}}$ returns the updated output y^{uk}. If $\mathcal{A}$ submits an invalid identifier, an incorrectly typed value, or causes an authenticated opening to fail, $\mathcal{F}_{\mathsf{AuSS}}$ aborts and $\mathcal{S}_{\mathsf{DOPRF}}$ forwards the same abort to $\mathcal{F}_{\mathsf{DOPRF}}$. If the opened value d is zero, both the real and ideal executions abort. Thus, conditioned on the same abort behavior, the simulated and real hybrid views are identically distributed. This proves that $\Pi_{\mathsf{DY-DOPRF}}$ securely realizes $\mathcal{F}_{\mathsf{DOPRF}}$ with abort.

Theorem 4. *In the $(\mathcal{F}_{\mathsf{AuSS}}, \mathcal{F}_{\mathsf{DOPRF}})$-hybrid model, the protocol $\Pi_{\mathsf{UM-PSO}}$ implements $\mathcal{F}_{\mathsf{UM-PSO}}$ correctly and securely against a malicious adversary who can control at most $t \leq n-1$ parties.*

Proof. Let $\mathcal{A}$ denote any PPT adversary who may corrupt a set of parties $\mathcal{C}$, where $|\mathcal{C}| = t \leq n-1$, and let $\mathcal{H}$ denote the set of honest parties $|\mathcal{H}| + |\mathcal{C}| = n$. Each party inputs a set with m items, where $N = nm$. We can construct a simulator $\mathcal{S}_{\mathsf{UM\text{-}PSO}}$ in the ideal world. $\mathcal{S}_{\mathsf{UM\text{-}PSO}}$ emulates $(\mathcal{F}_{\mathsf{UM-PSO}})$, interacts with the adversary $\mathcal{A}$. When any corrupted party in $\mathcal{C}$ aborts or $\mathcal{S}$ terminates the protocol, $\mathcal{S}_{\mathsf{UM-PSO}}$ needs to output whatever $\mathcal{C}$ outputs. Let $\widetilde{\mathbf{X}}_{\mathcal{C}} = \{\widetilde{x}_i\}_{i \in \mathcal{C}}$ denote the effective inputs committed by the corrupted parties, $\lambda_D = \mathcal{L}_D(\mathbf{X}; \rho)$, and $y_{\mathcal{C}} = f_{\mathsf{op}}(\mathbf{X})|_{\mathcal{C}}$ denote the prescribed output delivered to the corrupted parties for a set operation op. The ideal functionality samples the shuffle randomness ρ according to the same distribution as the real protocol. To preserve the leakage timing in executions with abort, it releases $E_{h,u}^{\mathbf{X},\rho}$ to the simulator only when the real protocol reaches node (h, u).

In the hybrid model, the simulator $\mathcal{S}_{\mathsf{UM-PSO}}$ does the following:

$\mathbf{Hybrid}_0$: This is the real execution of $\Pi_{\mathsf{UM-PSO}}$ in the $(\mathcal{F}_{\mathrm{AuSS}}, \mathcal{F}_{\mathrm{DOPRF}})$-hybrid model, given input vector $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n)$, with adversary $\mathcal{A}$ controlling the parties in $\mathcal{C}$. Let View_0 denote the resulting view of $\mathcal{A}$. For every corrupted party, its input is fixed through $\mathcal{F}_{\mathsf{AuSS}}$.Input and forwarded to $\mathcal{F}_{\mathsf{UM-PSO}}$. The honest parties' inputs remain hidden inside the ideal functionalities. Whenever $\mathcal{F}_{\mathsf{AuSS}}$.Shuffle is invoked, the simulator represents the shuffled records only by fresh anonymous positions. By the semantics of the ideal shuffle functionality, the permutation is hidden from the adversary. Hence, the simulator only maintains the public tree structure and anonymous positions, without maintaining a mapping from these positions to the honest parties' input items.

