Abstract

This paper studies a multi-party private set union (mPSU), a fundamental cryptographic problem that allows multiple parties to compute the union of their respective datasets without revealing any additional information. We propose an efficient mPSU protocol which is secure in the presence of any number of colluding semi-honest participants. Our protocol avoids computationally expensive homomorphic operations or generic multi-party computation, thus providing an efficient solution for mPSU.

The crux of our protocol lies in the utilization of new cryptographic tool, namely, Membership Oblivious Transfer (mOT). We believe that the mOT may be of independent interest. We implement our mPSU protocol and evaluate their performance. Our protocol shows an improvement of up to 80.84× in terms of running time and 405.73× bandwidth cost compared to the existing state-of-the-art protocols.

1 Introduction

Secure multi-party computation (MPC) enables multiple parties to compute an arbitrary function on their private input without revealing additional information. A special case of MPC is the private set operation, which provides a secure means for joining data distributed across disparate databases. Private set intersection (PSI) and private set union (PSU) are two common set operations in this category. PSI finds applications in a variety of privacy-sensitive scenarios such as measuring the effectiveness of online advertising [IKN+20, MMT+24], contract tracing [TSS+20, BBV+20], contact discovery [HWS+22], associated rule learning [GTY24], and cache sharing in IoT [NT21]. Similarly, PSU has numerous practical use cases. For example, PSU can be used to implement 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].

Over the last decade, a substantial body of research [PSZ18, RR22, BC23] has focused on PSI, whereas PSU has received relatively little attention. The majority of present practical PSU protocols [KRTW19, GMR+21, ZCL+23, JSZ+22, BPSY23] have only been optimized for the two-party setting. In this study, we investigate multi-party PSU (mPSU) in the semi-honest model, which allows more than two parties to compute the union of their private data sets without revealing additional information.
1.1 Multi-Party PSU vs 2-Party PSU

Multi-party PSU is a natural extension of the two-party PSU and enables much richer data sharing than a two-party PSU. Collection of data from more participants will surely improve the performance of the data-driven applications that we mentioned above. However, designing a multi-party protocol in secure computation is challenging as it usually requires a dishonest majority (e.g. provides security in the presence of a number of dishonest, colluding participants). Existing mPSU protocols in generic MPC [BA12, VCE22], or homomorphic encryption [KS05, Fri07, SCK12, GHJ22], are considerably more complex and expensive in the multiparty case than in the two-party case.

A possible solution for computing mPSU is leveraging efficient multi-party PSI protocols. Given their recent PSI improvements [CDG+21, NTY21] with practical implementations, one might think that mPSU can be computed directly from multi-party PSI using DeMorgan’s Law as $\bigcup_{i=1}^{n} X_i = U \setminus \bigcap_{i=1}^{n} (U \setminus X_i)$, where $U$ is a universe of input items. While this approach correctly and securely computes the set union, it is inefficient when $U$ is significantly larger than $\bigcup_{i=1}^{n} X_i$. Thus, this solution is still far from practical.

Another potential approach is to extend the aforementioned practical two-party PSI protocols [PSZ18, RR22, BC23] to the multi-party case. However, it remains unclear how to achieve a secure mPSU protocol through this extension since the intermediate result would leak information like the intersection or intersection cardinality or union of a subset of parties’ inputs which not only violates the mPSU functionality but also harms the privacy of the data owner.

In the application of Cyber risk assessment [HLS+16] previously mentioned and elaborated in [KRTW19, JSZ+22, ZCL+23], organizations want to compute the union of the IP blacklist while requires minimal leakage from the union. Consider data-driven applications as Private-ID [BKM+20] and association rule learning [KC04], where companies aim to build a joint dataset, better performance can be achieved by having richer data from more participants [HNP09]. However, leakage of the input dataset of any company leads to disastrous consequences. If we implement mPSU using pairwise 2-party PSU, in some cases, privacy may not be guaranteed at all! For example, in a 3-party cases where $P_1$ is the receiver, if $X_3 \cap (X_1 \cup X_2) = \phi$, $P_1$ learns the $P_3$’s data set completely. This issue arises not only when applying 2-party PSU protocol in a multi-party setting but also when $P_1$ colludes $P_2$, as even the leakage of the count of elements can lead to the same problem. This risk is exacerbated when dealing with sensitive information such as healthcare data or financial records. To address these privacy concerns, an mPSU protocol that prevents any additional information leakage and is resilient to arbitrary collusion is highly needed for the multi-party scenario.

To grasp the challenges of extending from two-party to multiple-party PSU, we begin by reviewing the state-of-the-art 2-party PSU protocols [GMR+21, ZCL+23, JSZ+22, BPSY23], which follow the framework of [KRTW19] based on oblivious transfer (OT), which consists of two main stages:

1. Reverse Private Membership Test (RPMT): The receiver learns the bit representing the membership of each element in the sender’s set. (e.g. for an element $x$ in sender’s set, the receiver with set $Y$ learns a bit $b = 1$ if $x \in Y$ and $b = 0$ otherwise.). Note that the bit $b$ reveals no additional information about the sender’s set $X$, apart from the intersection cardinality $|X \cap Y|$, which is already revealed by the final PSU output.

2. Oblivious Transfer (OT): The sender obliviously sends each item $x$ in its set $X$ to the receiver using OT. Concretely, the sender and the receiver invoke an OT functionality in which the
sender possesses messages \{⊥, x\} while the receiver holds the choice bit \(b\), where ⊥ represents a predefined special character. The bit \(b\) is the membership indicator bit which is derived from the preceding stage. The result of the OT provides the receiver with either ⊥ or the sender’s item \(x\) which is not the intersection item. By merging this outcome with its set \(Y\), the receiver can produce the set union. This OT step prevents the receiver from deducing the intersection set, thereby fulfilling the functionality of a two-party PSU.

To summarize, in the two-party protocol, the parties initially establish the sender’s element membership, followed by the receiver obliviously obtaining only the set difference from the sender. While this framework functions effectively and securely for the 2-party PSU, it cannot be directly extended to multi-party settings due to various sources of information leakage. To be more precise, assume there are \(n\) parties, each with a set \(X_i\), and \(P_1\) is the one who receives the final output. Considering a single element \(x \in \bigcup_{i=2}^{n} X_i \setminus X_1\) which will be learned by \(P_1\) from a PSU protocol, there are two types of information leakage considered in the multi-party setting:

- **Which party sends this element \(x\)?** The initial potential leakage arises from the origin of \(x\). If \(P_1\) and \(P_{i \in [2:n]}\) invoke OT in the same manner as 2-party protocols, \(P_1\) will know the contribution for the received element which is indeed an information leakage in the multi-party setting.

- **How many \(x\) are there?** Another potential leakage is the number of element \(x\). In a 2-party setting, this count is consistently one, as the sender is the sole provider of new elements to the receiver (assuming that the \(X_2\) is not a multi-set). In a multi-party setting, for element \(x \in \bigcup_{i=2}^{n} X_i \setminus X_1\), any \(P_{i \in [2:n]}\) can have it in the input set. So the number of duplication’s can range from 1 to \(n - 1\).

In general, any information that cannot be derived from the final output is not allowed. In the case of mPSU, the potential information leakage can be the union or intersection of the input from a subset of participants which can be addressed by avoiding the two leakages mentioned above. Thus, in the multi-party setting, the definition and execution of RPMT and OT must differ from those in the 2-party setting. Furthermore, another main challenge in designing mPSU is to prevent leakages in the event of collusion among a subset of parties.

### 1.2 Related Work

In this section, we focus on the state-of-the-art of multi-party PSU protocols. The earliest construction of such a protocol was proposed by Kissner and Song [KS05], which relied heavily on homomorphic encryption (the Paillier encryption) and the idea of polynomial representation. Input sets are represented as polynomials where each party \(P_{i \in [n]}\) represents an input set \(X_i = \{x_{i,1}, \ldots, x_{i,m}\}\) as a polynomial whose roots are its elements, which we denote \(f_i(x) = \prod_{j=1}^{m} (x - x_{i,j})\). All parties together compute the encryption of polynomial \(p = \prod_{i=1}^{n} f_i\) which presents the polynomial of the union \(\bigcup_{i=1}^{n} X_i\). Using polynomial evaluation on the encrypted \(p\), all parties are able to extract the union items without disclosing additional information. The protocol proposed in [KS05] has \(O(n^3m^2)\) computation complexity. Relying on the polynomial presentation technique, Frikken [Fri07] proposed an efficient mPSU protocol that requires the \(O(n^2m\log(m))\) number of multiplications. [SCK12] presented input sets using rational polynomial functions and reversed Laurent series. As a result, it showed a more efficient protocol than previous works [KS05, Fri07], but the protocol is secure up to \(n/2\) corrupted parties.
Blanton and Aguiar [BA12] presented a new direction to compute \( mPSU \) that avoids expensive homomorphic encryption but heavily relies on MPC. Their idea is to combine the input sets of all parties under a secret-shared form, perform an oblivious sort on the resulting set, and then remove the duplications by comparing the adjacent elements. In the context of MPC, a more practical sorting algorithm is Batcher’s network which requires \( O(mn \log(mn)) \) comparisons to sort the union sets. Due to the underlying MPC techniques, the protocol of [BA12] is inefficient when the \( m \) and \( n \) are large.

