PULSE: Parallel Private Set Union for Large-Scale Entities

Jiahui Gao
Arizona State University
jgao76@asu.edu

Son Nguyen
Arizona State University
snguye63@asu.edu

Marina Blanton
University at Buffalo
mblanton@buffalo.edu

Ni Trieu
Arizona State University
nitrieu@asu.edu

ABSTRACT

Multi-party private set union (mPSU) allows multiple parties to compute the union of their private input sets without revealing any additional information. Existing efficient mPSU protocols can be categorized into symmetric key encryption (SKE)-based and public key encryption (PKE)-based approaches. However, neither type of mPSU protocol scales efficiently to a large number of parties, as they fail to fully utilize available computational resources, leaving participants idle during various stages of the protocol execution.

This work examines the limitation of existing protocols and proposes a unified framework for designing efficient mPSU protocols. We then introduce an efficient Parallel mPSU for Large-Scale Entities (PULSE) that enables parallel computation, allowing all parties/entities to perform computations without idle time, leading to significant efficiency improvements, particularly as the number of parties increases. Our protocol is based on PKE and secure even when up to $n-1$ semi-honest parties are corrupted. We implemented PULSE and compared it to state-of-the-art mPSU protocols under different settings, showing a speedup of 1.91 to 3.57× for $n = 8$ parties for various set sizes.

1 INTRODUCTION

Private set union (PSU) enables parties to compute the union of their input sets without revealing any information beyond the desired output. In recent years, PSU in the 2-party setting has seen rapid advancements, particularly since Kolesnikov et al. [KRTW19] introduced an efficient PSU framework based on oblivious transfer (OT). This framework has been continuously refined by subsequent works [GMR+21, ZCL+23, JSZ+22, BPSY23, JSZG24, CSSW24, KLS24]. PSU has numerous practical applications, including implementing private-ID functionality [BKM+20], cyber risk assessment and management via joint IP blacklists and joint vulnerability data [HLS+16], private database supporting full join [KRTW19], association rule learning [KC04], joint graph computation [BS05], and aggregation of multi-domain network events [BSMD10]. In this paper, we focus on PSU in a multi-party setting, which facilitates richer data sharing/computing compared to the 2-party scenario. The functionality of multi-party private set union (mPSU) is shown in Figure 1.

To better see the methodology and the differences between PSU in the 2-party and multi-party settings, we first briefly review the 2-party OT-based PSU construction proposed in [KRTW19]. The solution has two phases: First, the receiver learns a bit b representing the membership of each element in the sender's set through reverse membership test (r-PMT). Second, the parties invoke an OT protocol, in which the sender inputs messages $\{\perp, x\}$, where $\perp$ represents a predefined special character, while the receiver inputs b as the choice bit. The receiver learns the sender's element x if it is not in the receiver's set, and $\perp$ otherwise.

To understand how a multi-party protocol evolves from the above PSU structure, two key security properties must be maintained: (i) membership privacy – no party should learn any information about the membership status of any element from other parties' datasets and (ii) element source privacy – for any element in the union, no party is able to determine which party contributed that element.

To achieve the first property, instead of using the reverse membership test (r-PMT), a secret-shared private membership test (SS-PMT) can be used, where the sender and the receiver each learn secret shares of the bit b. To achieve the second property, the parties can shuffle the union before revealing it, breaking the correspondence between elements and participants (Section 2 provides a more detailed discussion of these steps). The state-of-the-art mPSU protocols [LG23, GNT24, DCZB24, LL24, DCZ+25] follow this approach. These works successfully show how an efficient mPSU protocol can be built by leveraging rapid advancements with 2-party PSU protocols. However, they do not fully utilize the resources of multiple parties, resulting in significant idle time as the parties wait for one another.

1.1 Motivation

Building on recent 2-party PSU protocols [KRTW19, GMR+21, ZCL+23, JSZ+22, BPSY23, CSSW24, KLS24], construction of practical multi-party PSU (mPSU) protocols began with publications like [LG23, GNT24, DCZB24, LL24, DCZ+25] that rely on secret-shared private membership tests (SS-PMT), oblivious transfer (OT), and multi-party shuffle protocols. Compared to traditional mPSU protocols that heavily depend on generic multi-party computation (MPC) or homomorphic encryption (HE) techniques, these new approaches are orders of magnitude faster, making real-world deployment of mPSU both practical and efficient.

Existing mPSU works mainly focus on designing efficient protocols when the input set size of each party is large. However, in certain applications, such as IP blacklisting [HLS+16] or submodel federated learning [NWT+20, WU23], the number of parties involved in the mPSU protocol can, on the other hand, be quite large, making scalability a critical concern. For example, in federated learning scenarios, it is common to have more than 100 participants. The number of parties impacts performance mainly because of the round complexity – the state-of-the-art for mPSU has at least linear in the number of parties rounds of communication. This leaves the following open problem:

PARAMETERS: n parties $P_1, ..., P_n$ and the set size m.

FUNCTIONALITY:

- Wait to receive input X_i of size m from P_i.
- Give the union $\bigcup_{i=1}^{n} X_i$ to P_1.

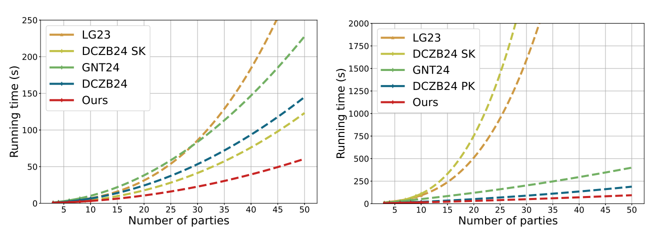
Figure 1: Multi-party Private Set Union Functionality.

*Is it possible to construct an mPSU protocol with $O(1)$ round complexity for the **most time-consuming computation**?*

1.2 Our Contributions

This paper answers the above question affirmatively by proposing a new mPSU protocol that is secure against up to $n-1$ corrupted parties in the semi-honest setting. Our contributions can be summarized as follows:

- We revisit the existing mPSU protocols of [LG23, GNT24, DCZB24, LL24, DCZ+25] in depth. We unify symmetric key encryption (SKE)-based and public key encryption (PKE)-based protocols into a single framework that consists of SS-PMT, message modification, and multi-party shuffle modules.
- We propose an efficient Parallel mPSU for Large-Scale Entities (PULSE) built upon PKE. It supports parallel computation and eliminates idle time for participating parties, making it especially efficient when the number of parties is large and each party's input set is small.
  Our approach introduces simple yet effective modifications to the underlying building blocks. Specifically, we first identify inefficiencies in existing multi-party shuffle protocols caused by sequential execution, which results in $O(n)$ round complexity. Although $O(n)$ complexity may seem unavoidable without incurring significant computational overhead, our new design for oblivious shuffling allows the most time-consuming computations to be performed in parallel, achieving $O(1)$ rounds. Furthermore, we optimize what we call the message modification module of the state-of-the-art mPSU protocols by reducing the round complexity from $O(n)$ to $O(1)$. We also introduce a batched membership oblivious transfer, which serves as a core building block of this module.
- We implement PULSE and compare its performance to state-of-the-art protocols [GNT24, DCZB24]. Our protocol achieves the fastest runtime for most settings, demonstrating up to 1.91–3.57× speedup over these protocols with 3 to 8 parties. When the number of participants is 50, we estimate a 2.39–4.24× performance improvement. As shown in Figure 2, performance improvement increases as the number of parties grows.



(a) LAN runtime. **(b) WAN runtime.**

Figure 2: Performance of mPSU protocols with 2^8-element input sets. Solid lines indicate the times were measured, while dashed lines are estimations using the Levenberg-Marquardt algorithm and the complexity of each protocol. The data for SKE-based protocol originates in [DCZB24].

The rest of the paper is organized as follows: We first give an overview of existing mPSU protocols as well as our techniques in Section 2. In Section 3, we introduce preliminaries for our main result. In Section 4, we present optimizations to the building blocks of the mPSU protocol. Our mPSU protocol is described and analyzed in Section 5, and Section 6 presents performance evaluation.

2 OVERVIEW OF MPSU PROTOCOLS

To better illustrate our improvements, we first review recent state-of-the-art mPSU protocols [LG23, GNT24, DCZB24, LL24, DCZ+25]. For completeness, a discussion of other mPSU protocols is included in Appendix A.

The most recent [DCZ+25] focuses on generic set operations, including private set intersection (PSI). As shown in [DCZ+25, Table 4], however, their protocol is less efficient than [DCZB24] in 95% of the evaluated cases. The protocol proposed in [LL24] adopts similar building blocks and design choices as [GNT24, DCZB24]. While it reduces the communication cost of [GNT24] by approximately 4–5×, it still does not outperform the solution of [DCZB24]. Therefore, we mainly focus on [LG23, GNT24, DCZB24], which represent the current state-of-the-art and/or introduce distinct PSU protocol designs.

2.1 Revisiting Existing Protocols

The designs of the state-of-the-art mPSU protocols have a similar structure. We combine different constructions and protocol variations in a single diagram, shown in Figure 3. This structure consists of three modules, detailed below.

The core idea for computing the union of n sets $X_{j\in[n]}$, each respectively held by party $P_{j\in[n]}$, is given by:

$$X_1 \cup (X_2 \setminus X_1) \cup \ldots \cup (X_n \setminus (X_1 \cup \ldots \cup X_{n-1})) \quad (1)$$

Here, P_1, acting as the leader, collects $X_2 \setminus X_1$ from P_2 to obtain $X_1 \cup X_2$, collects $X_3 \setminus (X_1 \cup X_2)$ from P_3 to obtain $X_1 \cup X_2 \cup X_3$, and this process continues until P_1 collects $X_n \setminus (X_1 \cup \ldots \cup X_{n-1})$ from P_n to obtain $X_1 \cup \ldots \cup X_n$.

From P_j's perspective, for each element $x_{j,k} \in X_j$, P_j must check the element's membership in the set $X_1 \cup \ldots \cup X_{j-1}$. If $x_{j,k} \in X_1 \cup \ldots \cup X_{j-1}$, P_j modifies this element to ensure it does not appear in the final result. This membership check is performed in a pairwise fashion between P_j and each $P_{i<j}$. All protocols from [LG23, GNT24, DCZB24] employ a secret-sharing-based membership test for this purpose, which we abstract as the first module of the mPSU framework and refer to as **Pairwise SS-PMT**. There are two approaches to implementing SS-PMT: [LG23] introduced a *multi-query SS-PMT* protocol, while [DCZB24] proposed a more

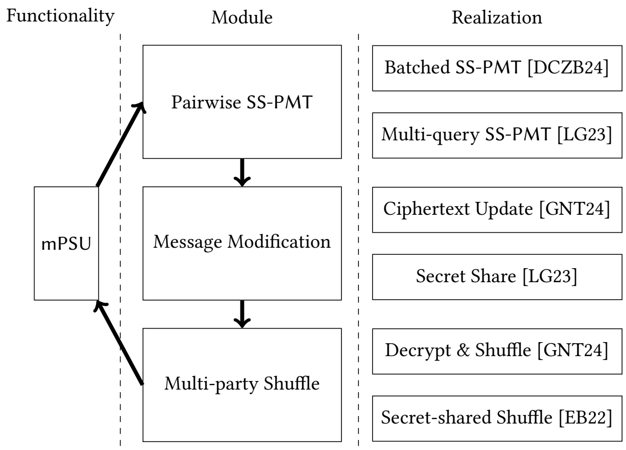


Figure 3: A unified mPSU framework. The arrows show data flow. The first module takes the input and the final module produces the output, representing the overall functionality.

efficient version using *batched* techniques. We discuss both of these variations in Section 3.3.

We refer to the second module as **Message Modification** which produces a correct/fake message for each element, given the shares learned from the Pairwise SS-PMT module. The implementation of this module can vary depending on the underlying encryption technique. That is, protocols may rely on SKE or PKE, with each variant employing different approaches. [LG23] introduced an SKE-based protocol that assumed that the leader does not collude with other parties. It was improved in [DCZB24] in terms of security and efficiency. On the other hand, [GNT24] designed a PKE-based protocol relying on multi-key ElGamal, while [DCZB24] built on it to have a protocol with enhanced optimization and faster implementation. The solutions proceed as follows:

- In an SKE-based protocol, for an element $x_{j,k} \in X_j$, P_j prepares a message $x_{j,k}||H(x_{j,k})$, where $||$ is concatenation and H is a hash function. After executing Pairwise SS-PMT with $P_{i<j}$ and receiving share bits $e^1_{ji,k}$ and $e^0_{ji,k}$ as the output, P_j and P_i proceed with executing random OT [Rab05]. Here, the party P_j acts as the sender with no input, while P_i acts as the receiver with input bit $e^1_{ji,k}$. As a result, P_j obtains two random values $(r^0_{ji,k}, r^1_{ji,k})$ and P_i receives $r^{e^1_{ji,k}}_{ji,k}$. P_j now computes its share as $x_{j,k}||H(x_{j,k}) \oplus \bigoplus_{i=1}^{j-1} r^{e^0_{ji,k}}_{ji,k}$, while P_i sets its share as $r^{e^1_{ji,k}}_{ji,k}$. We observe that these are the shares of the original message $x_{j,k}||H(x_{j,k})$ if $x_{j,k} \notin \bigcup_{i=1}^{j-1} X_i$, and are shares of some random value otherwise. We refer to this approach as "Secret Share" in the "Realization" column of Figure 3.
- In PKE-based approaches, the message for each element $x_{j,k} \in X_j$ is a ciphertext $\mathsf{Enc}(\mathsf{pk}, x_{j,k})$ encrypted using a threshold multi-key encryption, defined as requiring collaboration of more than a threshold number of parties to decrypt. After receiving bit shares from Pairwise SS-PMT, P_j and $P_{i<j}$ perform an OT where P_j acts as the sender with input messages $(\mathsf{Enc}(\mathsf{pk}, \perp), \mathsf{Enc}(\mathsf{pk}, x_{j,k}))$ and $\perp$ being a predefined special element.
  The parties invoke membership OT (mOT) [GNT24] such that if $x_{j,k} \in X_i$, P_i receives the fake message $\mathsf{Enc}(\mathsf{pk}, \perp)$; otherwise, P_i receives $\mathsf{Enc}(\mathsf{pk}, x_{j,k})$. P_i then rerandomizes the ciphertext and sends it back to P_j, who subsequently rerandomizes it again. The re-randomized ciphertext is later used in place of the true message when interacting with subsequent participants.
  As the computation progresses, the message corresponding to $x_{j,k}$ will remain $\mathsf{Enc}(\mathsf{pk}, x_{j,k})$ if $x_{j,k} \notin \bigcup_{i=1}^{j-1} X_i$, and become $\mathsf{Enc}(\mathsf{pk}, \perp)$ otherwise. We refer to this component as "Ciphertext Update" in the "Realization" column of Figure 3.