$\mathbf{Hybrid}_1$: For every deduplication node (h, u), we replace the DOPRF evaluated under the fresh node key by an independently sampled random function $R_{h,u} : \mathcal{X} \to \mathbb{G}$. For a fixed node, repeated evaluations of the same input return the same value, whereas random functions used at distinct nodes are independent. In the basic construction, each deduplication node uses an independently generated DOPRF key. Suppose that a PPT distinguisher can distinguish H_0 from H_1 with non-negligible advantage. By embedding a PRF challenge into one randomly selected deduplication node and simulating all remaining nodes normally, we obtain a distinguisher against the pseudorandomness of the underlying PRF. A standard hybrid argument over all deduplication nodes therefore gives $\mathsf{View}_0 \approx_c \mathsf{View}_1$.

$\mathbf{Hybrid}_2$: We next modify how the opened values at every deduplication node are generated. In this hybrid, they are simulated using only the randomized deduplication leakage $\mathcal{L}_D$. Consider a node (h, u) and let $E_{h,u} = E_{h,u}^{\mathbf{X},\rho}$. By induction over the tree levels, each child group contains at most one valid occurrence of every item. This property holds trivially for the initial singleton groups. At every internal node, if the same item occurs in both child groups, one copy is removed before the two groups are merged. Thus, the property is preserved at the next level.

Consequently, at node (h, u), every equality class consists either of a single unmatched position or of one position from each child group. Let $\mathcal{P}_{h,u} = \{L_1, \ldots, L_\ell, R_1, \ldots, R_r\}$ denote the anonymous positions in the two child groups. The simulator defines an equivalence relation over these positions by $L_a \sim R_b \iff E_{h,u}[a, b] = 1$, while every unmatched

position forms a singleton class.

For each equivalence class C, the simulator samples a fresh label $\tau_C \xleftarrow{\$} \mathbb{G}$ uniformly without replacement, and assigns τ_C to every position in C. Therefore, two simulated DOPRF outputs are equal if and only if the corresponding anonymous positions are matched according to $E_{h,u}$.

Conditioned on the event that the random function $R_{h,u}$ does not map two distinct inputs to the same group element, the distribution generated by the simulator is identical to the distribution in H_1. Let $Q_{h,u}$ denote the total number of DOPRF values opened at node (h,u). The probability of an accidental collision between distinct inputs is bounded by $\Pr[\mathsf{Collide}] \leq \sum_{h,u} \frac{Q_{h,u}(Q_{h,u}-1)}{2|\mathbb{G}|}$, which is negligible in κ. Accidental collisions among independently generated dummy values are likewise negligible and can be included in the same bad event. Therefore, $\mathsf{View}_1 \approx_c \mathsf{View}_2$.

In the online set operation phase, the comparisons are performed by $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Compare}$. Then, the downstream computation is performed by $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{Evaluate}$. All intermediate values remain secret shared and therefore reveal no additional information to the adversary.

Whenever the prescribed output must be reconstructed, the simulator obtains $y_{\mathcal{C}}$ from $\mathcal{F}_{\mathsf{UM-PSO}}$ and simulates the corresponding $\mathcal{F}_{\mathsf{AuSS}}.\mathsf{OpenCheck}$ execution so that the reconstructed value is exactly $y_{\mathcal{C}}$. Any failed authenticated opening or detected malicious deviation is translated into the same abort in the ideal execution.