[SM18, GHJ22] compute the \( mPSU \) using Bloom filter (BF). Specifically, each party \( P_i \in [1,n] \) inserts its input items into a local BF and transmits the encrypted version of the resulting BF to a designated leader party \( P_1 \). Subsequently, the \( P_1 \) aggregates the encrypted local BFs from all parties to generate a global BF, denoted by \( G \), from which the union items are computed. While the protocol presented in [SM18] makes use of an outsourcing server to compute \( G \), [GHJ22] is built on homomorphic encryption (HE), which requires a homomorphic computation per each entry of \( G \), and might need expensive multi-key HE. Moreover, the BF-based approach is associated with a high false positive rate.

In another work, Vos et al. [VCE22] proposed private OR protocols and build \( mPSU \) protocols upon it. They consider a relatively small universe (e.g. up to 32-bit long element). At the high-level idea, their approach presents the input set in a bit vector of length \( |U| \). The bit is set to 1 if its corresponding element belongs to the given input set and 0 otherwise. By invoking the proposed private OR protocol, the leader learns the bit vector of the union. While optimization is given by applying divide-and-conquer so that the long vector can be divided into small ones, it is still inefficient, especially for the standard input of 128-bit elements. Concurrently with our work, Liu and Gao [LG23] presented an efficient \( mPSU \) protocol but requires a weak security assumption wherein the leader is not in collusion with any other participating parties.

For a comprehensive analysis of representative multi-party PSU protocols that are resilient to the presence of any number of colluding semi-honest participants, we provide a summary of their theoretical complexity in Table 1. Additionally, in Section 5.2, we present a numerical performance comparison of our proposed protocols with prior works [Fri07, BA12, GHJ22].

Very recently, Dong et al. [DCZB24] points out a security flaw in our original protocol. Our updated protocol (Figure 8) is an easy fix by removing the redundant step for computing PRF. Refer to Appendix B for details. [DCZB24] proposed a new batched SS-PMT construction which outperforms the multi-query SS-PMT proposed in [LG23] in the usage of \( mPSU \). Following our \( mPSU \) framework where \( P_1 \) collects encryptions of the union items from other participants with the multi-key cryptosystem (Section 2.5) and Shuffle&Decrypt (Section 3.2), they presented a new efficient \( mPSU \) protocol from the batched SS-PMT and random OT. Their protocol is secure against arbitrary collusion and achieves linear computation and communication cost in terms of number of elements which outperform our work.

1.3 Technical Overview of Our Protocols

We present an efficient protocol for \( mPSU \) that guarantees security in the semi-honest setting. We demonstrate the practicality of our \( mPSU \) protocol with an implementation. It is shown to be efficient even for large sets with \( 2^{20} \) items distributed among 8 parties. The main reason for our protocol’s high performance is its reliance on fast symmetric-key primitives and ElGamal encryption. This is in contrast with prior protocols, which require expensive Paillier encryption on the polynomial set representation [KS05, Fri07] or each entry of the Bloom filter [GHJ22].
In our protocol, we assume the existence of a leader party denoted as $P_1$. This party learns the final result by growing the union starting with $X_1$. To be specific, $P_1$ learns $X_t \setminus \bigcup_{i=1}^{t-1} X_i$ from $P_t$. This is achieved by interacting sequentially with each party $P_2, \ldots, P_n$. All the parties agree on a multi-key cryptosystem for encryption, and sets are encrypted to prevent the leader from learning the partial union. Moreover, we propose new primitives Membership Oblivious Transfer (mOT) to ensure the correctness of the final result as well as prevent the information leakage introduced earlier in the Section 1.1.

Briefly, the mOT is a two-party protocol, in which a leader $P_1$ (also referred to as the receiver) holding a set $X_1$ interacts with the sender $P_i$ who possesses an input item $x_{t,j}, j \in [m]$, and two associated values $\{v_0, v_1\}$. Similar to the traditional OT [Rab98], the result is that the sender $P_t$ learns nothing whereas the receiver $P_1$ obtains one of the two sender’s associated values depending on whether $x_{t,j} \in X_1$.

In our mPSU protocol, sender $P_t$ prepares $v_0 = \text{Enc}(pk, 0)$ and $v_1 = \text{Enc}(pk, x_{t,j})$, where $pk$ is the public key for the multi-key cryptosystem. If $x_{t,j} \in X_1$, $P_1$ learns $\text{Enc}(pk, 0)$; otherwise it learns $\text{Enc}(pk, x_{t,j})$. By executing the mOT multiple times with $P_{t_i} \in [2,n]$ for each item $x_{t,j} \in X_t$, the leader $P_1$ obtains a set $E$ of encryptions $\text{Enc}(pk, x_{i,j})$ for $x_{i,j} \in \bigcup_{i=1}^{t-1} X_i \setminus X_1$ and the number of encryptions of zero. At this point, the set $E$ still contains the encryption of the $X_i \cup X_1$ for $i, t > 1$. To remove these encryptions, before executing with $P_1$, we require $P_t$ executes an mOT with each $P_{t_i} \in [2,t-1]$. As the result, $P_t$ holds $\text{Enc}(pk, x_{t,j})$ if the item $x_{t,j}$ is not in any set $X_2, \ldots, X_{t-1}$, and $\text{Enc}(pk, 0)$ otherwise. Now, the union can be obtained by decrypting $E$ and removing the zeros.

For the dishonest majority setting in which the protocol is secure against an arbitrary number
Parameters: \( n \) parties \( P_1, \ldots, P_n \), and the set size \( m \).

Functionality:

- Wait for input set \( X_i \) of size \( m \) from \( P_i \).
- Give \( P_1 \) the union \( \bigcup_{i=1}^{n} X_i \).

Figure 1: Multi-party Private Set Union Ideal Functionality

of colluding parties, the decryption should be executed by all the parties. Thus, we employ the multi-key cryptosystem based on the ElGamal encryption scheme (ref. Section 2.5). The decryption process involves a partial decryption that requires the individual party’s secret key. In our protocol, each party is required to perform its own private permutation on the partial decryption result before sending it to another party. This step aims to prevent a coalition of corrupt parties (including the leader \( P_1 \)) from learning which parties hold which elements. We implement the permutation and decryption using our simple building block “Oblivious Shuffle and Decryption” (Shuffle\&Decrypt), which is described in Section 3.2.

In brief, our contributions can be summarized as follows:

- We present an efficient mPSU construction, which eliminates the need for computationally expensive homomorphic operations or generic multi-party computation and is secure in the presence of any number of colluding semi-honest participants.
- We introduce new building blocks, namely Membership Oblivious Transfer (mOT) and Oblivious Shuffle and Decryption (Shuffle\&Decrypt), which may be of independent interest and can be used in other related protocols.
- We show that our protocol is significantly faster than previous work [Fri07, BA12, GHJ22]. For example, for four parties with a dataset of \( 2^{16} \) item each, our mPSU protocol shows an improvement up to \( 80.84 \times \) in terms of running time and up to \( 405.73 \times \) less bandwidth requirement when compared to the state-of-the-art protocols. Our implementation is publicly available at https://github.com/asu-crypto/mpsu.

2 Preliminaries

In this work, the computational and statistical security parameters are denoted by \( \kappa, \lambda \), respectively. We use \([m]\) to refer to the set \( \{1, \ldots, m\} \), and \([i, j]\) to denote the set \( \{i, \ldots, j\} \). We denote the concatenation of two strings \( x \) and \( y \) by \( x||y \). We use \( f \circ g \) to denote the composition of the functions \( f \) and \( g \).

2.1 Multi-party Private Set Union

The ideal functionality of multi-party PSU (mPSU) is given in Figure 1. It allows \( n \) parties, each holding a set \( X_i \) of the input items, to learn the union \( \bigcup_{i=1}^{n} X_i \) and nothing else. For simplicity, we assume that all parties have the same set size \( m \), which is publicly known.
**Threat Model and Security Goal.** From the ideal functionality of mPSU, we can see that the mPSU protocol is secure if the mPSU protocol is considered secure as long as it does not disclose any additional information beyond the union and $m$ to the parties, encompassing partial set union/intersection.

Note that our protocol can be easily extended to accommodate varying set sizes, while also protecting the set size of each party. This can be accomplished if all parties agree on an upper bound set size $m$ and utilize it as the input set size. Before initiating the protocol, each party can pad their set with a particular item, such as zero, to reach the size of $m$. It is customary in private set operation literature to assume that all parties have the same set size.

In this paper, we focus on the semi-honest setting, where it is assumed that parties adhere to the protocol description but attempt to glean additional information from the protocol’s transcript.