The final module in the mPSU framework is the **Multi-party Shuffle**, which is designed to protect the element source privacy as mentioned earlier.

SKE-based designs rely on a multi-party *secret-sharing shuffle* protocol [EB22] for this module, where each party holds a share of the vectors along with its own permutation. After this computation, the parties obtain a refreshed share corresponding to the vector permuted n times. The protocol has efficient online computation with round complexity of $O(n)$ and computation complexity of $O(n^2m)$. However, the protocol has poor offline computation complexity of $O(n^3m)$ indicating its unscalability for the scenario of a large number of parties.

In PKE-based designs, an *oblivious shuffle and decryption* protocol [GNT24] is employed, where each party performs partial decryption, permutes the collection of ciphertexts, and then passes it to the next party. This results in a protocol with a round complexity of $O(n)$. Compared to SKE-based approaches, this straightforward method offers a fair total runtime of $O(n^2m)$.

2.2 Our mPSU Protocol

The evaluation in [DCZB24] showed that the SKE-based mPSU does not scale well with a large number of participants due to the cubic complexity of the shuffle protocol. We estimate the performance of the SKE-based protocols using the original data from [DCZB24] and plot them along with the curves of PKE-based protocols in Figure 2. Although the SKE-based protocol may be faster in some cases in the LAN setting, it is significantly slower in the WAN setting due to the underlying shuffle protocol, which has higher complexity. Thus, we focus on further improving existing PKE-based protocols, following the three-module framework presented in Figure 3.

For the Pairwise SS-PMT module, we adopt the batched SS-PMT technique from [DCZB24] to further improve performance. Our main contributions lie in the second and third modules, which account for the majority of the computational cost in the overall mPSU protocol. In existing state-of-the-art PKE-based mPSU protocols, these modules involve a sequence of $O(n)$-round computations, resulting in significant idle time for the parties. In contrast, our protocol enables parallel execution of the most expensive operations, substantially reducing idle time and improving efficiency. Our new message modification module has round complexity of $O(1)$. For the multi-party shuffle module, we consider the shuffle and decryption separately. We enable the parallel computation for the decryption within $O(1)$ round in a straightforward manner. Even though we find inevitable to have a $O(n)$ round complexity for the shuffle, we propose new technique to improve the computation. We next give an overview of these two modules.

A Message Modification Module with $O(1)$ Rounds. As described in the previous subsection, in PKE-based protocols, P_j engages in computation sequentially with each $P_{i<j}$, leading to a round complexity of $O(n)$. In this work, we propose an efficient PKE-based protocol for message modification with $O(1)$ round complexity. The high-level idea is that for each element $x \in X_j$, each P_j interacts with every other party $P_{i<j}$ via mOT in parallel. To accomplish that, for each x, P_j prepares a pair of OT inputs $(\mathsf{Enc}(\mathsf{pk}, 0), \mathsf{Enc}(\mathsf{pk}, r_i))$ for P_i, where r_i is a random number unknown to P_j.[1] As a result of the mOT computation, P_i obtains a ciphertext $e_i = \mathsf{Enc}(\mathsf{pk}, 0)$ if $x \notin X_i$, and $e_i = \mathsf{Enc}(\mathsf{pk}, r_i)$ otherwise. Unlike prior work, where the sender's inputs into the OT protocol depend on the previous computation rounds (leading to inefficiencies due to idle time and requiring computation in the online phase), our protocol allows these OT inputs to be prepared during the offline phase.

Leveraging the additive homomorphism of the EC-ElGamal cryptosystem, P_i re-randomizes the ciphertext e_i and sends it back to P_j. Upon receiving all e_i values, P_j computes the sum of them and adds the sum to the encrypted message $\mathsf{Enc}(\mathsf{pk}, x\|0^\lambda)$). Here, we append extra zero bits for verification after decryption — the length λ is chosen to ensure that the probability of a verification error occurs with negligible probability (i.e., $2^{-\lambda}$). Mathematically, the obtained value is computed as $e = \mathsf{Enc}(\mathsf{pk}, x\|0^\lambda) + \sum_{i=0}^{j-1} e_i$. We can see that if x is not in the union of the previous sets $\bigcup_{i=1}^{j-1} X_i$, all e_i values are the encryptions of 0, implying that e has the form $\mathsf{Enc}(\mathsf{pk}, x\|0^\lambda)$; otherwise, it is an encryption of a random value. After decryption at a later point, we can determine whether the first half is a valid element by checking the last λ bits of the decrypted value and include it in the union accordingly. The details are provided in Section 5.1. Note that mOT executions can be implemented in parallel, which significantly improves the runtime of our protocol.

Additionally, we present batched mOT, which leverages the batched SS-PMT technique from [DCZB24] alongside a simple yet effective optimization of the mOT protocol from [GNT24]. We present the details in Section 4.1.

An Improved Oblivious Shuffle and Decrypt Protocol. In existing protocols, each party sequentially performs partial decryption, re-randomization, and permutation over the collection of ciphertexts. While the permutation step appears to be inherently sequential, we see that the partial decryption and re-randomization processes can be optimized for better efficiency.

We observe that partial decryption, which is more computationally expensive than re-randomization, can naturally support parallel computation by sharing the ciphertext. A more detailed explanation is provided in Sections 3.5 and 4.2. Re-randomizing a ciphertext under the EC-ElGamal cryptosystem is equivalent to adding the original ciphertext to an encryption of 0. Since this addition is inexpensive, efficient computation of $\mathsf{Enc}(\mathsf{pk}, 0)$ directly enables efficient re-randomization. We present our optimization for fast computation of a large number of $\mathsf{Enc}(\mathsf{pk}, 0)$ in Section 5.2.

With these two simple yet effective optimizations, we achieve an efficient oblivious shuffle and decrypt protocol for PKE-based mPSU functionality. Although the overall round complexity remains $O(n)$, all intensive computations can be performed in parallel or offline efficiently.

We compare the round complexity of our mPSU protocol (PULSE) to other PKE-based protocols [GNT24, DCZB24] in Table 1. Our protocol achieves constant round complexity for all modules except the shuffle, which is computationally cheap. This significantly improves the scalability of mPSU, especially when the number of participants is large.

Protocols	Pairwise SS-PMT	Message Modification	Multi-party Shuffle	
			Decryption	Shuffle
[GNT24]	$O(n)$	$O(n)$	$O(n)$	$O(n)$
[DCZB24]	$O(1)$	$O(n)$	$O(n)$	$O(n)$
PULSE	$O(1)$	$O(1)$	$O(1)$	$O(n)$

Table 1: Round Complexity of PKE-based mPSU Protocols.

3 PRELIMINARIES

In this work, we use n to refer to the number of parties and m to the size of each party's input set. We denote the total number of elements as $M = mn$. Computational and statistical security parameters are denoted by κ and λ, respectively. We use $[x]$ to denote the set $\{1, \ldots, x\}$, $[i, j]$ to denote the set $\{i, \ldots, j\}$, and $x||y$ to denote concatenation of two bit-strings x and y.

We use $(\mathsf{sk}, \mathsf{pk})$ to refer to the secret and public keys of a multi-key (threshold) encryption scheme. For simplicity, we occasionally abuse notation by applying a function to a set as if it were applied to each element individually. For example, $\mathsf{Enc}(\mathsf{pk}, X)$ denotes the set of encryptions of each element of the set X.

3.1 Security Model

We use a standard security definition for static semi-honest adversaries as formulated in [Gol09, Lin16]. For an mPSU protocol specifically, we follow the definition presented in [LG23], which is a multi-party variant in the presence of an adversary who is able to corrupt any subset of the participants.

Definition 1. Let $f : (\{0,1\}^*)^n \rightarrow (\{0,1\}^*)^n$ be an n-ary deterministic functionality where $f_i(x_1, \ldots, x_n)$ denotes the i-th element of $f(x_1, \ldots, x_n)$. For a subset $I \subset [n]$, let $f_I = \{f_i\}_{i \in I}$, and $X_I = \{X_i\}_{i \in I}$. Let view_i^π denote the view of party P_i during the execution of protocol π, and view_I^π denote the union of views $\{\mathsf{view}_i^\pi\}_{i \in I}$. We say that π securely computes f in the presence of semi-honest adversaries if for every $I \subset [n]$ there exists a probabilistic polynomial-time (PPT) algorithm S_i such that

$$\{(\mathcal{S}_I(X_I, f_I(X_1, \ldots, X_n)))\} \equiv \{(\mathsf{view}_I^\pi(X_1, \ldots, X_n))\} \quad (2)$$

where $\equiv$ denotes computational or statistical indistinguishability. Unlike the solution from [LG23], our protocol is secure for any

[1]Instead of sampling r and then encrypting it, we directly sample from the ciphertext space, which is more efficient and ensures that P_j does not know the plaintext r_i. The encryption scheme is the multi-key EC-ElGamal cryptosystem.

corruption threshold in the presence of semi-honest participants (i.e., without requiring an honest majority).

3.2 Hashing Scheme

Our solution relies on widely used simple and Cuckoo hashing schemes introduced in [PSSZ15, PSZ18]. We provide a brief review of these schemes below.

Simple hashing. For a hashing table with μ bins denoted as $B[1 \ldots \mu]$, define a random hash function $H : \{0,1\}^\star \rightarrow [\mu]$. To insert an element x into this table, simply place x in the bin of the index determined by the evaluation of $H(x)$. When multiple hash functions $H_1, \ldots, H_h$ are used, x is placed in multiple bins determined by the evaluations of the hash functions.

Cuckoo hashing. This time, there are also μ bins denoted as $B[1 \ldots \mu]$ and h random hash functions $H_1, \ldots, H_h : \{0,1\}^\star \rightarrow [\mu]$. The difference is that for Cuckoo hashing at most one element is allowed to be in a bin. To insert element x, first evaluate the hash functions $H_1(x), \ldots, H_h(x)$ to determine the candidate bins. If any bin $B_{H_i(x)}$ is empty for some $i \in [h]$, place x in that bin. If not, evict an element from one of the candidate bins, place x there, and insert the evicted element again. Based on the analysis in [PSSZ15, DRRT18], given the set size $|X|$, it is possible to set the parameters μ and h in such a way that with an overwhelming probability of $1 - 2^{-\lambda}$ there is an allocation with every bin containing at most one item.

3.3 Secret-Shared Private Membership Test

Secret-shared private membership test (SS-PMT) is widely used in applications beyond mPSU [PSTY19, LPR+21, CDG+21, PSWW18]. It is a two-party protocol where P_0, holding a set $X = \{x_1, \ldots, x_m\}$, interacts with P_1, who has a single input item y. An SS-PMT protocol provides both parties with a secret share of the membership bit. Specifically, the parties receive XOR shares of 1 if $y \in X$, and 0 otherwise.

To complete our review of mPSU techniques, here we briefly describe recent efficient SS-PMT solutions. [LG23] proposed a multi-query SS-PMT based on a multi-query reverse membership test (r-PMT) construction from [ZCL+23]. In r-PMT, instead of both parties learning secret shares of the indicator bit, P_0 learns whether P_1's query is in P_0's set. The first step is to use an oblivious key-value store (OKVS) [GPR+21] so that P_1 with query y will learn an encryption of a value s'. If $y \in X$, s' is equal to the secret value s chosen by P_0. Generic secure multiparty computation (namely, the Goldreich-Micali-Wigderson (GMW) protocol [GMW87]) is used to check this equality and P_0 learns the indicator bit by having P_1 disclose its share to P_0. [LG23] notice that SS-PMT can be easily realized if the sharing step at the end is omitted.

[DCZB24] proposed a batched version of SS-PMT using hashing. Given a set of hash functions $\{H_1 \ldots, H_h\}$, P_0 hashes the input set X into a simple hashing table, while P_1 hashes the query set Y into a Cuckoo hashing table. For the ith bin of the simple hashing table (denoted as B_i), P_0 chooses a random secret value s_i and computes a set S_i such that $|S_i| = |B_i|$. P_0 encodes an OKVS using keys of $B_1, \ldots, B_\mu$ with values $S_1, \ldots, S_\mu$ and sends it to P_1. P_1 decodes it with the element in the Cuckoo hashing table and learns value t_i for the ith bin. For each bin of the hashing table, P_0 and P_1 invoke a generic 2-PC protocol to test equality of s_i and t_i and learn secret shares of 1 if $s_i = t_i$ and 0 otherwise. The functionality is given in Figure 4. The authors provide a comparison of their batch SS-PMT with a multi-query SS-PMT from [LG23]. Despite the large size of the OKVS table in the batch solution, the use of GMW for decryption and comparison in the multi-query SS-PMT introduces a larger computational and communication cost. Thus, we use the batch SS-PMT in our mPSU protocol.

PARAMETERS: Two parties P_0 and P_1, message length ℓ, and batch size μ.

FUNCTIONALITY:
- Wait to receive input sets $\{X_1, \ldots, X_\mu\} \in ((\{0,1\}^\ell)^\star)^\mu$ from P_0.
- Wait to receive input queries $\{y_1, \ldots, y_\mu\} \in (\{0,1\}^\ell)^\mu$ from P_1.
- Give $\{b_{i,j}\}$ to $P_{i \in \{0,1\}}$, where $b_{0,j} \oplus b_{1,j} = 1$ if $y_j \in X_j$ and 0 otherwise for $j \in [\mu]$.

Figure 4: Batched Secret-Shared Private Membership Test (batch SS-PMT) Functionality.

3.4 Membership Oblivious Transfer (mOT)

Gao et al. [GNT24] introduced a new two-party protocol called Membership Oblivious Transfer (mOT) as part of their mPSU protocol. The idea is to enable the receiver to obtain the sender's OT messages based on the result of a membership test. Concretely, the sender holds a keyword $y \in \{0,1\}^\ell$ and two associated messages m_0, m_1. The receiver holds a set $X = \{x_1, x_2, \ldots, x_n\} \subset (\{0,1\}^\ell)^*$. The mOT functionality provides the receiver with a message m_b, where $b = 0$ if $y \in X$ and $b = 1$ otherwise, while the sender learns nothing. Neither party gains any information about the membership of y in X. The sender learns nothing about which message was sent to the receiver, and the receiver learns nothing about the message that was not received. A batched variant of the functionality is given in Figure 5.