In conclusion, Hybrid_2 is identical to the ideal execution with leakage $\mathcal{L}_D$. Combining the above hybrid transitions yields

$$\begin{aligned}&\mathrm{REAL}_{\Pi_{\text{UM-PSO}},\mathcal{A}}(1^\kappa, \mathbf{X}, \mathsf{op})\\ &\approx_c \mathrm{IDEAL}^{\mathcal{L}_D}_{\mathcal{F}_{\mathsf{UM-PSO}},\mathcal{S}_{\mathsf{UM-PSO}}}(1^\kappa, \mathbf{X}, \mathsf{op}).\end{aligned}$$

□

H. Performance of $\Pi_{\mathsf{UM-PSO}}$

We first evaluate the malicious multiparty secret shared shuffle protocol. We assume the length of each input item is 128 bits, and the total input size is $nm \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$ from $n = 3, 4, 5$ parties. The underlying network is the Waksman network [34] with $O(\log^2 M)$ depth and $O(M \log M)$ switches, where M is the number of input items. The running time and communication cost of Π_{mSSS} scale linearly with the input size in the online phase.

The experiment results for Π_{DOPRF} are shown in Table V. We require that the PRF results look random whenever the input item x is chosen arbitrarily. In this experiment, we consider that the input items are concatenated and secret shared, and then fed into the Π_{DOPRF} protocol. Therefore, Π_{DOPRF} processes all nm secret shared items from participating parties, where $nm \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$. We can conclude from the results that the computation cost and communication cost of Π_{DOPRF} scale linearly with the input size.

We measure the entire time and communication cost of the preprocessing phase, which includes a tree-structured mSSS procedure and levels of $\Pi_{\mathsf{DY\text{-}DOPRF}}$ computation, as shown in Table VI. In the tSSS procedure, the input size processed by Π_{mSSS} in each level is reduced by a factor of two. In level i, the input size is 2^{16-i}, and the running time of each level decreases by a factor of two. Meanwhile, the communication cost remains approximately constant, as the item length ℓ doubles while the input size decreases.

TABLE V
Running time (in seconds) and communication cost (in MB) of Π_{mSSS} and $\Pi_{\mathrm{DY\text{-}DOPRF}}$ in the preprocessing phase for different set sizes ($nm \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$) under different network settings and numbers of parties.

Network	n	Subprotocols	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
LAN	3	mSSS	0.07	0.37	0.65	2.45	13.5
		DOPRF	0.08	0.23	0.70	2.44	8.67
	4	mSSS	0.14	0.74	1.31	4.90	26.9
		DOPRF	0.10	0.26	0.74	2.74	10.4
	5	mSSS	0.24	1.23	2.17	8.17	44.8
		DOPRF	0.12	0.31	0.88	3.29	12.8
WAN 400Mbps	3	mSSS	1.49	1.82	7.29	13.43	71.3
		DOPRF	0.261	0.53	1.04	3.01	11.0
	4	mSSS	0.47	2.84	14.58	26.84	143
		DOPRF	0.277	0.584	1.25	3.97	15.2
	5	mSSS	0.78	4.73	24.3	44.7	238
		DOPRF	0.29	0.62	1.54	5.36	20.6
WAN 100Mbps	3	mSSS	3.12	4.65	19.1	57.0	204
		DOPRF	0.264	0.62	1.62	5.65	22.7
	4	mSSS	5.20	7.76	38.8	114	408
		DOPRF	0.318	0.79	2.56	9.55	38.7
	5	mSSS	8.67	12.9	63.8	190	680
		DOPRF	0.43	1.17	3.92	14.7	60.2
Comm.	3	mSSS	3.01	12.0	126	504	2016
		DOPRF	0.78	3.15	12.6	50.3	201
	4	mSSS	6.00	24.0	252	1008	4032
		DOPRF	1.56	6.28	25.1	101	403
	5	mSSS	10.0	40.0	420	1680	6720
		DOPRF	2.60	10.5	41.9	168	671

Online Set Operations. In this phase, the main operation that needs to be implemented is secure comparison. The evaluation results of the online computation phase are shown in Table VI. We evaluate the performance of the secret shared comparison protocol Π_{Comp}, which is also the common online set operation cost. We implement the protocol based on the MP-SPDZ framework, and it supports secret shared comparison under the malicious majority setting [43]. When the number of parties scales from 3 to 5, the running time and communication cost increase by around $2\times$. The thread used in this phase is set to 1 by default. This running time excludes the reveal phase.