### 2.2 Oblivious Transfer

Oblivious Transfer (OT) is a fundamental primitive of secure computation and was introduced by Rabin [Rab98]. It refers to the problem where a sender with two input strings $(x_0, x_1)$ interacts with a receiver who has an input choice bit $b$. The OT gives the receiver $x_b$ and nothing to the sender. Figure 2 presents the OT functionality.

**Parameters:** Two parties: Sender and Receiver

**Functionality:**

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

Figure 2: Oblivious Transfer (OT) Ideal Functionality.

### 2.3 Secret-shared Private Membership Test

Secret-shared Private Membership Test (SS-PMT) is the main building block in different applications [PSTY19, LPR+21, CDG+21, PSWW18, LG23, ZCL+23]. It refers to the two-party setting where a $P_0$ with input a set of items $X = \{x_1, \ldots, x_n\}$ interacts with a $P_1$ who has an input single item $y$. SS-PMT gives both parties a secret share of a membership bit, i.e. the two parties obtain XOR shares of 1 if $y \in X$ and 0 otherwise. Figure 3 presents the SS-PMT functionality.

### 2.4 Bin-and-ball Scheme

Our protocols employ hashing schemes such as the Cuckoo and Simple hashing schemes [PSSZ15, PSZ18] to allocate items into bins. We review the basics of the Cuckoo hashing and Simple hashing schemes [PSSZ15, PSZ18] as follows.

**Cuckoo hashing.** In basic Cuckoo hashing, there are $\mu$ bins denoted $B[1, \ldots, \mu]$, a stash, and $h$ random hash functions $H_1, \ldots, H_h : \{0, 1\}^* \rightarrow [\mu]$. One can use a variant of Cuckoo hashing such that each item $x \in X$ is placed in exactly one of $\mu$ bins. Using the Cuckoo analysis [PSSZ15,
Parameters: Two parties: $P_0$ and $P_1$, and the set size $n$.

Functionality:

- Wait for input a set of items $X = \{x_1, \ldots, x_n\} \subset (\{0, 1\}^*)^n$ from the $P_0$.
- Wait for input item $y \in \{0, 1\}^*$ from the $P_1$.
- Give $b_i$ to the $P_i \in \{0, 1\}$ where $b_0 \oplus b_1 = 1$ if $y \in X$ and 0 otherwise.

Figure 3: Secret-shared Private Membership Test (SS-PMT) Ideal Functionality.

[DRRT18] based on the set size $|X|$, the parameters $\mu, h$ are chosen so that with high probability $(1 - 2^{-\lambda})$ every bin contains at most one item, and no item has to be placed in the stash during the Cuckoo eviction (i.e. no stash is required).

Simple hashing. One can map its input set $Y$ into $\mu$ bins using the same set of $h$ Cuckoo hash functions (i.e. each item $y \in Y$ appears $h$ times in the hash table). Using a standard ball-and-bin analysis based on $h, \mu$, and $|X|$, one can deduce an upper bound $\eta$ such that no bin contains more than $\beta$ items with high probability $(1 - 2^{-\lambda})$.

2.5 Multi-key Cryptosystem

We revise the multi-key cryptosystem that is needed for our mPSU protocol. We first give an overview of each component of the cryptosystem. We then present a construction based on the ElGamal scheme. A multi-key cryptosystem is defined as a tuple of PPT algorithm $(\text{KeyGen}, \text{Enc}, \text{ParDec}, \text{FulDec}, \text{ReRand})$ with properties as follows:

- **Key Generation:** $\text{KeyGen}(1^\kappa, n)$. In a setting with $n$ parties, a key generation algorithm takes security parameter $\kappa$ as input and gives each party $P_i$ a secret key $sk_i$ and a joint public key $pk = \text{Combine}(sk_1, sk_2, \ldots, sk_n)$, where $\text{Combine}$ is an algorithm to generate the public key from the input secret keys depending on the construction.

- **Encryption:** $ct \leftarrow \text{Enc}(pk; m)$. Given a joint public key $pk$ and a message $m \leftarrow \mathcal{M}$ from the plaintext space $\mathcal{M}$, an encryption algorithm outputs a ciphertext $ct$.

- **Decryption:** There are two types of decryption:
  
  - Partial decryption $ct' \leftarrow \text{ParDec}(sk_i, ct, A)$. A partially decryption algorithm takes a secret key $sk_i$ and a ciphertext $ct \leftarrow \mathcal{C}$ encrypted under the partial public key $pk_A = \text{Combine}(\{sk_j \mid j \in A\})$ and outputs a ciphertext $ct' \leftarrow \mathcal{C}$ which is encrypted under the partial public key $pk_{A \setminus \{i\}} = \text{Combine}(\{sk_j \mid j \in A, j \neq i\})$. Note that in the context of the multi-key encryption system, we utilize set $A$ to represent the collection of public keys belonging to the parties within $A$.
  
  - Full decryption: $m \leftarrow \text{FulDec}(sk_1, sk_2, \ldots, sk_n; ct)$. A full decryption algorithm takes a ciphertext $ct \leftarrow \mathcal{C}$ encrypted under $pk$ and all the secret keys and outputs a message $m \leftarrow \mathcal{M}$.
• **Re-randomization:** $ct' \leftarrow \text{ReRand}(ct, pk)$. This algorithm takes a ciphertext $ct \leftarrow C$ encrypted under $pk$ and gives a re-randomized ciphertext $ct' \leftarrow C$ encrypted under the same $pk$ such that they are both encryptions of the same message $m \leftarrow M$.

The multi-key cryptosystem should satisfy correctness and security as defined in [Gen09, AJL+12, Bra12]. Informally, the multi-key cryptosystem satisfies correctness if $m = \text{FulDec}(sk_1, \ldots, sk_n, ct)$ or $m = \text{FulDec}(sk_1, \ldots, sk_{i-1}, sk_{i+1}, \ldots, sk_n, ct')$ for $ct = \text{Enc}(pk, m)$ and $ct' = \text{ParDec}(sk_i, ct_{\{1, \ldots, i-1, i+1, \ldots, n\}})$. For security, the ciphertext $ct$ or $ct'$ is random and reveals nothing about the plaintext. When $n = 1$, we have a single-key encryption scheme which is indeed the traditional ElGamal system [ElG84].

**A Construction** While there are many multi-key cryptosystems, we choose ElGamal system [ElG84] as it is easy to implement and efficient (we do not perform any arithmetic computation on the encryption). In the following, we present the ElGamal scheme in the multi-key setting with $n$ parties $P_1, \ldots, P_n$.

• **Key Generation:** Given a security parameter $\kappa$ and number of parties $n$. A cyclic group $G$ of order $p$ is chosen, and all the parties agree on a common generator $g$. Each party $P_i \in [n]$ chooses a random secret key $sk_i \leftarrow \{0, 1\}^\kappa$ and publishes the value of $h_i = g^{sk_i}$. We can define the public key $pk = \text{Combine}(sk_1, sk_2, \ldots, sk_n) = g^{\sum_{i=1}^n sk_i} = \prod_{i=1}^n h_i$.

• **Encryption:** To encrypt a message $m$, one can compute $ct = (ct_1, ct_2) = (g^r, m \cdot pk^r)$ where $r$ is a randomly chosen value from $\{0, 1\}^\kappa$.

• **Decryption:** The two decryption algorithms are as follows:
  - Partial decryption: To partially decrypt a ciphertext $ct = (ct_1, ct_2)$ encrypted under the partial public key $pk_A = \prod_{j \in A} h_j$, one output $ct' = \text{ParDec}(sk_i, ct, A) = (ct'_1, ct'_2)$, where $ct'_1 = ct_1 \cdot g^{r'}$, $ct'_2 = ct_2 \cdot ct_1^{-sk_i} \cdot (pk_{A \setminus \{i\}})^{r'}$, the $r' \leftarrow \{0, 1\}^\kappa$ is a random value, and $pk_{A \setminus \{i\}} = \prod_{j \in A \setminus \{i\}} h_j$. Note that the use of the random $r'$ aims to re-randomize the ciphertext.
  - Full decryption: To fully decrypt a ciphertext $ct = (ct_1, ct_2)$ encrypted under $pk = \prod_{i \in [n]} h_i$, one can compute $m = \text{FulDec}(sk_1, sk_2, \ldots, sk_n, ct) = ct_2 \cdot ct_1^{-\sum_{i=1}^n sk_i}$.

• **Re-randomization:** To rerandomize a ciphertext $ct$ encrypted under the $pk$, one can choose a random value $r'' \leftarrow \{0, 1\}^\kappa$, and compute $(ct'_1, ct'_2) = \text{ReRand}((ct_1, ct_2), pk)$ where $ct'_1 = ct_1 \cdot g^{r''}$ and $ct'_2 = ct_2 \cdot pk^{r''}$.

### 3 Our mPSU Building Blocks

We introduce two simple cryptographic gadgets that will serve as the fundamental building blocks in our mPSU protocol.