PARAMETERS: Sender $\mathcal{S}$ and receiver $\mathcal{R}$, message length ℓ, and batch size μ.

FUNCTIONALITY:
- Wait to receive messages $\{(m_{1,0}, m_{1,1}), \ldots, (m_{\mu,0}, m_{\mu,1})\} \subset ((\{0,1\}^\ell)^2)^\mu$ and queries $\{y_1, \ldots, y_\mu\} \subset (\{0,1\}^\ell)^\mu$ from $\mathcal{S}$.
- Wait to receive input $\{X_1, ..., X_\mu\} \subset ((\{0,1\}^\ell)^\star)^\mu$ from $\mathcal{R}$.
- Give $\mathcal{R}$ messages $\{m_1, \ldots, m_\mu\}$ where m_i equals to $m_{m,0}$ if $y_i \in X_i$, and m_1 otherwise.

Figure 5: Batched Membership Oblivious Transfer (mOT) Ideal Functionality.

3.5 Multi-Key EC-ElGamal Cryptosystem

We review the multi-key cryptosystem from [GNT24] along with its EC-ElGamal construction, which is fundamental to our mPSU protocol. Any realization of such a multi-key cryptosystem can be leveraged to construct our PKE-based mPSU protocol. We adopt elliptic curves due to their simplicity in both theoretical analysis and implementation.

The message space is assumed to be restricted to the point on the elliptic curve for now, which is the common setting as previous PKE-based mPSU protocols [GNT24, DCZB24]. We follow this setting for a fair comparison. When it comes to the practical usage of the mPSU protocols, it's not necessary to have this restriction. A detailed discussion about how supporting messages from arbitrary domains impacts the protocol is provided in Appendix B due to the page limit.

A **multi-key cryptosystem** [GNT24] is defined as a tuple of PPT algorithms (KeyGen, Enc, ParDec, FulDec, ReRand) specified as follows:

- **Key Generation**: $(\mathsf{pk}, \mathsf{sk}_1, \ldots, \mathsf{sk}_n) \leftarrow \mathsf{KeyGen}(1^\kappa, n)$. The key generation algorithm takes as input a security parameter κ and the number of parties n and outputs to each party P_i a secret key sk_i and a joint public key $\mathsf{pk} = \mathsf{Combine}(\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_n)$, where Combine is an algorithm to generate the corresponding public key from a set of secret keys.
  For EC-ElGamal, KeyGen consists of the following steps:
  - **Choose an elliptic curve:** Given a security parameter 1^κ, select an elliptic curve E over a large (as a function of κ) prime field $\mathbb{F}_q$ and a base point G of a large order.
  - **Secret keys:** Generate a secret key $\mathsf{sk} \leftarrow \mathbb{F}_q$ and split it into n additive shares $\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_n$ such that $\mathsf{sk} = \sum_{i=1}^{n} \mathsf{sk}_i$.
  - **Public key:** The public key pk is a point on the curve which is computed as $\mathsf{pk} = \mathsf{Combine}(\mathsf{sk}_1, \ldots, \mathsf{sk}_n)$, where $\mathsf{Combine}(\mathsf{sk}_1, \mathsf{sk}_2, \ldots, \mathsf{sk}_t)$ is defined as computing and outputting $\sum_{i=1}^{t} \mathsf{sk}_i G$, and thus $\mathsf{pk} = \mathsf{sk}G$.
- **Encryption**: $\mathsf{ct} \leftarrow \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$. Given a joint public key pk and a message m from the message space $\mathcal{M}$, the encryption algorithm computes a ciphertext ct.
  For EC-ElGamal, Enc is given by: Randomly select an integer $r \leftarrow \mathbb{F}_q$ and compute the ciphertext as a pair of points $\mathsf{Enc}(\mathsf{pk}, \mathsf{m}) = (\mathsf{ct}_1, \mathsf{ct}_2)$, where $\mathsf{ct}_1 = rG$ and $\mathsf{ct}_2 = \mathsf{m} + r\mathsf{pk}$.
- **Decryption**: There are two types of decryption algorithms:
  - **Partial decryption**: $\mathsf{ct}' \leftarrow \mathsf{ParDec}(\mathsf{sk}_i, \mathsf{ct}, A)$. The partial decryption algorithm takes a secret key sk_i, a ciphertext ct from the ciphertext space C, and a set of indices $A \subseteq [n]$ such that $i \in A$. The ciphertext is interpreted as being encrypted under the partial public key $\mathsf{pk}_A = \mathsf{Combine}(\{\mathsf{sk}_j \mid j \in A\})$ and the algorithm outputs another ciphertext $\mathsf{ct}' \leftarrow C$ encrypting the same message under the partial public key $\mathsf{pk}_{A\setminus\{i\}} = \mathsf{Combine}(\{\mathsf{sk}_j \mid j \in A, j \neq i\})$.
    For EC-ElGamal, to partially decrypt a ciphertext $(\mathsf{ct}_1, \mathsf{ct}_2)$ encrypted under the partial public key $\mathsf{pk}_A = \mathsf{Combine}(\{\mathsf{sk}_j \mid j \in A\}) = \Sigma_{j\in A}\mathsf{sk}_j G$, $\mathsf{ParDec}(\mathsf{ct}_1, \mathsf{ct}_2)$ is given as $(\mathsf{ct}'_1, \mathsf{ct}'_2)$ where $\mathsf{ct}'_1 = \mathsf{ct}_1$ and $\mathsf{ct}'_2 = \mathsf{ct}_2 - \mathsf{sk}_i\mathsf{ct}_1$. Note that the ciphertext $(\mathsf{ct}'_1, \mathsf{ct}'_2)$ can be then re-randomized so that the first part of the ciphertext is different after each partial decryption.
  - **Full decryption**: $\mathsf{m} \leftarrow \mathsf{FulDec}(\mathsf{sk}_1, \mathsf{sk}_2, ..., \mathsf{sk}_n; \mathsf{ct})$. The full decryption algorithm takes a ciphertext $\mathsf{ct} \leftarrow C$ encrypted under pk and all of the secret keys and outputs a message $\mathsf{m} \leftarrow \mathcal{M}$.
    For EC-ElGamal, to fully decrypt a ciphertext $\mathsf{ct} = (\mathsf{ct}_1, \mathsf{ct}_2)$ encrypted under $\mathsf{pk} = \Sigma_{i\in[n]}\mathsf{sk}_i G$, one computes:
    $$\mathsf{m} = \mathsf{ct}_2 - \Sigma_{i\in[n]}\mathsf{sk}_i\mathsf{ct}_1 \tag{3}$$
- **Re-randomization**: $\mathsf{ct}' \leftarrow \mathsf{ReRand}(\mathsf{ct}, \mathsf{pk})$. The re-randomization algorithm takes a ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$ and pk as input and outputs a ciphertext $\mathsf{ct}' \leftarrow C$ such that both ct and ct′ are encryptions of the same message $\mathsf{m} \leftarrow \mathcal{M}$ under pk.
  With EC-ElGamal, to rerandomize a ciphertext $(\mathsf{ct}_1, \mathsf{ct}_2)$ encrypted under the public key pk, one chooses a random value $r' \leftarrow \mathbb{F}_q$ and computes $\mathsf{ct}' = (\mathsf{ct}'_1, \mathsf{ct}'_2)$, where $\mathsf{ct}'_1 = \mathsf{ct}_1 + r'G$ and $\mathsf{ct}'_2 = \mathsf{ct}_2 + r'\mathsf{pk}$.
  Note: In our protocol, re-randomization is usually invoked after a partial decryption, in which case the public key corresponds to a partial key. For simplicity, we write "re-randomization with the corresponding public key" to refer to this situation.

A multi-key cryptosystem should satisfy correctness and security as defined in [Gen09, AJL+12, Bra12]; we refer the reader to these publications for additional information.

Homomorphic Computation. The EC-ElGamal cryptosystem introduced above also supports additive homomorphism, meaning that the addition of two ciphertexts gives a ciphertext encrypting the addition of the two plaintexts.

- **Addition:** Given two ciphertexts $\mathsf{ct}_1 = \mathsf{Enc}(\mathsf{pk}, \mathsf{m}_1) = (\mathsf{ct}_{1,1}, \mathsf{ct}_{1,2})$ and $\mathsf{ct}_2 = \mathsf{Enc}(\mathsf{pk}, \mathsf{m}_2) = (\mathsf{ct}_{2,1}, \mathsf{ct}_{2,2})$ that encrypt plaintexts m_1 and m_2, addition $\mathsf{Enc}(\mathsf{pk}, \mathsf{m}_1) + \mathsf{Enc}(\mathsf{pk}, \mathsf{m}_2) = \mathsf{Enc}(\mathsf{pk}, \mathsf{m}_1 + \mathsf{m}_2)$ is realized by adding the corresponding parts of the ciphertexts:
  $$\mathsf{ct}_{sum} = (\mathsf{ct}_{1,1} + \mathsf{ct}_{2,1}, \mathsf{ct}_{1,2} + \mathsf{ct}_{2,2}) \tag{4}$$
- **Scalar multiplication:** Given a ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}, \mathsf{m}) = (\mathsf{ct}_1, \mathsf{ct}_2)$ and a scalar α, scalar multiplication $\alpha\mathsf{Enc}(\mathsf{m}) = \mathsf{Enc}(\alpha\mathsf{m})$ is realized by multiplying each component of the ciphertext by α:
  $$\alpha\mathsf{ct} = (\alpha\mathsf{ct}_1, \alpha\mathsf{ct}_2) = (\alpha rG, \alpha\mathsf{m} + \alpha r\mathsf{pk}) \tag{5}$$
  This is an encryption of $\alpha\mathsf{m}$ under public key pk.

3.6 Oblivious Shuffle and Decryption

Gao et al. [GNT24] formalized a multi-party protocol known as oblivious shuffle and decryption (Shuffle&Decrypt), which operates under the multi-key cryptosystem introduced in Section 3.5. In this protocol, each party holds a share of the secret key sk_i and prepares a permutation function $\pi_i : [M] \rightarrow [M]$. Given a set of ciphertexts $\{\mathsf{ct}_1, \ldots, \mathsf{ct}_M\}$ encrypted by the corresponding public key pk, the parties aim to compute a shuffled version of the decrypted values.

PARAMETERS: n parties $P_1, \ldots, P_n$, parameter M, and a multi-key encryption scheme defined in Section 3.5

FUNCTIONALITY:

- Wait for input secret key sk_i and a permutation function $\pi_i : [M] \to [M]$ from each party $P_{i\in[n]}$. Here, $(\mathsf{pk}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{KeyGen}(1^\kappa, n)$.
- Wait for a set of ciphertexts $\{\mathsf{ct}_1, \ldots, \mathsf{ct}_M\}$, where $\mathsf{ct}_i = \mathsf{Enc}(\mathsf{pk}, x_i)$ from all parties $\{P_1, \ldots, P_n\}$.
- Give $\{x_{\pi(1)}, \ldots, x_{\pi(M)}\}$ to P_1 where $\pi = \pi_n \circ \pi_{n-1} \circ \ldots \circ \pi_1$.

Figure 6: Oblivious Shuffle and Decryption (Shuffle&Decrypt) Ideal Functionality [GNT24].

The details of the Shuffle&Decrypt functionality are shown in Figure 6.

4 BUILDING BLOCKS

This section presents our optimizations for the two most expensive and critical building blocks of our mPSU protocol.

4.1 Batched Membership Oblivious Transfer

In this section, we present a new batched membership OT protocol (mOT), which builds upon an optimized variant of the single-instance mOT from [GNT24], combined with the use of batched SS-PMT from [DCZB24].

The mOT Protocol of [GNT24]. Before presenting our optimization, we briefly describe the protocol from [GNT24], which is built using SS-PMT and standard OT. Initially, both parties invoke the SS-PMT protocol to obtain shares of a bit, denoted as $b_\mathcal{S}$ for the sender and $b_\mathcal{R}$ for the receiver. Following this, mOT is executed using these shares to transmit one of the messages (m_0, m_1).

In the mOT construction of [GNT24], the sender randomly selects a value $r \leftarrow \{0,1\}^\ell$ and masks the messages as $(r\oplus m_0, r\oplus m_1)$, which are then used as the input to OT. The receiver uses $b_\mathcal{R}$ as the input to OT, thereby obliviously obtaining $w = r \oplus m_{b_\mathcal{R}}$. Subsequently, the sender sends $u = r \oplus (b_\mathcal{S} \cdot (m_0 \oplus m_1))$ to the receiver, who then computes the final output of mOT as $u \oplus w$.

Our Improvement. The construction described above is straightforward, but we observed that it can be further optimized in terms of OT usage. Instead of using $b_\mathcal{R}$ as the choice bit in the OT and preparing the OT messages as $(r\oplus m_0, r\oplus m_1)$, the sender can adjust the order of the OT messages based on the value of $b_\mathcal{S}$. That is, the sender prepares the OT messages (m'_0, m'_1) as either (m_0, m_1) or (m_1, m_0) depending on $b_\mathcal{S}$. Specifically, if $b_\mathcal{S} = 0$, the pair (m'_0, m'_1) is equal to (m_0, m_1); otherwise, it is equal to (m_1, m_0). Therefore, when using $b_\mathcal{R}$ as the OT choice bit, the receiver obtains $m_{b_\mathcal{R}\oplus b_\mathcal{S}}$ as desired, and correctness of this approach is straightforward to verify. This optimization removes the need to send u as in the original mOT protocol, thereby reducing the communication cost.

Our Batched Membership Oblivious Transfer (mOT). To improve performance when the sender has a large number of input queries, we define a batch variant of the mOT functionality in Figure 5 and present its construction in Figure 7. A batch version of mOT can be realized by combining batch SS-PMT with any OT extension. For a batch size of μ, the sender has μ queries $\{y_1, \ldots, y_\mu\}$ and μ pairs of values $\{(m_{1,0}, m_{1,0}), \ldots, (m_{\mu,0}, m_{\mu,0})\}$. The receiver has μ sets. The receiver learns μ values $\{m_1, \ldots, m_\mu\}$, where $m_i = m_{i,0}$ if $y_i \in X_i$ and $m_i = m_{i,1}$ otherwise.