I. One-time Input-independent Setup

In the input-independent setup phase, parties will generate Correlated Randomness (CR) for later use in one-time, and we provide a separate evaluation of this phase for completeness and transparency. In practice, as PCG-based generation typically produces more CR than is actually consumed, its exact cost is difficult to measure accurately [44, 45]. Therefore, we report an estimate of the running time and communication cost.

TABLE VI
Running time (in seconds) and communication cost (in MB) of the preprocessing and online phases of UM-PSO for different set sizes ($nm \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$) under different network settings and numbers of parties.

Network	n	Phase	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
LAN	3	Pre.	11.3	15.8	27.4	62.6	194
		Online.	0.21	0.30	0.79	2.93	10.9
	4	Pre.	24.1	31.6	52.4	104	284
		Online.	0.23	0.41	0.80	2.97	11.8
	5	Pre.	46.1	57.7	85.9	153	392
		Online.	0.25	0.63	1.06	3.75	12.1
WAN 400Mbps	3	Pre.	13.5	21.9	65.7	226	686
		Online.	1.60	1.79	2.78	5.63	14.9
	4	Pre.	26.8	40.7	127	432	1233
		Online.	1.88	2.11	3.25	6.25	16.1
	5	Pre.	49.5	70	207	699	2032
		Online.	2.06	2.38	3.89	8.46	17.8
WAN 100Mbps	3	Pre.	15.3	31.5	179	749	2298
		Online.	1.76	2.39	3.07	6.84	16.3
	4	Pre.	30.8	60.2	355	1483	4341
		Online.	2.27	2.83	3.37	7.35	21.8
	5	Pre.	56.8	105	591	2453	7411
		Online.	2.55	2.99	3.96	9.01	26.8
Comm.	3	Pre.	37.8	181	1939	8869	27177
		Online.	0.35	1.41	3.44	20.7	89.6
	4	Pre.	75.6	363	3880	17738	52391
		Online.	0.58	2.31	5.65	33.9	146
	5	Pre.	126	605	6467	29564	90539
		Online.	0.84	3.35	8.18	49.1	212

TABLE VII
The running time (seconds) and communication cost (in GB) of CR generation for Π_{mSSS}, $\Pi_{\mathsf{DY\text{-}DOPRF}}$, and Π_{Comp} for different set sizes ($nm \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}\}$) under LAN setting with $n = 3, 4, 5$ parties.

Metric	n	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}
Time	3	156	1155	2231	4076	4h
	4	279	1687	3340	6843	7h
	5	439	2495	3252	10830	12h
Comm.	3	0.39	4.67	20.8	75.9	681
	4	0.77	9.33	41.6	190	1362
	5	1.29	15.6	69.3	316	2270

As shown in Table VII, CR generation is significantly more costly than the main protocol execution, and its communication cost is estimated from the amount of CR consumed by each protocol component. Since UM-PSO achieves substantially lower total running time and communication than the baseline protocol, its CR generation overhead is correspondingly much smaller. In contrast, the CR generation cost of the baseline protocol is difficult to measure accurately, owing to the considerably larger volume of randomness it requires.

J. Optimization

In this section, we will give the details of the optimized UM-PSO protocol $\Pi_{\mathsf{UM-PSO}}$, as shown in Figure 14.