• The first gadget is called “Membership Oblivious Transfer” (mOT) which enables a receiver to obtain one of two associated values from the sender based on the set membership. The mOT allows a leader party in our mPSU protocol to obliviously retrieve the items of other parties that are not in the intersection while maintaining privacy.
PARAMETERS: Sender $S$ and Receiver $R$, the receiver set size $m$, the length $\ell$.

FUNCTIONALITY:

- Wait for input keyword $y$ and a pair $(v_0, v_1) \in \{0, 1\}^\ell \times \{0, 1\}^\ell$ from $S$.
- Wait for input set $X = \{x_1, \ldots, x_m\}$ from $R$.
- Give $R$ the value $v$ where $v$ equals to $v_0$ if $y \in X$, and $v_1$ otherwise.

Figure 4: Membership Oblivious Transfer (mOT) Ideal Functionality

• In our mPSU protocol, the union result is stored under the multi-key encryption until the final step, which requires all parties to decrypt the ciphertexts together. The encryption protects against corrupted parties from learning partial union. We revise a multi-key cryptosystem in Section 2.5, and introduce a simple tool called “Shuffle and Decryption” (Shuffle&Decrypt) to implement the last step of our mPSU construction.

In the following, we present the definition and ideal functionality of each building block, which specify the input and output. Parties should not gain any additional knowledge beyond the desired output, ensuring the security of each introduced primitive.

3.1 Membership Oblivious Transfer (mOT)

Definition 1. Membership Oblivious Transfer (mOT) is a two-party protocol, in which a sender $S$ with a keyword $y \in \{0, 1\}^*$ and two associated values $(v_0, v_1) \in (\{0, 1\}^\ell)^2$ interacts with a receiver $R$ who has a set of keywords $X = \{x_1, \ldots, x_m\} \in (\{0, 1\}^*)^m$. Except randomnesses, the mOT functionality gives the receiver the value $v_b$ where $b = 0$ if $y \in X$ and $b = 1$ otherwise, and nothing to the sender.

Similar to the traditional OT, the associated values $v_0, v_1$ are indistinguishable with respect to their domain $\{0, 1\}^\ell$, so that the membership of $y$ in terms of $X$ is also not revealed to the receiver. We name our gadget “Membership Oblivious Transfer” as the receiver’s obtained value depends on whether $y \in X$. We formally describe the mOT ideal functionality in Figure 4.

From Definition 1, we see that if a construction for mOT is secure, it should satisfy two following properties:

- Similar to the traditional one-out-of-two oblivious transfer [Rab98], the receiver $R$ only learns one of the two associated values of the sender $S$. In addition, the receiver $R$ has no information about whether $y \in X$ is from the protocol’s output. In fact, the latter is satisfied if the associated values $(v_0, v_1)$ are sampled according to the same distribution.
- The sender $S$ learns nothing about the receiver’s input and output.

To sum up, our security objective for mOT is to enable the sender to anonymously transmit one of its associated values to the receiver based on the membership condition.
PARAMETERS:
- Sender $S$ and Receiver $R$, the receiver set size $m$, the length $\ell$.
- The OT and SS-PMT functionalities described in Section 2.

INPUT:
- Receiver $R$: $X = \{x_1, \ldots, x_m\} \subset \{0, 1\}^m$
- Sender $S$: $y \in \{0, 1\}^\ell$ and two associated values $(v_0, v_1) \subset (\{0, 1\}^\ell)^2$

PROTOCOL:
1. The sender $S$ and the receiver $R$ invoke a SS-PMT functionality where:
   - $R$ has an input set $X$, and $S$ has an input $y$.
   - $S$ and $R$ obtain the bit $b_S$ and $b_R$, respectively. Here, $b_S \oplus b_R = 1$ if $y \in X$ and 0 otherwise.
2. $S$ and $R$ invoke an OT instance where:
   - $S$ acts as an OT sender with input two strings $\{r \oplus v_0, r \oplus v_1\}$, where $r \leftarrow \{0, 1\}^\ell$ is a random chosen value.
   - $R$ acts as an OT receiver with input a choice bit $b_R$.
   - $R$ obtains $w = r \oplus v_{b_R}$.
3. $S$ sends $u = r \oplus b_S \cdot (v_1 \oplus v_0)$ to $R$ who outputs $v = u \oplus w$.

Figure 5: Membership Oblivious Transfer (mOT) Construction

Our mOT Protocol. Our mOT construction consists of two main phases. The first phase follows the popular steps in the circuit-PSI protocols [PSTY19, PSWW18], which enables the sender and the receiver to compute a secret share of a membership bit, i.e. the two parties obtain XOR shares of 1 or 0 if the sender’s keyword $y$ is or is not in the receiver’s set $X$.

The second phase allows the receiver to obtain the corresponding associated value from the sender, depending on whether the output of the first phase was shares of 0 or 1. Typically, this step can be done using generic two-party secure computation (e.g., garbled circuit) in the literature. However, it is relatively inefficient. Instead, we propose a simple solution that relies on OT. More precisely, the sender randomly chooses a value $r \leftarrow \{0, 1\}^\ell$ and masks its associated values by computing $(r \oplus v_0, r \oplus v_1)$. Denote a secret share bit of $S$ and $R$ to be $b_S$ and $b_R$ received from the first phase, respectively. Using the choice bit $b_R$, the receiver obliviously obtains $w = r \oplus v_{b_R}$ when interacting with the sender with input $(r \oplus v_0, r \oplus v_1)$ via OT. Next, the sender sends $u = r \oplus b_S \cdot (v_1 \oplus v_0)$ to the receiver $R$. The value $u$ helps to remove the mask $r$ from the $w$ by computing $v = u \oplus w$, which is the receiver’s output. We formally present the construction of our mOT in Figure 5.

For the correctness of the mOT construction, one can rewrite $w = r \oplus b_R \cdot v_1 \oplus (1 \oplus b_R) \cdot v_0$. 11
Hence, \( v = u \oplus w = (b_R \oplus b_S) \cdot v_1 + (1 \oplus b_R \oplus b_S) \cdot v_0 \) which equals to \( v_{b_R \oplus b_S} \) as desired (recall that \( b_R \oplus b_S = 1 \) if \( y \in X \) and 0 otherwise). We present the security statement of our mOT protocol below.

**Theorem 2.** The mOT protocol described in Figure 5 securely implements the mOT functionality defined in Figure 4 in the semi-honest setting, given the OT and SS-PMT functionalities described in Section 2.

**Proof.** We construct simulators \( \text{Sim}_S \) and \( \text{Sim}_R \) to simulate the view of corrupted sender \( S \) and corrupted receiver \( R \), respectively. We argue the indistinguishability of the simulator and the real execution.

**Simulating \( S \):** The simulator \( \text{Sim}_S \) has input \( (y,v_0,v_1) \) and receives output from the SS-PMT ideal functionality, consisting of a secret-shared membership bit \( b_S \). For the OT execution, the simulator \( \text{Sim}_S \) obtains nothing, except the random OT transcript which is random. Since the output of SS-PMT is secret-shared amongst the corrupt sender and honest receiver, one can replace the bit \( b_S \) with a random. It is straightforward to check that the simulation is perfect.

**Simulating \( R \):** \( \text{Sim}_R \) with input \( X \) receives nothing from the SS-PMT ideal functionality, expect a secret-shared membership bit \( b_R \). \( \text{Sim}_R \) obtains \( w \) from the OT and \( u \) from the sender in the last step. We show that the output of the simulator \( \text{Sim}_R \) is indistinguishable from the real execution. For this, we formally show the simulation by proceeding with the sequence of hybrid transcripts \( T_0, T_1, T_2 \) where \( T_0 \) is real view of the receiver, and \( T_2 \) is the output of \( \text{Sim}_R \).

- Let \( T_1 \) be the same as \( T_0 \), except the SS-PMT output which can be replaced with random as the honest sender holds a secret-shared of the output. Thus, \( T_0 \) and \( T_1 \) are indistinguishable.
- Let \( T_2 \) be the same as \( T_1 \), except the OT execution and obtaining \( u \). Due to the underlying security property of OT, the receiver only learns one of the two strings related to \( v_0 \) or \( v_1 \). In addition, the sender’s associated values were masked with a random value \( r \) before the OT execution. Thus, \( w \) reveals nothing about \( v_i \in \{0,1\} \). When having \( u = r \oplus b_S \cdot (v_1 \oplus v_0) \), the corrupt receiver might try to unmask \( r \) by computing \( u \oplus w \). However, the resulting value is indeed the protocol’s output which can be simulated. Therefore, we can replace both \( w \) and \( u \) with random (the receiver sees a system of two equations that contains three unknown variables). In summary, \( T_2 \) and \( T_1 \) are indistinguishable.

\[ \square \]

### 3.2 Oblivious Shuffle and Decryption (Shuffle&Decrypt)

**Definition 3.** Oblivious Shuffle and Decryption (Shuffle&Decrypt) is a n-party protocol, in which each party \( P_i \in [n] \) holds a permutation \( \pi_i : [m] \rightarrow [m] \) and a secret key \( sk_i \) of the multi-key cryptosystem as \((pk, \{sk_i\}_{i \in [n]}) \leftarrow \text{KeyGen}(1^n, n)\). Given a set of ciphertexts \( \{ct_1, \ldots, ct_m\} \) where \( ct_i = \text{Enc}(pk, x_i) \), except randomnesses, the Shuffle&Decrypt functionality gives \( \{x_{\pi(1)}, \ldots, x_{\pi(m)}\} \) to the party \( P_1 \) where \( \pi = \pi_n \circ \pi_{n-1} \circ \ldots \circ \pi_1 \), and nothing to other parties.