PARAMETERS:

- Sender $\mathcal{S}$ and Receiver $\mathcal{R}$, message length ℓ, and batch size μ.
- The OT and SS-PMT functionalities described in Appendix E and Section 3.3, respectively.

INPUT:

- Receiver $\mathcal{R}$: $\{X_1, \ldots, X_\mu\} \subset ((\{0,1\}^\ell)^*)^\mu$
- Sender $\mathcal{S}$: $Y = \{y_1, \ldots, y_\mu\} \subset (\{0,1\}^\ell)^\mu$ and a set of message pairs $\{(m_{0,0}, m_{0,1}), \ldots, (m_{\mu,0}, m_{\mu,1})\} \subset ((\{0,1\}^\ell)^2)^\mu$

PROTOCOL:

(1) $\mathcal{S}$ and $\mathcal{R}$ invoke batched SS-PMT, where:
  - $\mathcal{R}$ has input sets $\{X_1, \ldots, X_\mu\}$ and $\mathcal{S}$ has input queries $\{y_1, \ldots, y_\mu\}$.
  - $\mathcal{S}$ obtains bits $\{b_{\mathcal{S},0}, \ldots, b_{\mathcal{S},\mu}\}$ and $\mathcal{R}$ obtains bits $\{b_{\mathcal{R},0}, \ldots, b_{\mathcal{R},\mu}\}$, such that $b_{\mathcal{S},j} \oplus b_{\mathcal{R},j} = 1$ if $y_i \in X_i$ and 0 otherwise for $i \in [\mu]$.

(2) For each $i \in [\mu]$, $\mathcal{S}$ and $\mathcal{R}$ invoke an OT instance, where:
  - $\mathcal{S}$ acts as an OT sender with input messages $(m_{i,0}, m_{i,1})$ if $b_{\mathcal{S},i} = 0$ and $(m_{i,1}, m_{i,0})$ if $b_{\mathcal{S},j} = 1$.
  - $\mathcal{R}$ acts as an OT receiver with choice bit $b_{\mathcal{R},i}$ and obtains m_i.

(3) $\mathcal{R}$ outputs $\{m_1, \ldots, m_\mu\}$.

Figure 7: Our Batched Membership Oblivious Transfer (mOT) Construction.

Correctness and Security. Correctness of the protocol is straightforward to verify. Its security relies on the underlying SS-PMT and OT protocols. Since the output of SS-PMT is secret-shared using randomly generated shares, it reveals no information about the set membership. The OT protocol ensures that the receiver learns only the correct message without learning any additional information. Therefore, we omit a formal security proof of Theorem 2 below.

THEOREM 2. *The batched mOT protocol described in Figure 7 securely implements its functionality defined in Figure 5 in the semi-honest setting.*

4.2 An Efficient Oblivious Shuffle and Decryption (Shuffle&Decrypt)

The experimental results from [DCZB24] highlight the impact of efficient underlying elliptic curve (EC) implementations on the performance of PKE-based mPSU protocols. In scenarios closer to real-world applications such as WANs, PKE-based protocols demonstrate significantly better end-to-end performance compared to SKE-based protocols. A central component of the mPSU protocol is an oblivious shuffle and decryption protocol, which heavily relies on PKE operations.

In this section, we first review Shuffle&Decrypt protocols used in the state-of-the-art PKE-based mPSU protocols [GNT24, DCZB24]. We then present an optimization to enable parallel execution of the most time-consuming computations.

Existing Shuffle&Decrypt Protocols. Existing PKE-based mPSU protocols use the Shuffle&Decrypt construction from [GNT24]. The process is straightforward: each party P_i partially decrypts using its secret key sk_i a set of ciphertexts it receives, re-randomizes and shuffles the resulting ciphertexts, and then sends them to the next party.

The protocol clearly takes n rounds. The most time-consuming operation is scalar multiplication on the elliptic curve. As described in Section 3.5, each partial decryption requires one scalar multiplication, and each re-randomization requires two. Therefore, in the mPSU setting, with n parties and inputs sets of size m, each party performs $3mn$ scalar multiplications sequentially, resulting in the total time complexity of $O(mn^2)$. This approach is inefficient, especially for mPSU protocols with a large number of participants. The details of the Shuffle&Decrypt protocol of [GNT24] can be found in Appendix F.

Our Shuffle&Decrypt Protocol. To address the inefficiency of existing Shuffle&Decrypt protocols, we propose a new Shuffle&Decrypt solution that separates the shuffling and decryption phases. This separation enables parallel execution of the decryption phase, in contrast to prior approaches that perform partial decryption sequentially.

Our protocol consists of three phases. The first is an offline phase that prepares a set of encryptions of 0, which are used in the second, re-randomization, phase. In the second phase, re-randomization is efficiently performed by adding a ciphertext ct to an encryption of 0, i.e., $\mathsf{ReRand}(\mathsf{ct}) = \mathsf{ct} + \mathsf{Enc}(0, \mathsf{pk})$. We present an efficient method for computing encryptions of 0 given a public key in Section 5.2.

The final phase is decryption. In the EC-ElGamal cryptosystem used in our protocol, each ciphertext $\mathsf{ct} = (\mathsf{ct}_1, \mathsf{ct}_2)$ encrypting a plaintext m has the following form:

$$\mathsf{ct}_1 = rG \qquad \mathsf{ct}_2 = \mathsf{m} + r\mathsf{pk}$$

In existing protocols [GNT24, DCZB24], during shuffling and partial decryption, each party P_i performs the computation specified below and forwards the result to the next party:

$$\mathsf{ct}_1' = \mathsf{ct}_1 + r'G \qquad \mathsf{ct}_2' = \mathsf{ct}_2 - \mathsf{sk}_i\mathsf{ct}_1 + r'(\mathsf{pk} - \mathsf{sk}_i)$$

where r' is a new random value used for re-randomization.

However, in our "Decrypt" phase, we only need to perform decryption operations, thus, do not require the additional (highlighted) terms associated with the value r'. Concretely, during partial decryption—where each party P_i removes the contribution of their secret key share sk_i—the computation relies only on ct_2, and is performed as $\mathsf{ct}_2' = \mathsf{ct}_2 - \mathsf{sk}_i\mathsf{ct}_1$. Clearly, the full decryption can be executed in parallel where P_1 is the final recipient of the plaintext from a ciphertext $\mathsf{ct} = (\mathsf{ct}_1, \mathsf{ct}_2)$. Concretely,

- P_1 broadcasts ct_1 to all other parties $P_{i\in[2,n]}$.
- Each P_i computes $\mathsf{sk}_i\mathsf{ct}_1$ in parallel and sends the result back to P_1.
- Finally, P_1 computes the message m using the formula:

$$\mathsf{m} = \mathsf{ct}_2 - \sum_{i\in[n]} \mathsf{sk}_i\mathsf{ct}_1$$

PARAMETERS: n parties $P_1, \ldots, P_n$, the set size M, the element length ℓ, EC-ElGamal cryptosystem introduced in Section 3.5.

INPUT:
- Each party $P_{i\in[n]}$: The secret key sk_i and a permutation function $\pi_i : [M] \to [M]$. Here, $(\mathsf{pk}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{KeyGen}(1^\kappa, n)$.
- All parties: $C_0 = \{\mathsf{ct}_1^0, \ldots, \mathsf{ct}_M^0\}$ where $\mathsf{ct}_i^0 = \mathsf{Enc}(\mathsf{pk}, x_i)$.

PROTOCOL:

Phase 0: Pre-processing

(1) Party P_i generates M ciphertexts of zero as $\theta^i_{j\in[M]} = \mathsf{Enc}(\mathsf{pk}, 0) = (r_jG, r_j\mathsf{pk})$ for some random r_j.

Phase 1: Shuffle and Re-randomize

For $i = 1$ to n:

(1) P_i re-randomizes and shuffles the ciphertexts C_{i-1}, and send $C_i = \{\mathsf{ct}_1^i, \ldots, \mathsf{ct}_M^i\}$ to $P_{i\%n+1}$, where $\mathsf{ct}_j^i = \mathsf{ct}^{i-1}_{\pi_i(j)} + \theta^i_j$.

Phase 2: Decrypt

(1) Party P_1 sends the first part of ciphertexts as $C_{n,1} = \{\mathsf{ct}^n_{j,1} \mid \mathsf{ct}^n_j = (\mathsf{ct}^n_{j,1}, \mathsf{ct}^n_{j,2}), \forall j \in [M]\}$ to all parties $P_{i\in[2,n]}$.

(2) Each $P_{i\in[n]}$ in parallel computes $C^i_{par} = \{\mathsf{ct}^i_{par,1}, \ldots, \mathsf{ct}^i_{par,M}\}$ where $\mathsf{ct}^i_{par,j} = \mathsf{sk}_i\mathsf{ct}^n_{j,1}$. Party $P_{i\in[2,n]}$ sends it back to party P_1.

(3) Party P_1, upon receiving C^i_{par} from all $P_{i\in[2,n]}$, computes the final decryption $V = \{v_1, \ldots, v_M\}$ where $v_j = \mathsf{ct}^n_{j,2} - \Sigma^n_{i=1}\mathsf{ct}^i_{par,j}$.

Figure 8: Our Shuffle&Decrypt Protocol.

We provide description of our Shuffle&Decrypt protocol in Figure 8. Security of our Shuffle&Decrypt protocol is stated as follows:

THEOREM 3. *Given the multi-key cryptosystem defined in Section 3.5, the Shuffle&Decrypt protocol described in Figure 8 securely implements the Shuffle&Decrypt functionality defined in Figure 6 in the presence of any semi-honest adversary that corrupts up to $n-1$ parties.*

Since rerandomization is already performed in Phase 1, the decryption phase remains secure. That is, any subset of corrupt parties cannot link a decrypted message to the original plaintext. The complete proof can be found in Appendix C.

Complexity. When invoking mPSU with n parties, the number of ciphertexts in this protocol is $M = mn$. To shuffle and re-randomize ciphertexts in the first phase, each party will receive and send them all to other parties, leading to communication complexity of $O(mn)$ for each party. Given mn ciphertexts, the cost for re-randomization is $O(mn)$ (i.e., two point additions for each ciphertext). The overall time is therefore $O(mn^2)$ with $O(n)$ rounds.

During the second phase, P_1 needs to send ct_1 to and receives $\mathsf{sk}_i\mathsf{ct}_1$ from each $P_{i\in[2,n]}$ for each ciphertext, leading to $O(mn^2)$ communication cost for P_1 and $O(mn)$ for all other parties $P_{i\in[2,n]}$. The computation complexity is $O(mn^2)$ for P_1 who performs n point addition for each ciphertext. The computation complexity, on the other hand, is $O(mn)$ for $P_{i\in[2,n]}$ who performs 1 point multiplication for each ciphertext. The round complexity is $O(1)$.

The most expensive operation when working with ciphertexts is point multiplication. By enabling parallel computation of multiplications, our protocol achieves significant performance improvements when the number of parties is sufficiently large.

5 OUR MPSU PROTOCOL

This section presents our PULSE protocol, which closely follows the overview in Section 2. Our protocol is based on EC-ElGamal cryptosystem and is detailed in Figure 9.

5.1 The Protocol Description

There are n parties $P_1,\ldots,P_n$, and each party P_i has an input set X_i. The union $\bigcup_{i=1}^n X_i$ can be expressed as:

$$X_1 \cup (X_2 \setminus X_1) \cup \ldots \cup (X_n \setminus (X_1 \cup \ldots \cup X_{n-1}))$$

and protocol design closely follows this formula. To compute the union of $X_1,\ldots,X_n$, we start with the set X_1. Then the elements in $X_2 \setminus X_1$ are added to the union. This process continues until all new elements from every input set are included. Thus, the main task for each P_i is to compute $X_i \setminus (X_1 \cup \ldots \cup X_{i-1})$, which traditionally seems to require sequential execution, as shown in previous works. However, in our protocol, we leverage the homomorphic properties of the EC-ElGamal cryptosystem to enable this computation in parallel.

Existing PKE-based mPSU Protocols. In [GNT24, DCZB24], this process involves each party P_i sequentially interacting with $P_1,\ldots,P_{i-1}$ using SS-PMT and OT. This allows the parties to obtain encryptions of the union items (which is the message modification module introduced in Section 2). Specifically, for each element $x_{i,j} \in X_i$, a ciphertext is maintained: if $x_{i,j}$ appears in $X_1\cup\ldots\cup X_{i-1}$, the ciphertext is modified to $\mathsf{Enc}(\mathsf{pk},\perp)$ during the sequential interaction; otherwise, the ciphertext stays as $\mathsf{Enc}(\mathsf{pk},x_{i,j})$. Finally in the multi-party shuffle module, all n parties invoke the Shuffle&Decrypt protocol to decrypt these ciphertexts. The union set is then determined by collecting all values that are not equal to $\perp$.

To understand the sequential nature of their protocol, let us break down the pairwise computation between P_1 and P_j. For each element $x \in X_j$, P_j acts as the mOT sender with query x and messages ($m_0 = \mathsf{Enc}(\mathsf{pk},x), m_1 = \mathsf{Enc}(\mathsf{pk},\perp)$), while P_1 serves as the mOT receiver with input set X_1. If $x \in X_1$, P_1 will obtain a ciphertext e that equals $\mathsf{Enc}(\mathsf{pk},\perp)$; otherwise, $e = \mathsf{Enc}(\mathsf{pk},x)$. P_1 then re-randomizes the ciphertext and sends it back to P_j. P_j retains the value of $\mathsf{ReRand}(e)$ and uses it as the message m_0 when interacting with P_2 later. It is clear that the mOT messages depend on the output from the interaction with the previous party. Consequently, for the last party P_n, the protocol requires $O(n)$ rounds of communication. All parties $P_{i\in[n-1]}$ have to wait for P_n before they can enter the next shuffle stage.

Our PULSE Protocol. In this work, we propose a new mPSU protocol that achieves a constant number of rounds for the message modification phase. The key idea is to replace the mOT messages[2] ($\mathsf{Enc}(\mathsf{pk},x)$, $\mathsf{Enc}(\mathsf{pk},\perp)$) of the party P_j with ($\mathsf{Enc}(\mathsf{pk},0), f$) for each mOT execution between P_j and P_i with $i < j$ and then let P_j modify its own encryption ($\mathsf{Enc}(\mathsf{pk},x)$ at the end of all parallel mOT executions. Here, f represents a ciphertext ($\mathsf{ct}_1,\mathsf{ct}_2$) where both ct_1 and ct_2 are two random points on the elliptic curve. This f is a valid encryption of a random value, and party P_j that samples f does not know the underlying plaintext. We prefer to express f as $\mathsf{Enc}(\mathsf{pk},r)$, where r is a random value chosen anew for each mOT execution and is unknown to P_j. This approach ensures that the two ciphertexts are independent of private inputs, allowing them to be computed during a pre-processing phase. Moreover, computing $\mathsf{Enc}(\mathsf{pk},0)$ can be efficiently performed in an amortized or batched manner as described in Section 5.2, while $\mathsf{Enc}(\mathsf{pk},r)$ is highly efficient and only requires sampling a random point on the elliptic curve.