We first use the toy example in Figure 4 as a showcase. After the Level-0 shuffle, parties compute PRF values for the shared items within each shuffled group. Specifically, the shared items in $G_{\pi_0(0,1)}$ and $G_{\pi_0(0,2)}$ are evaluated under the shared key $\langle mk_{1,1}\rangle$, i.e., $G_{(1,1)}[1].\mathrm{PRF} = \mathrm{F}_{\mathrm{DY}}(mk_{1,1}, G_{\pi_0(0,1)})$, and $G_{(1,1)}[2].\mathrm{PRF} = \mathrm{F}_{\mathrm{DY}}(mk_{1,1}, G_{\pi_0(0,2)})$. while the items in $G_{\pi_0(0,3)}$ and $G_{\pi_0(0,4)}$ are evaluated under a shared key $\langle mk_{1,2}\rangle$, i.e., $G_{(1,2)}[1].\mathrm{PRF} = \mathrm{F}_{\mathrm{DY}}(mk_{1,2}, G_{\pi_0(0,3)})$, $G_{(1,2)}[2].\mathrm{PRF} = \mathrm{F}_{\mathrm{DY}}(mk_{1,2}, G_{\pi_0(0,4)})$. In the next level shuffling process, the shared keys will also be shuffled with the groups, and we assume $G_{(1,2)}$ are shuffled to $G_{\pi_1(1,1)}$. Rather than computing the PRF on the shared items in $G_{\pi_1(1,1)}$, we directly call $\mathcal{F}_{\mathsf{DOPRF}}.\mathsf{Update}$ with the same shared keys used in the last round on the PRF values, i.e.,

$$\underbrace{G_{(1,2)}[1].\mathrm{PRF}, G_{(1,2)}[2].\mathrm{PRF}}_{mk_{1,1}}, \underbrace{G_{(1,1)}[1].\mathrm{PRF}, G_{(1,1)}[2].\mathrm{PRF}}_{mk_{1,2}}$$

Parties call $\mathcal{F}_{\mathsf{DOPRF}}.\mathsf{Update}$ can reduce one round of communication compared with invoking $\mathcal{F}_{\mathsf{DOPRF}}.\mathsf{Eval}$ to compute a PRF value. With this optimization, parties can reduce one round of communication for computing the PRF values in Step 2.(a) ($h \geq 2$) in Figure 8.

By the pseudorandomness/re-randomization security of the updatable DY-PRF, replacing a fresh Eval in Levels $2 \leq h$ with the Update procedure preserves the distribution required in Hybrid 1. Hence, the optimized construction realizes the same functionality and leakage.

K. Applications

- **Vertical Federated Learning.** It is a privacy-preserving Machine Learning (ML) framework that the training datasets are vertically partitioned [46] and come from multiple parties. Each element in the datasets has an ID and corresponding features. In the preprocessing phase of vertical federated ML, parties need to align the elements in their input datasets. That is, finding elements in multiple datasets that share the same ID but contain different features [47]. Multi-party Private Set Intersection (mPSI) is a good choice for this scenario, allowing parties to securely retrieve all elements with the same ID while keeping unrelated elements private, enabling the execution of subsequent computations.
- **Privacy-preserving Data Valuation.** An interesting application of PSO is within data quality valuation in decentralized learning [48]. In the scenario of multiple parties to collaboratively train machine learning models without exposing their individual datasets, different metrics, e.g., Kullback–Leibler divergence (KLD), Shapley Value (SV), are used to evaluate the quality of the datasets and how the datasets contribute to the trained model [49, 50]. In KLD, the key step involves computing the union of all datasets to estimate the overall data distribution, and then measuring the divergence of a specific dataset's distribution relative to this aggregated distribution. For SV, the marginal contribution of each party's data is assessed—often through comparing predicted labels with ground truth, such as by computing the intersection between the inferred and actual labels. This integration of PSO is essential for advancing

the field of privacy-preserving data valuation, particularly in the context of decentralized learning and profit sharing.

- **Private Heavy Hitter Estimation.** The objective of this task is to identify which strings are "popular" strings shared across multiple parties' datasets, based on an analysis of the most frequent elements [51, 52]. Given multiple input sets, an element is considered a top-k heavy hitter if its frequency ranks among the top k frequencies of all possible elements. We can easily adapt our mPSU protocol to compute the heavy hitters across all input sets. Instead of removing duplicate elements from the parties' inputs to compute the union set, we aggregate all input items and compute the frequency of each element across all parties without revealing the elements.