The private permutation aims to remove the linkage between the ciphertext \( ct_i \) and the plaintext \( x_i \). We formally describe the Shuffle&Decrypt ideal functionality in Figure 6.
PARAMETERS: \( n \) parties, parameter \( m \), and a multi-key encryption scheme defined in Section 2.5

FUNCTIONALITY:

- Wait for input secret key \( s_{k_1} \) and a permutation function \( \pi_i : [m] \to [m] \) from each party \( P_{i \in [n]} \). Here, \((pk, \{sk_i\}_{i \in [n]}) \leftarrow \text{KeyGen}(1^*, n)\).
- Wait for a combined input a set of ciphertexts \( \{ct_1, \ldots, ct_m\} \) where \( ct_i = \text{Enc}(pk, x_i) \) 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

Our Shuffle\&Decrypt Protocol. The Shuffle\&Decrypt construction is simple and directly built from calling algorithms provided in the multi-key cryptosystem. First, the \( P_1 \) re-randomizes the ciphertexts and then permutes the result. \( P_1 \) then sends the permuted set \( C_1 \) to \( P_2 \). The re-randomization aims to hide the permutation function from \( P_2 \). The \( P_2 \) now performs partial decryption using its secret key \( s_{k_2} \). This decryption removes the role of \( s_{k_2} \) from the original ciphertext. \( P_2 \) then applies the permutation \( \pi_2 \) on the resulting ciphertexts \( C_2 \) and forwards them to \( P_3 \). Note that \( P_2 \) does not need to rerandomize \( C_2 \) as the \( C_2 \) is in the random distribution and thus it hides the permutation of \( P_2 \). The process repeats sequentially through \( P_4, \ldots, P_n \). After the partial decryption was executed by \( P_n \), the ciphertexts require only the secret key \( s_{k_1} \) for the final decryption. \( P_n \) now sends these ciphertexts in the permuted order to \( P_1 \) which performs the partial decryption and outputs the final result. Figure 7 presents the Shuffle\&Decrypt construction. From the high-level description, it is clear that the protocol is correct given the correctness of the underlying multi-key cryptosystem. We present the security statement of the Shuffle\&Decrypt protocol below.

Theorem 4. Given the multi-key cryptosystem defined in Section 2.5, the Shuffle\&Decrypt protocol described in Figure 7 securely implements the Shuffle\&Decrypt functionality defined in Figure 6 in the semi-honest model, against any number of corrupt, colluding, semi-honest parties.

Proof. Let \( A \) be a coalition of corrupt parties. The view of \( A \) is a set of ciphertexts \( \{C_i \mid P_i \in A\} \), and the output of the Shuffle\&Decrypt which is \( \{x_{\pi_1}, \ldots, x_{\pi_m}\} \) if the leader \( P_1 \in A \).

Thanks to the property of the multi-key cryptosystem, \( C_{i \in [n]} \) reveals nothing about the underlying plaintexts. If \( P_1 \) is honest, the randomization hides the party’s permutation function. Moreover, when assuming \( \{P_i, P_j\} \in A \) but \( \{P_{i+1}, \ldots, P_{j-1}\} \not\in A \), one might think that \( A \) might learn the permutation functions of honest parties \( \{P_{i+1}, \ldots, P_{j-1}\} \). However, the output of the partial decryption gives ciphertexts in the random distribution. Thus, the resulting view is random to \( A \) (i.e., the corrupt coalition’s view is simulated).

4 Our mPSU Construction

Figure 8 presents our main mPSU protocol, which guarantees security against any number of corrupt, colluding, semi-honest parties. The protocol makes use of our new mOT and Shuffle\&Decrypt
4.1 Our Protocol

The design of a secure mPSU protocol presents significant challenges, specifically with regard to (1) ensuring that the output does not contain duplicate items, (2) preventing the disclosure of partial union results, and (3) hiding which items from which parties. To illustrate the high-level idea of our protocol, we consider a simple 4-party case where the leader party \( P_1 \) has a set \( X_1 \) of items while each of the remaining party \( P_i \) for \( i \in [2,4] \) possesses a single item \( X_i = \{ x_i \} \). We assume that the item \( x_2 \) of \( P_2 \) and \( x_3 \) of \( P_3 \) are not in the \( X_1 \) but \( x_4 \) of \( P_4 \) is (i.e. \( x_2 = x_3 \notin X_1 \) and \( x_4 \in X_1 \)).

Regarding (1), a potential approach is to enable the leader \( P_1 \) to engage with the other parties and obtain an encryption of \( x_i \) if \( x_i \notin X_1 \) and an encryption of the zero otherwise. An encryption of zero indicates the presence of common items between \( P_1 \) and \( P_i \), which can be removed after decryption. To this end, the \( P_1 \) and \( P_i \) invoke the mOT instance in which \( P_1 \) acts as the sender with input \( (x_i, Enc(pk,0), Enc(pk,x_i)) \) and \( P_1 \) acts the receiver with input \( X_1 \), thereby obtaining the desired encryption. After executing the mOT instances, the leader party \( P_1 \) acquires \( E = \{ Enc(pk,x_2), Enc(pk,x_3), Enc(pk,0) \} \) from the party \( P_2, P_3 \) and \( P_4 \), respectively. The set \( E \) allows
Table 2: Illustration of our mSU protocol for 4 parties. $P_1$ has an input set of $X_1$ while $P_i$ have input set of only one item $x_i$ for $i \in [2,4]$. In addition, we assume that $x_2 = x_3 \not\in X_1$ and $x_4 \in X_1$. $\leftarrow \Rightarrow$ denotes the execution of protocol $P$ between two parties. Colors indicating the corresponding output for each invocation of protocol. For example, in round 1, $P_1$ and $P_2$ invoke the mOT (Step (3,a) in Figure 8). $P_1$ updates its set $E$ by the received the message $\text{Enc}(x_2)$ from $P_2$. $P_3$ and $P_4$ invoke mOT protocol with $P_2$ concurrently to update their encryption (Step (3,b) in Figure 8). $P_3$ receives an encryption of zero $\text{Enc}(0)$ since $x_3 = x_2$ and $P_4$ receives the same encryption of $x_4$ as $\text{Enc}(x_4)$. The following rounds are similar.

<table>
<thead>
<tr>
<th>Round</th>
<th>Protocol</th>
<th>Inputs</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Round 1</td>
<td>mOT</td>
<td>$P_1 : X_1$, $P_2 : {x_2}$</td>
<td>$P_3 : {x_3}$</td>
</tr>
<tr>
<td>Round 2</td>
<td>mOT</td>
<td>$E = {\text{Enc}(x_2)}$</td>
<td>$P_4 : {x_4}$</td>
</tr>
<tr>
<td>Round 3</td>
<td>mOT</td>
<td>$E = {\text{Enc}(x_2), \text{Enc}(0)}$</td>
<td></td>
</tr>
<tr>
<td>Shuffle</td>
<td>$\text{Enc}(0), \text{Enc}(x_4), \text{Enc}(x_2)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decrypt</td>
<td>${0, x_4, x_2}$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Output</td>
<td>$X_1 \cup {x_4, x_2}$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The above protocol description does not entirely address the issue of removing duplicate items since $x_2$ could be identical to $x_3$. This causes the challenge (2) as mentioned above. To overcome the limitation, we leverage the mOT in the following manner.

Similar as before, the protocol starts with the leader party $P_1$ which has an input $X_1$ and learns $\text{Enc}(\text{pk}, x_2)$ after executing an mOT with $P_2$. Next, $P_2$ performs an mOT with each of the parties $P_{i\in[3,4]}$. Specifically, $P_{i>2}$ acts as the sender with inputs $(x_i, \text{Enc}(\text{pk}, 0), \text{Enc}(\text{pk}, x_i))$, while $P_2$ acts as the receiver with input $X_2$. As the result, the $P_2$ learns $e_2^i$, where $e_2^i$ is the encryption of zero if $x_i \in X_2$ and the encryption of $x_i$ otherwise. The obtained encryption $e_2^i$ is then randomized by $P_2$ before being sent back to $P_i$, which is used as the input to the next mOT between $P_1$ and $P_i$.