Now, P_j and P_i first invoke a, mOT protocol, where for each $x \in X_j$, P_j acts as the sender with query x and messages ($m_0 = \mathsf{Enc}(\mathsf{pk},0), m_1 = \mathsf{Enc}(\mathsf{pk},r)$) and P_i acts as the receiver with set X_i. If $x \in X_i$, P_i obtains a ciphertext e_i that equals $\mathsf{Enc}(\mathsf{pk},r)$; otherwise, $e_i = \mathsf{Enc}(\mathsf{pk},0)$. Next, P_i re-randomizes the ciphertext and sends it back to P_j. The re-randomization is designed to prevent P_j from determining which value the OT receiver P_i obtained. Note that the encryption uses a multi-key system, so even if P_j colludes with all parties except P_i, they learn nothing. Finally, the ciphertext e corresponding to x is computed as $e = \mathsf{Enc}(\mathsf{pk},x\|0^\lambda) + \sum_{i=1}^{j-1} e_i$. We use λ extra 0 bits to introduce redundancy and verify whether the decrypted element belongs to any $X_{i<j}$. Specifically, if P_j's item x appears in some set $X_{i<j}$, the corresponding e_i is an encryption of a random element. As a result, $\sum_{i=1}^{j-1} e_i$ becomes an encryption of a random value, which makes the plaintext v of the value e random as well. For each decrypted value v whose last λ bits are 0, we truncate these 0 bits and add the result to the final union. The parameter λ serves as a statistical security parameter, ensuring a negligible error rate of $2^{-\lambda}$.

Note that the last λ bits of the value v are secure to reveal, as the original underlying random message from P_i remains unknown to any party due to the multi-key encryption scheme. Further details on its implementation are provided in Section 6.

Clearly, party P_j can perform all of the above computations in parallel with all other parties $P_{i<j}$. The remainder of the protocol is to decrypt e in a privacy-preserving manner. That is, each party $P_{i\in[2:n]}$ sends its collection of ciphertexts e to P_1. Next, all parties execute our Shuffle&Decrypt protocol proposed in Section 4.2. For each decrypted value v where $v = s\|0^\lambda$, s is added to the final union.

5.1.1 Correctness.

We consider two cases depending on whether a specific element $x \in X_j$ from party P_j is present in any set X_i from party P_i for $1 \le i < j \le n$.

- **Case 1**: There is at least one other set X_i contributed by party P_i that contains the element x. In this case, P_i receives an encryption of a random value $\mathsf{Enc}(\mathsf{pk},r)$ from

[2]We use mOT throughout this discussion, while the formal presentation of our protocol in Figure 9 and its implementation utilize batched mOT.

PARAMETERS:

- n parties $P_{i\in[n]}$ for $n > 1$.
- The batched mOT and Shuffle&Decrypt ideal functionalities, described in Figure 5 and Figure 6, respectively.
- The multi-key cryptosystem (KeyGen, Enc, ParDec, FulDec, ReRand) described in Section 3.5.
- Hashing parameters: a number of bins μ, the h hash functions $H_{j\in[h]} : \{0,1\}^\star \to [\mu]$.

INPUT:

- Party $P_{i\in[n]}$ has $X_i = \{x_{i,1}, \ldots, x_{i,m}\} \subset \{\{0,1\}^\ell\}^m$.

PROTOCOL:

Phase 0: Setup

(1) All n parties call the key generation algorithm $\mathsf{KeyGen}(1^\lambda, 1^\kappa)$. Each P_i receives a private key sk_i and a joint public key pk.

(2) Pre-processing:

(a) P_1 hashes set X_1 into a simple hashing table with μ bins $S_{1,1}, \ldots, S_{1,\mu}$.

(b) $P_{j\in[2,n]}$ hashes set X_j into a cuckoo hashing table with μ bins $C_{j,1}, \ldots, C_{j,\mu}$ and a simple hashing table with μ bins $S_{j,1}, \ldots, S_{j,\mu}$.

(c) $P_{j\in[2,n]}$ computes the encryption $e_{j,k} = \mathsf{Enc}(\mathsf{pk}, C_{j,k}||0^\lambda)$, for non-empty bin $C_{j,k}, k \in [\mu]$. If $C_{j,k}$ is empty, $P_{j\in[2,n]}$ samples e^j_k as random ciphertext and pads it with a random value.

(d) $P_{j\in[2,n]}$ computes a set of $(j-1)\mu$ encryptions of zero as $Z = \{z_{i,k} \mid z_{i,k} = \mathsf{Enc}(\mathsf{pk}, 0)\}_{i\in[j-1],k\in[\mu]}$

(e) $P_{j\in[2,n]}$ samples a set R of $(j-1)\mu$ random ciphertexts which denoted as $F = \{f_{i,k} \mid f_{i,k} \text{ is random}\}_{i\in[j-1],k\in[\mu]}$

(f) $P_{j\in[2,n]}$ initials an empty set E_j.

Phase 1: Pairwise SS-PMT and Message Modification

(3) For each pair of P_i and P_j where $1 \le i < j \le n$:

(a) P_i and P_j invoke a batch mOT protocol where:

- P_i acts as the receiver with inputs $\{S_{i,1}, \ldots, S_{i,\mu}\}$.
- P_j acts as the sender with input queries$\{C_{j,1}, \ldots, C_{j,\mu}\}$ and corresponding messages $\{(z_{i,1}, f_{i,1}), \ldots, (z_{i,\mu}, f_{i,\mu})\}$.
- P_i obtains messages $\{e_{i,j,1}, \ldots, e_{i,j,\mu}\}$.

(b) For $k \in [\mu]$, P_i updates $e_{i,j,k} := \mathsf{ReRand}(\mathsf{pk}, e_{i,j,k})$, and sends $e_{i,j,k}$ back to P_j.

(4) $P_{j\in[2,n]}$ appends $e_{j,k} := e_{j,k} + \Sigma_{i=1}^{j-1} e_{i,j,k}$ to E_j for $k \in [\mu]$.

(5) $P_{j\in[2,n]}$ sends E_j to P_1.

Phase 2: Multi-party Shuffle

(6) All the parties invoke the Shuffle&Decrypt functionality where:

- P_1 inputs $E = \bigcup_{i=2}^n E_i$, the sk_1 and a random permutation $\pi_1 : [M] \to [M]$.
- P_i inputs the private key sk_i and a random permutation $\pi_i : [M] \to [M]$.
- P_1 obtains a set V.

(7) P_1 initials an empty set U. For each $v \in V$, if $v = s||0^\lambda$ holds for some s, P_1 computes $U = U \cup \{s\}$. P_1 outputs $U \cup X_1$.

Figure 9: Our mPSU Protocol (PULSE).

the mOT protocol with P_j in Step (3a). Consequently, the second half of the ciphertext addition in Step (4) will not equal to $\mathsf{Enc}(\mathsf{pk}, 0)$ with high probability. Due to the homomorphism of EC-ElGamal, the decrypted value from Shuffle&Decrypt will not be $x\|0^\lambda$. Thus, x will not be included in the final result.

- **Case 2**: There are no other sets X_i that contain x. In this case, P_i receives an encryption of 0 from the mOT protocol with P_j in Step (3a). Consequently, the second term of the addition in Step (4) (i.e., the sum) will equal to $\mathsf{Enc}(\mathsf{pk}, 0)$. Due to the homomorphism of EC-ElGamal, the decrypted value from Shuffle&Decrypt will stay unchanged as $x\|0^\lambda$. Thus, x will be included in the final result.

Moreover, $\forall x \in X_1$ will be included in the final result. Therefore, the mPSU protocol described in Figure 9 correctly computes the functionality described in Figure 1.

5.1.2 Security. Security of PULSE is stated in the following theorem, showing that it is secure in the presence of any number of semi-honest participants:

THEOREM 4. *Given the multi-key cryptosystem described in Section 3.5, the mPSU protocol described in Figure 9 securely implements the mPSU functionality defined in Figure 1 in the presence of a semi-honest adversary that corrupts up to* $n-1$ *parties in the* $(\mathcal{F}_{Shuffle\&Decrypt}, \mathcal{F}_{mOT})$*-hybrid model.*

Security of PULSE directly follows from the security of mOT and Shuffle&Decrypt. All messages are encrypted using the multi-key cryptosystem introduced in Section 3.5. Note that we use $f =$

$\mathsf{Enc}(\mathsf{pk}, r)$ to denote a random ciphertext rather than the encryption of a random value. Therefore, the value r remains unknown to any adversary unless they corrupt all the parties and can decrypt the ciphertext f. The full proof is given in Appendix D.

5.1.3 Complexity. The computation and communication costs for our mPSU protocol primarily include the following:

Hashing. We select parameters for constructing a simple hash and cuckoo hash in Step (2) using [PSZ18]. Specifically, we use three hash functions and set the number of bins to $1.27m$ for m elements to ensure that cuckoo hashing succeeds—i.e., to find an allocation where every bin contains at most one item—with high probability $(1 - 2^{-40})$.

Batched mOT. The two core building blocks of mOT are SS-PMT and OT, which we discuss separately:

- SS-PMT: We use batched SS-PMT proposed by [DCZB24] that relies on hashing tables which were set up as described above. The SS-PMT sender encodes an OKVS with elements from the simple-hashing table. Using three hash functions leads to $3m$ key-value pairs, and encoding takes $O(m)$ for each instance. The communication cost for sending the OKVS table is also $O(m)$. The SS-PMT receiver decodes the elements in each bin of the cuckoo hashing table, which has complexity $O(m)$. The two parties consequently invoke a generic 2-PC protocol such as GC to perform equality checks for each bin, which requires $O(\lambda)$ AND gates and $O(1)$ rounds.
- OT: We use the IKNP OT extension [IKNP03], which provides computation and communication complexity of $O(m)$ in $O(1)$ rounds.

Shuffle&Decrypt. The complexity analysis of Shuffle&Decrypt was provided in Section 4.2.

5.2 Efficient Computation for Zero Encryptions

In the EC-ElGamal scheme, an encryption of 0 is expressed as $\mathsf{Enc}(\mathsf{pk}, 0) = (rG, r\mathsf{pk})$, where r is a random scalar. To compute $\mathsf{Enc}(\mathsf{pk}, 0)$, we first select r and then perform two scalar multiplications: one that uses the base G and another that uses the public key pk. In our mPSU protocol, each party needs to compute a significant number of encryptions of 0, specifically $(i-1)\mu + n\mu$ of them. The first $(i-1)\mu$ are used as mOT inputs, while the remaining $n\mu$ are consumed by re-randomization in Shuffle&Decrypt. Therefore, we show how to optimize this computation in the batched setting using the Hidden Subset Sum (HSS) technique from [BPV98, NS99]. This technique is designed for generating a large number of (r_i, r_iG) pairs efficiently.

At a high level, the approach involves pre-computing and storing a set of pairs $S = \{(s_i, s_iG)\}_{i \in [n_s]}$, where n_s is relatively small. These values can then be used to generate a large number of pairs, $n \gg n_s$, efficiently.

To generate an additional random pair (r, rG), follow these steps:

- Choose a random subset $R \subseteq [n_s]$ of size s.
- For each $j \in R$, compute $r = \sum_{j \in R} s_j$ and $rG = \sum_{j \in R}(s_jG)$.

The above computation indicates that r is essentially a random subset sum of the s_i values. To generate n tuples of the form (r_i, r_iG) such that r is $2^{-\lambda}$-close to uniformly distributed, we need to determine the parameters n_s and s. Based on the adversary analysis of the random distribution of r in [NS99], we calculate these parameters for realistic values of $\lambda = 40$, various values of n, and a 255-bit cyclic group. The results are presented in Table 2.

n	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}	2^{24}
n_s	2^7	2^8	2^9	2^{10}	2^{13}	2^{14}	2^{15}
s	25	20	17	15	11	11	10

Table 2: Parameters for generating n pseudorandom tuples of the form (r_i, r_iG) given n_s precomputed pairs.

6 IMPLEMENTATION AND PERFORMANCE

We implemented PULSE and evaluated its performance across a varying number of parties and set sizes. All evaluations use a statistical security parameter $\lambda = 40$ and a computational security parameter $\kappa = 128$. Experiments were conducted on a single server with AMD EPYC 74F3 processors and 256 GB of RAM. All parties were run within the same network, but network conditions were simulated using the Linux tc command: a LAN setting with 0.1 ms round-trip latency and 10 Gbps bandwidth; a WAN setting with 80 ms latency and 400 Mbps bandwidth. This is a commonly used setting for evaluate performance of mPSU protocols.

6.1 Performance for Oblivious Shuffle and Decryption Protocols

We implemented our Shuffle&Decrypt protocol from Section 4.2 as well as the protocol used in other PKE-based mPSU works [GNT24, DCZB24] and compare their performance. The results are shown in Table 3 for both network settings and different set sizes. Our protocol is up to 2.20 times faster than the protocol in [GNT24, DCZB24] , while requiring approximately 1.88 times more communication cost. The improvement increases as the number of parties increases.

6.2 Performance for PULSE

We also implemented the entire PULSE protocol. To implement batch SS-PMT, we used OKVS and GMW from [RR22]. We also use the IKNT OT-extension [IKNP03] from `libOTe` [PR] to implement mOT. The EC-ElGamal cryptosystem is implemented using the NIST P-256 curve from OpenSSL. These choices are consistent for all the state-of-the-art mPSU works [GNT24, DCZB24] [3]. Our implementation will be available on GitHub.

As described in Section 5, the parameters are set to limit the probability of error to at most $2^{-\lambda}$. To implement $\mathsf{Enc}(\mathsf{pk}, x||0^{\lambda})$ using EC-ElGamal, we use concatenation of two EC-ElGamal ciphertexts $\mathsf{Enc}(\mathsf{pk}, x)||\mathsf{Enc}(\mathsf{pk}, 0)$. The zero element is the additive identity point on the curve (point at infinity), to which we refer as 0. The NIST P-256 curve provides a much lower than $2^{-\lambda}$ probability of error for verification purposes. Compressed representation of points on this curve results in a ciphertext being represented using 66 bytes. To verify the final output, P_i first decrypts the second

[3] [LL24, DCZ+25] shows that their results do not outperform the numbers reported in [DCZB24].

	m	Prot.	$n = 3$	$n = 4$	$n = 6$	$n = 8$
LAN (s)	2^8	[GNT24]	0.21	0.41	0.99	1.82
		OURS	0.15	0.26	0.53	0.90
	2^{10}	[GNT24]	0.43	0.78	3.93	7.23
		OURS	0.35	0.49	2.02	3.44
	2^{12}	[GNT24]	3.31	6.46	15.77	29.00
		OURS	2.17	3.74	7.88	13.49
	2^{14}	[GNT24]	13.29	26.06	63.69	117.01
		OURS	8.28	14.553	31.45	53.458
	2^{16}	[GNT24]	53.54	103.99	253.61	467.05
		OURS	32.28	57.31	121.80	211.95
WAN (s)	2^8	[GNT24]	0.89	1.37	4.20	7.43
		OURS	1.03	1.50	3.90	6.66
	2^{10}	[GNT24]	2.83	4.34	9.10	14.24
		OURS	2.74	4.51	8.27	11.51
	2^{12}	[GNT24]	5.65	10.01	22.02	37.64
		OURS	5.28	8.73	15.47	24.45
	2^{14}	[GNT24]	16.75	30.86	71.38	127.75
		OURS	12.88	21.20	41.36	67.67
	2^{16}	[GNT24]	58.02	110.55	266.81	490.82
		OURS	38.62	66.29	139.22	239.35
Comm. (MB)	2^8	[GNT24]	0.10	0.19	0.48	0.90
		OURS	0.16	0.34	0.89	1.69
	2^{10}	[GNT24]	0.39	0.77	1.93	3.61
		OURS	0.64	1.35	3.54	6.77
	2^{12}	[GNT24]	1.55	3.09	7.73	14.44
		OURS	2.58	5.41	14.18	27.07
	2^{14}	[GNT24]	6.19	12.38	30.94	57.75
		OURS	10.31	21.66	56.72	108.28
	2^{16}	[GNT24]	24.75	49.50	123.75	231.00
		OURS	41.25	86.63	226.88	433.13

Table 3: Performance for Shuffle&Decrypt protocols. The running time is in seconds and communication cost is in MB. Communication cost is the total cost for all parties. Best performance is highlighted in blue.

ciphertext. If the value is 0, decryption can be performed on the first half to learn x; otherwise, no further computation is needed. During our evaluation, we decrypt both parts to benchmark the performance. Even though this almost doubles the computational cost for all heavy PKE-related operations as well as communication due to the extra bits, our protocol still provides much better performance compared to the previous results.

Comparison with Previous Work. A comprehensive evaluation was provided in [DCZB24] for their PKE-based and SKE-based mPSU protocols. In general, SKE-based mPSU is much more expensive for a large number of parties due to the high communication cost stemming from the shuffle protocol. For that reason we only compare PULSE to PKE-based protocols. Unfortunately, the implementation of [DCZB24] is not publicly available. To have a fair comparison with their results, we estimated their performance based on our implementation since many building blocks are shared. Each party uses one thread for its own computation and uses $n-1$ threads to communicate with other parties in parallel. We test PULSE with different set sizes $m = \{2^8, 2^{12}, 2^{16}\}$ and a variable

	m	Prot.	$n = 3$	$n = 4$	$n = 6$	$n = 8$
LAN (s)	2^8	[GNT24]	1.10	1.88	3.94	6.72
		[DCZB24]	0.67	1.17	2.50	4.27
		Ours	0.65	0.87	1.44	2.23
	2^{12}	[GNT24]	16.49	27.96	59.65	103.86
		[DCZB24]	6.53	11.79	26.69	47.40
		Ours	5.75	9.00	17.82	30.38
	2^{16}	[GNT24]	284.47	490.53	1061.45	1838.59
		[DCZB24]	102.88	187.20	422.56	754.08
		Ours	95.36	154.29	307.10	514.59
WAN (s)	2^8	[GNT24]	9.49	15.07	26.69	38.72
		[DCZB24]	4.13	5.43	10.40	15.70
		Ours	4.39	5.48	8.76	11.27
	2^{12}	[GNT24]	31.61	50.30	97.07	154.20
		[DCZB24]	13.10	20.27	39.20	63.03
		Ours	12.20	17.36	29.06	44.20
	2^{16}	[GNT24]	336.84	568.27	1189.99	2017.89
		[DCZB24]	124.05	215.86	468.76	815.37
		Ours	113.83	175.03	339.97	572.20
Comm. (MB)	2^8	[GNT24]	1.46	2.20	3.68	5.16
		[DCZB24]	2.34	3.51	5.85*	8.19*
		Ours	1.10	1.74	3.23	4.97
	2^{12}	[GNT24]	20.25	30.48	50.96	71.47
		[DCZB24]	8.15	12.82	23.95*	38.07*
		Ours	11.82	19.28	37.28	59.41
	2^{16}	[GNT24]	321.40	483.69	808.79	1134.21
		[DCZB24]	65.41	98.12	163.50*	228.90*
		Ours	184.41	301.37	584.79	934.20

Table 4: Performance for mPSU Protocols. The running time is in seconds and communication cost is in MB. Communication cost is the cost for P_1. Best performances are highlighted in blue. $*$ indicates estimation based on the number reported in[DCZB24].

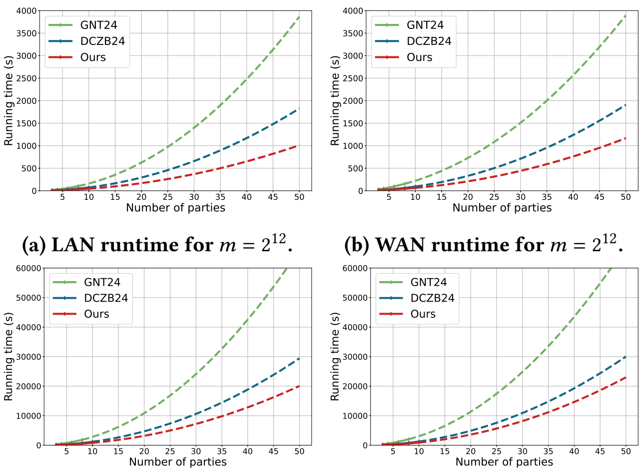
number of parties up to 8. The end-to-end running time and the communication cost are shown in Table 4.

Our protocol has the fastest running time compared to the state-of-the-art PKE-based protocols for most of the settings in both LAN and WAN. For example, for 8 parties each with a set size of 2^8 elements, our protocol is 1.91× faster than [DCZB24] and 3.01× faster than [GNT24] in LAN setting, and is 1.39× and 3.44× faster in WAN correspondingly.

We believe that PULSE has a much better running time for a large number of participants. However, it is difficult to obtain accurate times on a single server. To further demonstrate scalability of our protocol, we use the numbers in Table 4 to estimate the running time with a larger number of parties based on the complexity of each protocol[4]. We do a curve-fitting process using SciPy library for Python. Given the complexity of each protocol, Levenberg-Marquardt algorithm [Lev44] determines the best curve based on the data. The performance estimates for both network settings with a set size of 2^8 were shown earlier in Figure 2. Additionally, Figure 10 presents performance of mPSU protocols for $n \in \{2^{12}, 2^{16}\}$ as the number of parties increases.

[4]We use the $n = \{3, 4, 6, 8\}$ for [GNT24], and $n = \{3, 4, 6, 8, 10\}$ for [DCZB24] and our work.

Table 4 displays the amount of communication as well. Compared to PKE-based protocols, PULSE has $1.04 - 2.13\times$ less communication for a small set size of 2^8, while it has up to $4.08\times$ higher communication cost in some other cases. This is mainly because we pad zeros for the final verification. The result indicates that our protocol is more competitive for small sets.



(a) LAN runtime for $m = 2^{12}$. **(b) WAN runtime for $m = 2^{12}$.**

(c) LAN runtime for $m = 2^{16}$. **(d) WAN runtime for $m = 2^{16}$.**

Figure 10: Performance of mPSU Protocols with $\{2^{12}, 2^{16}\}$-element Input Sets. Solid lines indicate the times were measured, while dashed lines are estimations using the Levenberg-Marquardt algorithm and the complexity of each protocol.

7 CONCLUSION

In this work, we present a detailed study of mPSU protocols. We present a unified framework for mPSU that covers both SKE-based and PKE-based methods. We propose an efficient Parallel mPSU for Large-Scale Entities (PULSE) built upon PKE. It supports parallel computation and eliminates idle time for participating parties – for the first time – making it especially efficient when the number of parties is large and each party's input set is small. Compared to state-of-the-art mPSU protocols, our approach achieves significant improvements in end-to-end runtime, particularly as the number of parties increases.

Future work includes extending our protocol to the malicious setting, further optimizing communication overhead, and improving performance for large set sizes.

ACKNOWLEDGMENTS

This work was supported in part by ARPA-H SP4701-23-C-007, NSF 2451972, and NSF 2213057 grants. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the funding sources.

REFERENCES

[AJL+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. Multiparty computation with low communication, computation and interaction via threshold FHE. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 483–501, April 2012.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 40–41. ACM Press, May 2012.

[BKM+20] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Paper 2020/599, 2020. https://eprint.iacr.org/2020/599.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology — ASIACRYPT 2001*, pages 514–532, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[BPSY23] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Near-Optimal oblivious Key-Value stores for efficient PSI, PSU and Volume-Hiding Multi-Maps. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 301–318, Anaheim, CA, August 2023. USENIX Association.

[BPV98] Victor Boyko, Marcus Peinado, and Ramarathnam Venkatesan. Speeding up discrete log and factoring based schemes via precomputations. In Kaisa Nyberg, editor, *EUROCRYPT'98*, volume 1403 of *LNCS*, pages 221–235, May / June 1998.

[Bra12] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 868–886, August 2012.

[BS05] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 236–252, December 2005.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas Dimitropoulos. SEPIA: Privacy-Preserving aggregation of Multi-Domain network events and statistics. In *19th USENIX Security Symposium (USENIX Security 10)*, Washington, DC, August 2010. USENIX Association.

[CDG+21] Nishanth Chandran, Nishka Dasgupta, Divya Gupta, Sai Lakshmi Bhavana Obbattu, Sruthi Sekar, and Akash Shah. Efficient linear multiparty PSI and extensions to circuit/quorum PSI. pages 1182–1204. ACM Press, 2021.

[CDKS19] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 395–412. ACM Press, November 2019.

[CSSW24] Gowri R Chandran, Thomas Schneider, Maximilian Stillger, and Christian Weinert. Concretely efficient private set union via circuit-based PSI. Cryptology ePrint Archive, Paper 2024/1494, 2024.

[DCZ+25] Minglang Dong, Yu Chen, Cong Zhang, Yujie Bai, and Yang Cao. Multi-party private set operations from predicative zero-sharing. In *ACM Computer and Communications Security (CCS)*, 2025.

[DCZB24] Minglang Dong, Yu Chen, Cong Zhang, and Yujie Bai. Breaking free: Efficient multi-party private set union without non-collusion assumptions. Cryptology ePrint Archive, Paper 2024/1146, 2024. https://eprint.iacr.org/2024/1146.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. Pir-psi: Scaling private contact discovery. Cryptology ePrint Archive, Paper 2018/579, 2018. https://eprint.iacr.org/2018/579.

[EB22] Saba Eskandarian and Dan Boneh. Clarion: Anonymous communication from multiparty shuffling protocols. In *29th Annual Network and Distributed System Security Symposium, NDSS 2022, San Diego, California, USA, April 24-28, 2022.* The Internet Society, 2022.

[Fri07] Keith B. Frikken. Privacy-preserving set union. In Jonathan Katz and Moti Yung, editors, *ACNS 07*, volume 4521 of *LNCS*, pages 237–252, June 2007.

[Gen09] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the Forty-First Annual ACM Symposium on Theory of Computing*, STOC '09, page 169–178, New York, NY, USA, 2009. Association for Computing Machinery.

[GHJ22] Xuhui Gong, Qiang-Sheng Hua, and Hai Jin. Nearly optimal protocols for computing multi-party private set union. In *2022 IEEE/ACM 30th International Symposium on Quality of Service (IWQoS)*, pages 1–10, 2022.

[GMR+21] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In Juan A. Garay, editor, *Public-Key Cryptography – PKC 2021*, pages 591–617, Cham, 2021. Springer International Publishing.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing (STOC)*, pages 218–229. ACM, 1987.