Using the above mOT, we can remove the intersection items between $X_2$ and $X_{i>2}$. Therefore, when $P_1$ with input $X_1$ and $P_3$ with input $(x_3, \text{Enc}(\text{pk}, 0), e_3^2)$ execute an mOT, the receiver $P_1$ does not obtains the encryption of the items $x_3$ that is in the intersection $X_2 \cap X_3$. Concretely, $e_3^2 = \text{Enc}(\text{pk}, 0)$ as $x_3 \in X_2$.

At this point, if the $P_4$ repeats the previous step performed by $P_3$ using the encryption $e_2^2$ as the input to mOT with the receiver $P_1$, there is a risk that the $P_1$ might obtain the encryption of the same item twice if $x_3 = x_4$. Therefore, it is necessary for $P_4$ executes an mOT with $P_3$ to remove the intersection between $X_3$ and $X_4$.

In summary, our protocol can be viewed as a sequence of mOT instances between $P_1$ and $P_i$, where $P_1$ has input $X_1$ while $P_i$ inputs is $(x_i, \text{Enc}(\text{pk}, 0), e_{i-1}^i)$, $\forall i \in X_i$. Here $e_{i-1}^i$ is the result obtained from a sequence of mOT executions between between $P_i$ and each $P_{\ell \in [2:i-1]}$. This process helps to eliminate duplications between $X_1$ and $X_i$.

Upon the completion of $(n - 1)$ instances of the mOT protocol between the leader party $P_1$ and other parties $P_i$, the leader $P_1$ has acquired an encryption set $E$, containing encryptions $\text{Enc}(\text{pk}, x)$
for $x \in \bigcup_{i=2}^{n} X_i$ and $\tau$ encryptions of zero, where $\tau = \sum_{i=1}^{n} |X_i| - |\bigcup_{i=1}^{n} X_i|$ indicates the number of duplicate items. In order to satisfy requirement (3) of the mPSU protocol, we employ the Shuffle&Decrypt functionality, which permits each party to apply its own permutation function on the encryption set $E$.

We demonstrate our mPSU protocol execution in Table 2 pertaining to the 4-party scenario described above. Our protocol maintains security even if some parties collude. For example, the adversary cannot determine the numbers of zero encryption before Shuffle&Decrypt execution due to the IND-CPA security provided by the multi-key encryption.

At this stage, we are currently focusing on a simple scenario where each $P_{i \in [2,n]}$ possesses only one item. In order to generalize our method to a set $X_i$, we apply a popular technique known as the bin-and-ball technique. At the high level, the party $P_{i \in [2,n]}$ places its input values into $\beta$ bins through the use of Cuckoo hashing, where each bin is allowed to contain at most one item. The leader $P_1$ utilizes the same set of Cuckoo hash functions to map the input values in $S$ into $\beta$ bins using Simple hashing. The mapping allows the parties to execute the simple case above bin-by-bin efficiently. As a result, for each bin, the $P_1$ obtains encryptions of the partial union set which are subsequently combined into a big encryption set $E$ before being subjected to decryption.

### 4.2 Correctness and Security

**Correctness.** We consider three following cases depending on whether a specific item $x_{i,k} \in X_i$ of the smallest-index party $P_1$ is in $P_1$ or other parties $P_t$ for $n \geq t > i > 1$. Since $P_1$ is the smallest index that has $x_{i,k}$, no previous parties have $x_{i,k}$. Thus, $P_1$ obtains $e_{b,i-1}^i = \text{Enc}(pk, x_{i,k})$ after interacting with $P_{i \in [2,i-1]}$ via a sequence of the mOT instances.

- **Case 1** ($x_{i,k} \in X_1$) – the $P_1$ has $x_{i,k}$: As $x_{i,k} \in X_1$, the mOT with $P_1$ in Step (3,a) gives $P_1$ the encryption of zero $\text{Enc}(pk, 0)$. As a result, $x_{i,k}$ does not appear in the final result from the Shuffle&Decrypt execution.
- **Case 2** ($x_{i,k} \notin X_1$ and $x_{i,k} \in X_t$) – the $P_1$ does not have $x_{i,k}$, but another party $P_t$ has $x_{t,j} = x_{i,k}$ for $t > i$: The mOT execution between $P_1$ and $P_t$ in Step (3,b), on input including $e_{b,i-1}^i$, provides $P_t$ with the encryption of zero. This encryption is then rerandomized before being sent back to $P_t$. In other words, $P_t$ obtains $e_{b,i}^t = \text{Enc}(pk, 0)$ after Step (3,b). Thus, when executing with $P_1$ in the following round, $P_1$ obtains $\text{Enc}(pk, 0)$, ensuring that the $x_{i,k}$ will appear in the final union output.
- **Case 3** ($x_{i,k} \notin \bigcup_{j=1}^{n} X_j$) – no party has $x_{i,k}$: The mOT executions between $P_i$ and $P_t$ in Step (3) maintain the encryption of $\text{Enc}(pk, x_{i,k})$. Thus, $P_1$ then obtains an $\text{Enc}(pk, x_{i,k})$ from $P_t$ in Step (3,a). Consequently, $x_{i,k}$ appears in the final result from the Shuffle&Decrypt functionality.

**Security.** The security of our mPSU protocol is given as below. At a high level, the multi-key encryption system is secure because all the information viewed by a malicious adversary remains encrypted when the mOT is secure.

**Theorem 5.** Given the multi-key cryptosystem, mOT and Shuffle&Decrypt functionalities described in Section 2.5, Figure 4, and Figure 6, respectively, the mPSU protocol described in Figure 8 securely implements the mPSU functionality defined in Figure 1 in the semi-honest model, against any number of corrupt, colluding, semi-honest parties.
PARAMETERS:

- \( n \) parties \( P_i \in [n] \) for \( n > 1 \).
- The mOT and Shuffle&Decrypt functionalities described in Figure 4 and Figure 6, respectively.
- A multi-key cryptosystem \((\text{KeyGen}, \text{Enc}, \text{ParDec}, \text{FulDec}, \text{ReRand})\) defined in Section 2.5.
- Hashing parameters: a number of bins \( \mu \), maximum bin sizes \( \beta : \mathbb{Z} \rightarrow \mathbb{Z} \) for simple-hashing bins, the \( h \) hash functions \( H_j \in [h] : \{0, 1\}^* \rightarrow [\mu] \).

INPUT:

- Party \( P_i \in [n] \) has \( X_i = \{x_{i,1}, \ldots, x_{i,m}\} \).

PROTOCOL:

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

2. Local Execution:
   
   (a) \( P_i \in [2, n] \) hashes items \( X_i \) into \( \mu \) bins using the Cuckoo hashing. Let \( C^i_b \) denote the items in the \( P_i \)'s \( b \)th bin. \( P_i \) computes the encryption \( e^i_{b,1} = \text{Enc}(pk, C^i_b) \), for \( b \in [\mu] \).
   
   (b) \( P_i \in [n] \) hashes \( X_i \) into \( \mu \) bins under \( k \) hash functions. Let \( S^i_b \) denote the set of items in the \( P_i \)'s \( b \)th bin. \( P_i \) pads \( S^i_b \) with dummy values to the maximum bin size \( \beta \).
   
   (c) For bin \( b \in [\mu] \), the \( P_1 \) initially an empty set \( E_b \).

3. \( P_1 \) sequentially interacts with \( P_i \) for \( i \in [2, n] \) as follow.
   
   (a) For each bin \( b \in [\mu] \), \( P_1 \) and \( P_i \) invoke a mOT instance where:

   - \( P_1 \) acts as the receiver with input \( S^i_b \).
   - \( P_i \) acts as the sender with input \( (C^i_b, \text{Enc}(pk, 0), \bar{e}_b) \). Here, \( \bar{e}_b = \text{Enc}(pk, 0) \) if \( C^i_b = \emptyset \), otherwise, \( \bar{e}_b = e^i_{b,i-1} \).
   - \( P_1 \) obtains \( e_b \).
   - \( P_1 \) appends \( e_b \) to \( E_b \).

   (b) For each bin \( b \in [\mu] \), the \( P_i \) and \( P_{i+1} \in [i+1, n] \) invoke a mOT instance where:

   - \( P_i \) acts as the receiver with input \( S^i_b \).
   - \( P_t \) acts as the sender with input \( (C^t_b, \text{Enc}(pk, 0), e^t_{b,i-1}) \).
   - \( P_t \) obtains \( c \) and sends \( c' = \text{ReRand}(c, pk) \) to \( P_t \).
   - \( P_t \) computes \( e^t_{b,i} = \text{ReRand}(c', pk) \).

4. All the parties invoke the Shuffle&Decrypt functionality where:

   - \( P_1 \) inputs \( E = \bigcup_{b=2}^{\mu} E_b \), the \( sk_1 \) and a random permutation \( \pi_1 : [m] \rightarrow [m] \).
   - \( P_i \) inputs the private key \( sk_i \) and a random permutation \( \pi_i : [m] \rightarrow [m] \).
   - \( P_1 \) obtains a set \( U \).