[GNT24] Jiahui Gao, Son Nguyen, and Ni Trieu. Toward a practical multi-party private set union. In *The 24th Privacy Enhancing Technologies Symposium*

(PoPETs), pages 622–635, 2024.
[Gol09] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, USA, 1st edition, 2009.
[GPR+21] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. LNCS, pages 395–425, 2021.
[HLS+16] Kyle Hogan, Noah Luther, Nabil Schear, Emily Shen, David Stott, Sophia Yakoubov, and Arkady Yerukhimovich. Secure multiparty computation for cooperative cyber risk assessment. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 75–76, 2016.
[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161, August 2003.
[JSZ+22] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2947–2964, Boston, MA, August 2022. USENIX Association.
[JSZG24] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, and Dawu Gu. Scalable private set union, with stronger security. Cryptology ePrint Archive, Paper 2024/922, 2024.
[KC04] M. Kantarcioglu and C. Clifton. Privacy-preserving distributed mining of association rules on horizontally partitioned data. *IEEE Transactions on Knowledge and Data Engineering*, 16(9):1026–1037, 2004.
[KLS24] Jiseung Kim, Hyung Tae Lee, and Yongha Son. Revisiting shuffle-based private set unions with reduced communication. Cryptology ePrint Archive, Paper 2024/1560, 2024.
[Kob87] Neal Koblitz. Elliptic curve cryptosystems. *MATHEMATICS OF COMPUTATIONs*, 48:203–209, 1987.
[KRTW19] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 636–666, December 2019.
[KS05] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 241–257, August 2005.
[Lev44] Kenneth Levenberg. A method for the solution of certain non – linear problems in least squares. *Quarterly of Applied Mathematics*, 2:164–168, 1944.
[LG23] Xiang Liu and Ying Gao. Scalable multi-party private set union from multi-query secret-shared private membership test. LNCS, pages 237–271, 2023.
[Lin16] Yehuda Lindell. How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Paper 2016/046, 2016.
[LL24] Qiang Liu and Joon-Woo Lee. Efficient multi-party private set union resistant to maximum collusion attacks. Cryptology ePrint Archive, Paper 2024/2096, 2024.
[LPR+21] Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Karn Seth, and Ni Trieu. Private join and compute from PIR with default. LNCS, pages 605–634, 2021.
[MTBH21] Christian Mouchet, Juan Ramón Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *PoPETs*, 2021(4):291–311, October 2021.
[NS99] Phong Q. Nguyen and Jacques Stern. The hardness of the hidden subset sum problem and its cryptographic implications. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 31–46, August 1999.
[NWT+20] Chaoyue Niu, Fan Wu, Shaojie Tang, Lifeng Hua, Rongfei Jia, Chengfei Lv, Zhihua Wu, and Guihai Chen. Billion-scale federated learning on mobile clients: a submodel design with tunable privacy. In *Proceedings of the 26th Annual International Conference on Mobile Computing and Networking*, MobiCom '20, New York, NY, USA, 2020. Association for Computing Machinery.
[PR] Lance Roy Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.
[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 515–530. USENIX Association, August 2015.
[PSTY19] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 122–153, May 2019.
[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 125–157, April / May 2018.
[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2), jan 2018.
[Rab98] Tal Rabin. A simplified approach to threshold and proactive RSA. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 89–104, August 1998.
[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Paper 2005/187, 2005.
[RR22] Srinivasan Raghuraman and Peter Rindal. Blazing fast PSI from improved OKVS and subfield VOLE. pages 2505–2517. ACM Press, 2022.
[SCK12] Jae Hong Seo, Jung Cheon, and Jonathan Katz. Constant-round multiparty private set union using reversed laurent series. volume 7293, pages 398–412, 05 2012.
[SEA23] Microsoft SEAL (release 4.1). https://github.com/Microsoft/SEAL, January 2023. Microsoft Research, Redmond, WA.
[SM18] Katsunari Shishido and Atsuko Miyaji. Efficient and quasi-accurate multiparty private set union. In *2018 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 309–314, 2018.
[VCE22] Jelle Vos, Mauro Conti, and Zekeriya Erkin. Fast multi-party private set operations in the star topology from secure ands and ors. Cryptology ePrint Archive, Paper 2022/721, 2022. https://eprint.iacr.org/2022/721.
[WU23] Zhusheng Wang and Sennur Ulukus. Private federated submodel learning via private set union, 2023.
[ZCL+23] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. Linear private set union from Multi-Query reverse private membership test. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 337–354, Anaheim, CA, August 2023. USENIX Association.

A RELATED WORK

In this section, we focus on the state-of-the-art of multi-party PSU protocols.

mPSU from Polynomial Representation. Kissner and Song [KS05] proposed the first construction of mPSU, which represents elements of an input set as roots of a polynomial (i.e., $P_{i\in[n]}$ represents its input set $X_i = \{x_{i,1}, \dots, x_{i,m}\}$ as a polynomial $f_i(x) = \prod_{j=1}^{m}(x - x_{i,j})$). The parties compute the global representation of the union as $p = \prod_{i=1}^{n} f_i$ in a secure manner heavily relying on homomorphic encryption. The roots of the polynomial p induce the union of $\bigcup_{i=1}^{n} X_i$. This protocol has computation complexity $O(n^3m^2)$ and uses expensive additive homomorphic encryption (AHE) such as Paillier, which makes it impractical. To improve performance of [KS05]'s solution, Frikken [Fri07] proposed an efficient mPSU protocol, which is also based on polynomial representation. Both protocols tolerate any number of corrupted parties but require $O(m)$ rounds. Similar to the idea of polynomial representation, Seo et al. [SCK12] represented input sets by rational polynomial functions and reversed Laurent series. Their mPSU protocol achieves a constant number of rounds but is secure only up to $n/2$ corrupted parties.

mPSU from Oblivious Sorting. Blanton and Aguiar [BA12] presented a new series of private set operation protocol, including union, that can avoid expensive homomorphic encryption. The idea is to perform an oblivious sorting algorithm over the aggregation of all input sets in a secret-shared manner. The union is given by comparing the adjacent elements to remove duplicates. Their approach relies on generic MPC methods such as secret sharing, resulting in a large number of rounds, which is inefficient in WAN settings. If implemented alternative techniques such as garbled circuit evaluation, the communication cost is high.

mPSU from Bloom Filter. Bloom filter (BF) is a data structure that can efficitenly perform membership queries and is widely used in private set operations' protocol design. [SM18, GHJ22] proposed mPSU protocols based on the BF.

Similar to the idea of polynomial representation, each party locally encodes a BF using their own input set, and a global BF is securely constructed from these local ones. The union can be computed based on the global BF. To reduce the false positive rate introduced by the nature of BF, a table of size $O(\lambda mn)$ is required, which makes the protocol inefficient in terms communication cost.

mPSU for Small Universe. Vos et al. [VCE22] proposed an mPSU protocol for a small universe where the bitlength of elements is at most 32. Each party has a binary vector of length $|\mathcal{U}|$, where the ith bit is set to 1 if the corresponding element belongs to the input set and otherwise it is set to 0. A global vector is computed by invoking a newly proposed private OR protocol. The union can be determined trivially by extracting the elements corresponding to 1 bits in the vector. While the solution uses a divide-and-conquer optimization, its performance is heavily influenced by the size of the universe.

mPSU from SS-PMT and OT. The state-of-the-art mPSU protocols [LG23, GNT24, DCZB24] are based on the technique of secret-sharing private membership test (SS-PMT) with Oblivious Transfer (OT). The idea of using SS-PMT comes from the usage of reverse membership test (r-PMT) with OT in recent 2-party PSU works [GMR+21, ZCL+23, JSZ+22, BPSY23]. In short, each party invokes SS-PMT with other parties to test the membership of its element in other input sets. The shares produced by SS-PMT are used by the parties in OT to send and receive messages that perform modification of input elements. Finally, the parties invoke a multiparty shuffle protocol to erase any patterns that result from message modification in the previous stage. At the end, the union can be constructed by discarding erased (invalid) elements.

Liu and Gao [LG23] proposed a multi-query SS-PMT based on the multi-query r-PMT construction from [ZCL+23] and used it to construct an mPSU protocol. In their mPSU protocol, element x is encoded as $x||H(x)$ where H is a public hash function, and is shared among parties. By invoking random OT with the output bit of SS-PMT, the parties will obtain two different random values if there is a duplication. Adding these random values to the shares of x will violate the $s||H(s)$ format with high probability. Finally, the parties invoke a multi-party secret-shared shuffle protocol of [EB22] to protect the origin of the output values. The leader consequently collects all shares and reconstructs the union. The solution's drawback is that it requires that the leader does not collude with any other party.

While the protocol of [LG23] is primarily based on symmetric key encryption (SKE) techniques, Gao et al. [GNT24] proposed an mPSU protocol using public key encryption (PKE). Instead of using secret sharing, their protocol requires each party to encrypt the input under a public key encryption scheme supporting rerandomization (e.g., ElGamal). Using the share bit from the SS-PMT for each element, the parties invoke OT to rerandomize the ciphertext or substitute it with an encryption of 0 if there is a duplication. Then the parties invoke a Shuffle&Decrypt protocol and learn the union by removing 0s from the decrypted values. This protocol is secure against up to $n-1$ corrupted parties. However, their SS-PMT is not efficient due to many dummy values, and the performance reported is not close to the numbers in [LG23].

Dong et al. [DCZB24] proposed a batched SS-PMT protocol that outperforms the multi-query SS-PMT of [LG23] in the context of mPSU. Based on this new SS-PMT construction, the authors further improve the results of [LG23] and [GNT24] in terms of security and efficiency. In particular, they are able to achieve an efficient implementation of a PKE-based mPSU, which brings our attention to this framework.

Liu and Lee [LL24] follow the PKE-based approach and proposed a new mPSU protocol using SS-PMT and OT. Instead of only allowing P_1 to learn the output, the solution shares the output among all parties. It uses similar building blocks and a similar design to those in [GNT24, DCZB24], with the implementation showing no better numbers that what is reported in [DCZB24]. In addition, recent work of Dong et al. [DCZ+25] focuses on generic set operations, including PSU. As shown in [DCZ+25, Table 4], their protocol is less efficient than [DCZB24] in 95% of the evaluated cases. Thus, we omit a detailed comparisons with these two protocols.

B SUPPORTING MESSAGES FROM ARBITRARY DOMAINS

As mentioned in Section 3.5, it is commonly accepted that the message space of a PKE-based mPSU protocol is restricted to points on an elliptic curve. This is inherited from the choice of ElGamal encryption and its efficient elliptic curve implementation. However, in practice, a more flexible message space is desired to enable general-purpose use for different applications. The most natural form of set elements is binary strings of an arbitrary length. In this section, we discuss how to support this general form of input set elements in our mPSU protocol. We discuss two aspects:

- supporting arbitrary binary strings of a short length.
- extending the construction to handle longer set elements.

B.1 Supporting Arbitrary Strings of Short Length

The problem of supporting set elements represented as arbitrary binary strings can be reduced to finding a suitable mapping from the space of binary strings to the space of points on an elliptic curve.

We adopt a Koblitz-style *try-and-increment* encoding technique [Kob87, BLS01]. Recall that a valid elliptic curve point (x, y) satisfies the equation $y^2 = x^3 + ax + b$, where a and b are predefined curve constants. Given an m-bit message z, we define the following two algorithms:

- **Encoding**: $\mathcal{E} : \{0,1\}^m \to \mathcal{P}$. Set the most significant m bits of x to z and, starting from 0, increment the remaining k least significant bits of x until there exists a value y such that the point (x, y) lies on the specified elliptic curve.
- **Decoding**: $\mathcal{D} : \mathcal{P} \to \{0,1\}^m$. Decode by extracting the most significant m bits of x.

This encoding fails with probability 2^{-k}. On average, finding a suitable x succeeds in two trials.

Next, note that this encoding/decoding scheme does not preserve the additive homomorphism of EC-ElGamal, i.e.,

$$\mathcal{D}(\mathsf{Dec}(\mathsf{Enc}(\mathcal{E}(m_1)) + \mathsf{Enc}(\mathcal{E}(m_2)))) \neq m_1 + m_2.$$

Fortunately, our protocol does not rely on the message-space homomorphism. In particular, the ciphertext addition is only used for the following two purposes in our protocol:

(1) **Re-randomization**: $\mathsf{Enc}(z)+\mathsf{Enc}(0) = \mathsf{Enc}(z)$, so the message remains unchanged.
(2) **Message Modification**: in step (4) of Phase 1, if all $e_{i,j,k}$ are $\mathsf{Enc}(0)$, correctness holds as above. If any $e_{i,j,k}$ is an encryption of random value, the sum $e_{j,k}$ remains unrecoverable.

In both cases, the correctness of our mPSU protocol holds.

For curve P-256, we set $k = 40$ and also reserve 40 bits for trailing 0s as we need to compute $\mathsf{Enc}(z||0^{40})$ in our mPSU protocol. This leaves 176 bits available for each message. In terms of computation cost, encoding adds minimal overhead — just slightly more than a point multiplication on average. Importantly, by using this mapping technique, the assumption that input set elements must be points on the curve can be removed. As a result, by packing the input and trailing 0s into one message, we only need one EC-ElGamal ciphertext instead of two in our proposed mPSU protocol. This directly reduces the communication cost of our protocol by around 50%.

On the other hand, the need for mapping between binary strings and points is only due to the choice of PKE schemes, which is EC-ElGamal in this and prior works. Several lattice-based multi-key encryption schemes support arbitrary message spaces without additional constraints. For example, [CDKS19, MTBH21] are based on the Ring-LWE assumption.

While Ring-LWE HE supports over 1024-bit messages which can be considered as arbitrary message space, it is less efficient. In our tests using BFV implementation from the SEAL library [SEA23], each ciphertext addition (the dominant operation) took 3.91 μs — about 21× slower than EC-ElGamal (0.18 μs). The communication cost is also much higher. While our double ciphertexts require 1,056 bits for each message, Ring-LWE ciphertexts are 55,296 bits — roughly 54× larger — since batching is infeasible due to the shuffle step. However, as shown in [DCZB24], SKE-based protocols have 12× higher runtime and 46× more communication overhead than PKE-based ones for just 10 parties with 2^{16} input items. This cost grows cubically, i.e., $O(n^3)$, with the number of parties. Moreover, these performances of SKE-based protocols were obtained only with 64-bit inputs, which are much smaller compared to the capability of PKE-based protocols. Hence, even PKE-based mPSU constructions instantiated from Ring-LWE schemes would offer comparable overheads to SKE-based approaches.

Nevertheless, current PKE-based mPSU protocols built on EC-ElGamal remain practical despite input domain restrictions (even without employing Koblitz-style try-and-increment technique). For example, the full version of [DCZB24] presents a multiparty private-ID protocol built on mPSU that avoids recovery of elements from elliptic curve points.

B.2 Extension to Support Long Set Elements

As described above, our solution requires that each element in the input set can be encoded in a single multi-key encryption ciphertext. In our current instantiation, each set element can be up to 176 bits or 22 bytes long. When it is desirable to support set elements of longer lengths, our solution can be extended to handle elements that cannot be encoded into a single ciphertext. In this section, we describe a solution to support elements of arbitrary (longer) length.

The high-level idea is to encrypt a long message x using symmetric key encryption with a freshly generated key k as $\mathsf{Enc}(k, x)$. During the message modification phase, the symmetric key is either retained or erased inside multi-key encryption. If the key is retained (i.e., the underlying message is part of the output and must be preserved), the message is decrypted at the end. Otherwise, if the key is erased, all blocks of the message become unrecoverable, achieving all-or-nothing message recovery.

Because any input element can be part of the output, it is necessary to maintain (encrypted) input messages throughout the shuffle phase. Thus, we encrypt $\mathsf{Enc}(k, x)$ using the multi-key encryption scheme introduced earlier (it goes without saying that all inputs are represented using the same number of ciphertext blocks). This is the only phase that requires long input elements to be represented in their entirety, while a single ciphertext block is sufficient in other phases (e.g., element matching can be performed on input elements' hashes instead of the original values).

In more detail, let input set element x be longer than what the HE message space supports. We use its hash $H(x)$ for matching, but represent it as a vector of ciphertexts during the shuffle. To create a vector of ciphertexts, we encrypt x using a symmetric key encryption scheme with a randomly generated key k and obtain $c = \mathsf{Enc}(k, x)$. Here, x will be partitioned into blocks during encryption and c will also consist of the corresponding symmetric encryption blocks. We consequently split c into chunks suitable for encryption under our multi-key encryption schemes. Specifically, let $c = c_1||c_2||\ldots||c_t$, where we create a vector of ciphertexts $e_1, e_2, \ldots, e_t$ with $e_i = \mathsf{Enc}(\mathsf{pk}, c_i)$.

The modifications to our mPSU protocol in Figure 9 are as follows:

(1) During the setup (Phase 0), all parties can use a hash of each message from their set, i.e., $H(x)$ instead x, to construct simple and Cuckoo hashing tables. That is, instead of testing whether x is in X_i, we are testing whether $H(x)$ is in the set containing hashes of all elements of X_i.
(2) In Step (2c), instead of creating ciphertext $e_{k,j}$ as $\mathsf{Enc}(\mathsf{pk}, x||0^\lambda)$, party P_i generates it as $\mathsf{Enc}(\mathsf{pk}, k||0^\lambda)$, where k is a one-time symmetric key associated with P_i's message x. The parties invoke mOT as before using the content of their bins, and the only difference is that ciphertexts $e_{k,j}$ are now constructed in a new way.
(3) In Step (5), when sending $e_{j,k}$ to P_1 at the end of message modification, party P_i appends to $e_{j,k}$ a vector of ciphertexts $e_1, e_2, \ldots, e_t$ created as described above. That is, $c = \mathsf{Enc}(k, x)$, $c = c_1||c_2||\ldots||c_t$, and $e_i = \mathsf{Enc}(\mathsf{pk}, c_i)$. Party P_i then sends $e_{j,k}$ together with $e_1, e_2, \ldots, e_t$ (i.e., a vector of $t + 1$ ciphertexts).
(4) During the shuffle and decryption (Step (6)), the entire vector of $t + 1$ ciphertexts is treated as a single bundle and processed jointly in place of a single ciphertext $e_{j,k}$. All $t+1$ ciphertexts get partially decrypted and randomized by each party during the shuffle.

Once all ciphertexts have been fully decrypted, the message x remains private if the symmetric key k has been erased during the message modification phase; and otherwise x can be recovered by decrypting the symmetric key ciphertext with k.

This extension allows our mPSU protocol to support input elements of arbitrary length while preserving correctness and maintaining practical efficiency. That is, all phases except the shuffle and decrypt operate on values of constant size independent of the bitlength of set elements.

C PROOF FOR THEOREM 3

Proof. We prove Theorem 3 by first creating a simulator that constructs the view of an adversary corrupting a coalition of parties $\{P_c \mid c \in C\}$, where $C \subset [n]$ denotes the set of indices of the corrupted parties. We consequently show that the simulated view is computationally indistinguishable from the view of a real protocol execution. We use notation Sim_c to denote the simulator for a corrupted party P_c. We consider two cases depending on whether P_1 is corrupted or not. The simulator runs the protocol with the following changes:

- **Case 1**: P_1 is not corrupted. To simulate the shuffle and re-randomization step, for each corrupted P_c, if $P_{c-1} \notin C$, sample a set of random ciphertexts encrypted under pk and append them to the view of Sim_c. Due to the security of the multi-key cryptosystem, this is indistinguishable from the real execution.
  To simulate the decryption step, sample a set of random ciphertexts $\{\mathsf{ct}_i \mid i \in [M]\}$ and append them to the view of Sim_c.
- **Case 2**: P_1 is corrupted. To simulate the shuffle and re-randomization step, the view of $\{\mathsf{Sim}_c \mid c \in C, c \neq 1\}$ is constructed in the same way as in Case 1. Now we need to additionally simulate the view of P_1. Given the set $V = \{v_1, \ldots, v_M\}$ that P_1 is to learn as the output, Sim_1 computes $\mathsf{Enc}(\mathsf{pk}, v_j)$ for each $j \in [M]$, samples a random permutation $\pi : [M] \to [M]$, applies it to the ciphertexts, and appends the resulting set of ciphertexts to the view of Sim_1.
  To simulate the decryption step, we need to construct messages from $\{P_i \mid i \in H\}$ and append them to the view of Sim_1. For each ciphertext $\mathsf{ct} = \mathsf{Enc}(\mathsf{pk}, \mathsf{m})$ where $\mathsf{ct} = (\mathsf{ct}_1, \mathsf{ct}_2)$, Sim_1 first computes $\mathsf{ct}_H = \mathsf{m} - \mathsf{ct}_2 + \Sigma_{c \in C} \mathsf{sk}_c \cdot \mathsf{ct}_1$. It consequently generates random elements $\mathsf{ct}_{h,1}$ for each $h \in H$ subject to the constraint $\Sigma_{h \in H} \mathsf{ct}_{h,1} = \mathsf{ct}_H$ and appends them to the view. Note that values $\mathsf{ct}_{h,1}$ are computationally indistinguishable to the messages received from P_h in the last step for each $h \in H$.

The combined view produced by Sim_c for all $c \in C$ is computationally indistinguishable from the real execution. This completes the proof. ☐

D PROOF FOR THEOREM 4

Proof. We start the proof of Theorem 4 by creating a simulator that constructs the view of an adversary corrupting a coalition of parties $\{P_c | c \in C \subset [n]\}$, where C is the set of indices of the corrupted parties. Consequently, we show that the simulated view is computationally indistinguishable from the view of a real protocol execution. We use the notation Sim_c to denote the simulator for a single corrupted party P_c and $\mathsf{Sim}_{\mathcal{A}}$ the simulator for the adversary corrupting all parties in C. As before, we consider two cases depending on whether P_1 is corrupted or not. The simulator works as follows:

Case 1: P_1 is not corrupted.

- In Step (1), $\mathsf{Sim}_{\mathcal{A}}$ randomly selects secret key sk_c for each P_c and randomly samples public key pk. It appends $(\mathsf{sk}_c, \mathsf{pk})$ to the view of each Sim_c.
- In Step (3), consider each pair of P_i, P_j with $1 \le i < j \le n$:
  - If $i \in C$ and $j \in C$, P_i and P_j execute the protocol as usual.
  - If $i \notin C$ and $j \in C$, Sim_j generates randomly sampled bins $S_{i,k}$ for $k \in [\mu]$ representing P_i's mOT inputs (note that the simulators for all corrupt parties consistently use bins generated on P_i's behalf). Sim_j calls the batch mOT simulator $\mathsf{Sim}^{\mathcal{S}}_{batch\mathsf{mOT}}$ with P_j's inputs $C_{j,k}, z_{i,k}, f_{i,k}$ and bins $S_{i,k}$ for $k \in [\mu]$ and appends the output to the view. Sim_j creates a randomly generated ciphertext and appends it to the view as $e_{i,j,k}$ for each $k \in [\mu]$.
  - If $i \in C$ and $j \notin C$, Sim_i calls the $\mathsf{Sim}^{\mathcal{R}}_{batch\mathsf{mOT}}$ with P_i's bins $S_{i,k}$ and simulated (randomly generated) bins $C_{j,k}$ and ciphertexts $\mathsf{Enc}(\mathsf{pk}, 0)$, $\mathsf{Enc}(\mathsf{pk}, r)$ for $k \in [\mu]$. It appends the simulated view to its view.
- In Step (6), each Sim_c calls the $\mathsf{Sim}^{P_c}_{\mathsf{Shuffle\&Decrypt}}$ with input sk_i and a random permutation π_i, and appends its output to the view of Sim_c.

$\mathsf{Sim}_{\mathcal{A}}$ combines the views of Sim_c for each $c \in C$. Now we argue the output of $\mathsf{Sim}_{\mathcal{A}}$ is indistinguishable from the real execution. The argument relies on the security of batch mOT and Shuffle&Decrypt. We set up all invocations of mOT to use mutually independent messages that form the output in both real ($z_{i,k}$ and $f_{i,k}$ supplied by P_j when interacting with P_i) and simulated (fresh $\mathsf{Enc}(\mathsf{pk}, 0)$ and $\mathsf{Enc}(\mathsf{pk}, r)$) executions. This means that even when the adversary has the combined view for each corrupted party P_c, the output of batch mOT provides no information to the corrupted parties and is indistinguishable from the real execution.

In addition, when $i \notin C$ and $j \in C$ in Step (3), the randomly drawn ciphertext used in the simulation is indistinguishable from a ciphertext of a real execution by the security of EC-ElGamal cryptosystem re-randomization.

Lastly, the simulation of Step (6) is the same as the simulation of Case 1 in the proof of Theorem 3 and retains the indistinguishability property.

Case 2: P_1 is corrupted. This means that the simulator has access to the output $\bigcup_{i=1}^{n} X_i$ and will use it in constructing P_1's view. Portions of the simulation proceed as described for Case 1, while we introduce the changes as described below. At high level, ciphertexts corresponding to messages from $\bigcup_{c \in C} X_c$ are retained in corrupt parties' ciphertexts, while the remaining MPSU output messages $(\bigcup_{i=1}^{n} X_i) \setminus (\bigcup_{c \in C} X_c)$ are contributed by the simulator on behalf of honest parties.

- In Step (3), when $i \notin C$ and $j \in C$, Sim_j calls the simulator $\mathsf{Sim}^{\mathcal{S}}_{batch\mathsf{mOT}}$ in step (3)(a) as before. To create the ciphertext P_j receives in step (3)(b), instead of choosing a

PARAMETERS: n parties, the set size M. A multi-key encryption scheme defined in Section 3.5

INPUT:

- Each party $P_{i\in[n]}$: The secret key sk_i and a permutation function $\pi_i : [M] \to [M]$. Here, $(\mathsf{pk}, \{\mathsf{sk}_i\}_{i\in[n]}) \leftarrow \mathsf{KeyGen}(1^\kappa, n)$.
- P_1 has a set $C = \{\mathsf{ct}_1^1, \ldots, \mathsf{ct}_M^1\}$

PROTOCOL:

(1) P_1 re-randomizes $\mathsf{ct}_j = \mathsf{ReRand}(\mathsf{ct}_j, \mathsf{pk}), \forall j \in [M]$, and sends $C_1 = \{\mathsf{ct}_1^1, \ldots, \mathsf{ct}_m^1\}$ to P_2 where $\mathsf{ct}_j^1 = \mathsf{ct}_{\pi_1(j)}, \forall j \in [M]$.

(2) For $i = 2$ to n:
  - P_i computes a partial decryption $\bar{\mathsf{ct}}_j^i = \mathsf{ParDec}(\mathsf{sk}_i, \mathsf{ct}_j^{i-1}, A_i), \forall j \in [M]$, where $A_i = \{1, i, i+1, \ldots, n\}$.
  - P_i permutes the set $\{\bar{\mathsf{ct}}_1^i, \ldots, \bar{\mathsf{ct}}_M^i\}$ as $\mathsf{ct}_j^i = \bar{\mathsf{ct}}_{\pi_i(j)}, \forall j \in [M]$.
  - P_i sends $C_i = \{\mathsf{ct}_1^i, \ldots, \mathsf{ct}_M^i\}$ to $P_{(i+1) \bmod n}$.

(3) P_1 outputs $\mathsf{ParDec}(\mathsf{sk}_1, \mathsf{ct}_j^n, \{1\}), \forall j \in [M]$.

Figure 12: Oblivious Shuffle and Decryption (Shuffle&Decrypt) Construction [GNT24].

PARAMETERS: Two parties: Sender and Receiver

FUNCTIONALITY:

- Wait for input strings $(x_0, x_1) \subset (\{0,1\}^\star)^2$ from the sender.
- Wait for input choice bit $b \in \{0,1\}$ from the receiver.
- Give x_b to the receiver.

Figure 11: Oblivious Transfer (OT) Functionality.

random ciphertext as in Case 1, we set it to be $\mathsf{Enc}(\mathsf{pk}, 0)$ or $\mathsf{Enc}(\mathsf{pk}, r)$ based on the corrupt parties' inputs. Specifically, for each element $x \in X_j$, if $x \in \bigcup_{c\in C, c<j} X_c$, Sim_j samples a random ciphertext $\mathsf{Enc}(\mathsf{pk}, r)$ and appends it to P_j's view in that step. Otherwise, it uses $\mathsf{Enc}(\mathsf{pk}, 0)$.

- In Step (6), we distribute the remaining output values $\bigcup_{i=1}^n X_i \setminus \bigcup_{c\in C} X_c$ that P_1 is to recover from the ciphertexts among the ciphertexts contributed by the (simulated) honest parties. To this extend, the simulator Sim_1 appends ciphertext $\mathsf{Enc}(\mathsf{pk}, x)$ to the view if $x \in \bigcup_{i=1}^n X_i \setminus \bigcup_{c\in C} X_c$, and otherwise appends random $\mathsf{Enc}(\mathsf{pk}, r)$.

As a result, we obtain correctness. Namely, P_1 recovers $\bigcup_{i=1}^n X_i \setminus X_1$ from the ciphertexts after the shuffle and decrypt phase, which is the same as in the real protocol execution.

To see why indistinguishability holds, one important observation is that participant P_j does not know the random values that its ciphertexts $f_{i,k}$ (generated in step (2)(e) and used while interacting with P_i in step (3)(a)) encode. This means that when element x is erased from the output by adding $\mathsf{Enc}(\mathsf{pk}, r)$ and $\mathsf{Enc}(\mathsf{pk}, x)$, no party, including the owner of $\mathsf{Enc}(\mathsf{pk}, r)$, knows the value of r. If this was not the case, it would be possible in some circumstances for corrupt parties to recognize their (erased) messages among decrypted ciphertexts. However, with the current design, the corrupt parties are unable to distinguish erased messages from random elements from the plaintext space, and the simulation achieves indistinguishability. □

E OBLIVIOUS TRANSFER

Oblivious Transfer (OT) is a fundamental primitive of secure computation introduced by Rabin [Rab98]. It refers to the problem where a sender with two input strings (x_0, x_1) interacts with a receiver with an input bit b. As a result of the (1-out-of-2) OT, the receiver learns x_b, and the sender learns nothing. Figure 11 presents the OT functionality.

F OBLIVIOUS SHUFFLE AND DECRYPTION

Figure 12 presents the oblivious shuffle and decryption protocol of [GNT24].