5. \( P_1 \) removes all zero from \( U \), and outputs \( \bigcup \mathcal{H} \cup X_1 \).

Figure 8: Our mPSU Protocol in the Dishonest Majority Setting
Proof. Let $C$ and $H$ be a coalition of corrupt and honest parties, respectively. We must show how to simulate $C$'s view in the ideal model. We consider three following cases based on whether $C$ has an item $x$:

1. $C$ does not have $x$, but $H$ has $x$: We consider two cases. First, if $H$ contains only one honest party $P_1$, then $P_1$ has $x$. Consider the case where $P_i$ is $P_1$. During the protocol execution, $P_1$ only acts as the receiver via $\text{mOT}$ in Step (3,a) and participates in the shuffle and decryption in Step (4). Assuming that these two building blocks are secure, $P_1$ does not reveal anything to $C$. If $i \neq 1$, the corrupted parties $C$ should contain the leader $P_1$, thus, they can deduce that the honest party $P_i$ has $x$ from the output of the set union. Hence, there is nothing to hide about whether $P_i$ has $x$ in this case.

Second, if $H$ has more than one honest party (say $P_i$ and $P_{j>i}$). We consider two following subcases:

- Only $P_i$ has $x$: we must show that the protocol must hide the identity of $P_i$. If $P_1 \in H$, only the honest party $P_1$ learns the union $\bigcup_{i=1}^n X_i$ in Step 5. In addition, the $\text{mOT}$ between $P_i$ and previous corrupt parties $P_t \in C$ reveals nothing to $C$ as the obtained output is under the multi-key encryption and rerandomized before the next execution. Thus, the simulation is simple.

  If $P_1 \in C$, the corrupt $P_1$ obtains $\text{Enc}(pk, x)$ from $P_i$. Since the encryption is protected under the $\text{Shuffle\&Decrypt}$ functionality until the $P_1$ learns the union sets which was permuted by the honest party $P_1$, the encryption reveals nothing to $C$.

- Both $P_i$ and $P_j$ have $x$: If $i = 1$, then the honest leader $P_i$ receives encryptions of zeros $\text{Enc}(pk, 0)$ when executing $\text{mOT}$ with $P_j$. If another party $P_{t>j}$ possesses $x$, the $\text{mOT}$ execution between $P_j$ and $P_t$ results in $P_i$ receiving the $\text{Enc}(pk, 0)$, while $P_t$ learns nothing. Thus, when doing the permutation in Step 5, the $C$ learns nothing about which parties in $H$ have $x$. If $P_1 \in C$, the corrupt $P_1$ receives the encryption $\text{Enc}(pk, x)$ from $P_i$ and $\text{Enc}(pk, 0)$ from $P_j$. Similarly, thanks to the CCA property of the encryption scheme and the permutation in $\text{Shuffle\&Decrypt}$, $C$ cannot distinguish the two encryptions. Thus, the protocol hides the identity of which honest party has $x$.

2. $C$ have $x$, but $H$ does not have $x$: We must show that the protocol must hide the information that $H$ does not have $x$. Consider the $\text{mOT}$ execution where a party in $H$ acts as the sender and a party in $C$ acts as the receiver, the corrupt set $C$ receives $\text{Enc}(pk, x)$ which is rerandomized by $H$. Thus, $C$ learns nothing. In the final step, the encryption set $E$ contains $\text{Enc}(pk, x)$, which was permuted by the honest parties $H$. Hence, all honest parties have an indistinguishable effect on the $\text{Shuffle\&Decrypt}$ step.

3. Both $C$ and $H$ have $x$. When $C$ acts as the receiver and invokes the $\text{mOT}$ with an honest sender, if the sender’s keyword $x$ is not in the receiver’s set, the receiver obtains the encryption of the keyword $x$. Otherwise, the receiver obtains the encryption of zero. The $C$ cannot differentiate between the two cases. When $C$ acts as the sender in $\text{mOT}$, $C$ obtains nothing but might receive the rerandomization of the output from the receiver in Step (3,b). Since the message was rerandomized by $H$, $C$ cannot infer the underlying encryption. Moreover, $\text{Enc}(pk, x)$ appears only once in the encryption set $E$, so the $C$ learns nothing about whether the $H$ has $x$.
4.3 Complexity

<table>
<thead>
<tr>
<th>Our Protocol</th>
<th>( m = 2^3 )</th>
<th>( m = 2^{12} )</th>
<th>( m = 2^{20} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( t = 1 )</td>
<td>11.10</td>
<td>187.23</td>
<td>400.50</td>
</tr>
<tr>
<td>( t = 4 )</td>
<td>4.68</td>
<td>24.99</td>
<td>127.95</td>
</tr>
<tr>
<td>( t = 16 )</td>
<td>31.20</td>
<td>60.95</td>
<td>204.32</td>
</tr>
</tbody>
</table>

LAN (s)  
- \( n = 3 \): 4.35  
- \( n = 4 \): 18.56  
- \( n = 6 \): 25.69

WAN1 (s)  
- \( n = 3 \): 5.05  
- \( n = 4 \): 8.38  
- \( n = 6 \): 15.54

WAN2 (s)  
- \( n = 3 \): 9.49  
- \( n = 4 \): 15.01  
- \( n = 6 \): 38.72

Table 3: The running time and communication cost of our mPSU protocol: the number of parties \( n \in \{3, 4, 6, 8\} \), set size \( m \in \{2^8, 2^{12}, 2^{16}, 2^{20}\} \), and numbers of thread \( t \in \{1, 4, 16\} \). The reported running time represents the time taken for the entire protocol to complete. Communication cost is computed as the average cost across all parties.

<table>
<thead>
<tr>
<th>( m = 2^3 )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAN</td>
<td>WAN1</td>
<td>LAN</td>
<td>WAN1</td>
</tr>
<tr>
<td>Total</td>
<td>2.18</td>
<td>1.98</td>
<td>30.88</td>
</tr>
<tr>
<td>mDiff</td>
<td>2.11</td>
<td>1.96</td>
<td>29.99</td>
</tr>
<tr>
<td>Shuffle&amp;Decrypt</td>
<td>0.08</td>
<td>0.02</td>
<td>13.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( m = 2^{12} )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAN</td>
<td>WAN1</td>
<td>LAN</td>
<td>WAN1</td>
</tr>
<tr>
<td>Total</td>
<td>43.28</td>
<td>40.83</td>
<td>568.27</td>
</tr>
<tr>
<td>mDiff</td>
<td>28.99</td>
<td>11.91</td>
<td>30.98</td>
</tr>
<tr>
<td>Shuffle&amp;Decrypt</td>
<td>1.19</td>
<td>14.05</td>
<td>19.32</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>( m = 2^{20} )</th>
<th>( P_1 )</th>
<th>( P_2 )</th>
<th>( P_3 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAN</td>
<td>WAN1</td>
<td>LAN</td>
<td>WAN1</td>
</tr>
<tr>
<td>Total</td>
<td>390.25</td>
<td>303.31</td>
<td>309.82</td>
</tr>
<tr>
<td>mDiff</td>
<td>289.28</td>
<td>303.31</td>
<td>309.82</td>
</tr>
<tr>
<td>Shuffle&amp;Decrypt</td>
<td>19.04</td>
<td>303.31</td>
<td>309.82</td>
</tr>
</tbody>
</table>

Table 4: The breakdown running time and communication cost for each party in our mPSU protocol \( (n = 4) \).

We presented the communication, computation, and round complexities of our mPSU protocol in Figure 1 and elaborate on them here. It is clear that our protocol has \( n \) rounds for both Step (3) and Step (4). Leveraging the bin-and-ball technique introduced in \([PSSZ15, PSZ18]\), parties hash elements into Cuckoo and Simple hashing tables consisting of \( O(m) \) bins. Each bin of the Simple hashing table accommodates up to \( O(\log m / \log m) \) elements. In round \( i \) – 1, party \( P_i \) engages in mOT with \( P_{i+1} \) and \( P_{2i} \), each incurring a cost of \( O(\log m / \log m) \) in terms of communication and computation per bin. This yields a total cost of \( O(n^2 \log m / \log m) \).

Remark. In our protocol, the non-linearity with respect to \( m \) comes from the mOT, specifically the SS-PMT. We discuss the construction in Appendix A. \([LG23, DCZB24]\) proposed SS-PMT protocol with linear complexity. By incorporating these new constructions into our mOT, our mPSU framework can achieve linear complexity in terms of the number of parties and set size.

19
5 Implementation and Performance

We implement our protocol and evaluate it with various number of parties, set sizes, and number of threads. All evaluations were performed with an item input length of 128 bits, a statistical security parameter $\lambda = 40$, and a computational security parameter $\kappa = 128$. We do a number of experiments on a single server that has AMD EPYC 74F3 processors and 256GB of RAM. We run all parties in the same network, but simulate a network connection using the Linux tc command: a LAN setting with 0.02ms round-trip latency, 10 Gbps network bandwidth; two WAN settings: WAN$_1$ with 10ms and WAN$_2$ with 80ms round-trip latency, and both have 400 Mbps network bandwidth.

Our mPSU protocol is built on ElGamal encryption scheme (multi-key cryptosystem), SS-PMT, and OT (mOT). We implement the multi-key ElGamal encryption scheme using the elliptic curve code (Curve25519) from Relic [REL]. For the SS-PMT implementation which requires garbled circuit for two strings comparison, we use the EMP-toolkit library [WMK16]. Finally, we use the OT-extension [IKNP03] provided in [PR] to implement mOT. Our complete implementation is available on GitHub.

Our protocol scales well using multi-threading between the parties. In each round, the party $P_i \in [2, n-1]$ can use $n-i+1$ threads so that each party operates mOT building block with other parties $P_1$ and $P_j \in [i+1, n]$ at the same time. In addition, each pair of parties can use multiple threads to execute these building blocks bin-by-bin in parallel. We evaluate it on the number of threads $t \in \{1, 4, 16\}$ to show the performance of our protocols running with multi-threading.

5.1 Performance of Our mPSU Protocol

Table 3 presents the overall runtime and communication overhead of our mPSU protocol. From the empirical numbers, we can see that the performance difference between WAN$_1$, WAN$_2$, and LAN is primarily due to the latency for the smaller input. The gap increases with the number of parties which is also observed in other protocols with an $O(n)$ or higher round complexity.

Additionally, we present the breakdown cost of our protocol for each party in 4-party scenarios with varying set sizes in Table 4. Specifically, we present the performance metrics of the mOT in Step (3) and the Shuffle&Decrypt in Step (4) in our protocol in terms of running time and communication cost. All reported running time values in Table 3 and Table 4 represent end-to-end time.

5.2 Comparison with Previous Work

To demonstrate the performance of our mPSU protocols with a comparison, we have implemented the semi-honest protocols proposed in [Fri07, BA12] and estimate the performance for protocol proposed in [GHJ22]. Table 5 and Figure 9 present the running time and communication cost of various mPSU protocols [Fri07, BA12, GHJ22] which are secure in the dishonest majority and semi-honest setting. We do not incorporate the results from [VCE22] into our comparison, as their protocol only works for a small universe. Even in their largest setting, with a universe size of $2^{32}$, it is considerably smaller than the general scenario involving 128-bit elements. According to [VCE22, Figure 7], in a scenario involving 5 parties, each with only 32 elements of 32-bit length,

Footnote:

1The recent mPSU protocol [LG23] provides a weak security guarantee wherein the leader does not collude with any parties.
their protocol takes around 10 seconds. Interestingly, this is comparable to the runtime of our protocol involving 6 parties, each with 256 elements in a 128-bit universe.

In [BA12], each input set \(X_i\) is initially shared among \(n\) parties using a secret-sharing scheme. Subsequently, these parties employ a generic secure computation technique to compute the union on the shares. Our implementation of the [BA12]’s method, however, is limited to the two-party scenario where each \(X_i\) is secret-shared between only two parties (which is in favor of [BA12]). Consequently, the secure computation takes place exclusively between these two parties. We implement [BA12] using EMP-toolkit library [WMK16] which provides most of the state-of-the-art techniques for two-party secure computation in the semi-honest setting. As shown in Table 5, for \(n = 4\), our protocol is \(1.44 - 3.22\) times faster in the LAN setting and \(8.50 - 80.84\times\) faster than [BA12] in the WAN2. Additionally, the cost for [BA12] is significantly \((168.33 - 405.73\times)\) higher than our protocol for set size \(m\in \{2^8, 2^{12}, 2^{16}, 2^{20}\}\).

We report the partial running time and communication cost of the mPSU protocol proposed by [GHJ22]. The first step of their protocol is for each party to locally compute an encryption of a local Bloom filter. To achieve a false positive rate of \(2^{-40}\), the table size should be at least \(60nm\). We estimate the time and communication cost for this single step of each party based on the performance shown in [MHL21] (as well as our [Fri07]’s implementation), where each Paillier encryption takes about 2.5 ms with a key length of 2048 bits, and report the numbers in Table 5.

Our mPSU protocol outperforms previous works in the LAN setting. Despite the low communication cost due to the usage of homomorphic encryption, the running time of [Fri07, GHJ22] is not practical even for small set sizes. Thus, we skip the evaluation of the [GHJ22, Fri07] in the WAN setting. We also show the concrete number for each protocol in Table 5. [GHJ22] has a constant round of 7 and [BA12] has a round complexity of \(O(\log(nm))\) sensitive to \(m\). Both [Fri07] and our protocol gives \(O(n)\) rounds independent of input size. We believe that our protocol provides the best trade-off between the running time, bandwidth cost, and round complexity.

6 Conclusion

In this work, we propose an efficient mPSU protocol in the semi-honest setting against an adversary that colludes an arbitrary number of participants. Our protocol is built on mOT which we believed of independent interests. Our protocol significantly outperforms prior mPSU works in the same security setting in terms of running time and communication cost. Our mPSU framework is the generalization of the well-studied 2-party PSU protocols to the multi-party setting. We highlight some directions for future work:

- Improving scalability: Unlike the 2-party PSU and some other efficient private set intersection protocols, our protocol still heavily relies on public key techniques which is the bottleneck of the performance. It is possible to use symmetric-key techniques such as secret-sharing to replace the encryption scheme and implement Shuffle&Decrypt. We leave the mPSU protocol constructed mainly on the symmetric key techniques as the future work.

- This study concentrates on semi-honest mPSU, which we consider a preliminary stage in advancing towards efficient malicious MPSU. To the best of our knowledge, there is no existing tailored protocols for malicious PSU in both two-party and multi-party settings. To achieve malicious PSU, one can employ cryptographic commitment techniques at each step of the protocol, albeit with added costs.
Figure 9: Multi-party PSU protocols with set size of $2^{20}$ among 4 parties on the network 10Gbps with 0.02ms latency.
<table>
<thead>
<tr>
<th></th>
<th>m</th>
<th>Ours</th>
<th>[GHJ22]</th>
<th>[BA12]</th>
<th>[Fri07]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Running Time</td>
<td>$2^8$</td>
<td>1.88</td>
<td>155.63</td>
<td>2.70</td>
<td>6009.00</td>
</tr>
<tr>
<td>LAN (second)</td>
<td>$2^{12}$</td>
<td>27.96</td>
<td>2490.04</td>
<td>57.73</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>$2^{16}$</td>
<td>490.53</td>
<td>39840.65</td>
<td>1158.53</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>$2^{20}$</td>
<td>7841.32</td>
<td>637450.40</td>
<td>25279.39</td>
<td>-</td>
</tr>
<tr>
<td>Running Time</td>
<td>$2^8$</td>
<td>15.07</td>
<td>-</td>
<td>128.05</td>
<td>-</td>
</tr>
<tr>
<td>WAN2 (second)</td>
<td>$2^{12}$</td>
<td>50.30</td>
<td>-</td>
<td>2387.70</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>$2^{16}$</td>
<td>568.27</td>
<td>-</td>
<td>45939.46</td>
<td>-</td>
</tr>
<tr>
<td>Comm. Cost</td>
<td>$2^8$</td>
<td>8.80</td>
<td>15.72</td>
<td>1481.34</td>
<td>4.25</td>
</tr>
<tr>
<td>(MB)</td>
<td>$2^{12}$</td>
<td>121.92</td>
<td>251.65</td>
<td>30116.50</td>
<td>68.00</td>
</tr>
<tr>
<td></td>
<td>$2^{16}$</td>
<td>1934.76</td>
<td>4026.53</td>
<td>617047.00</td>
<td>1088.00</td>
</tr>
<tr>
<td></td>
<td>$2^{20}$</td>
<td>30956.16</td>
<td>64424.48</td>
<td>12559743.00</td>
<td>17408.00</td>
</tr>
<tr>
<td>Number of Rounds</td>
<td>$2^8$</td>
<td>17</td>
<td>7</td>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>$2^{12}$</td>
<td>14</td>
<td>14</td>
<td>18</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Performance comparison of different mPSU protocols with $n = 4$ parties, each having $m \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$. The communication cost is computed as the overall cost across all parties. Concrete number of round is given here. The numbers for [BA12] are lower-bounds based on complexity $O(\log(nm))$ in Table 1.

Acknowledgement

The authors were partially supported by NSF awards #2101052, #2115075, and ARPA-H SP4701-23-C-0074. We are grateful to the PoPETs 2024 anonymous reviewers for their invaluable feedback. Dong et al. [DCZB24] identified a security flaw in our original protocol against arbitrary collusion through a valid attack. We appredicate them for pointing out a bug in the original version.

References


[MHL21] Huanyu Ma, Shuai Han, and Hao Lei. Optimized paillier’s cryptosystem with fast encryption and decryption. In Annual Computer Security Applications Conference,


A Secret-shared Private Membership Test

In this section, we present the secret-shared private membership test (SS-PMT) construction \cite{GPR21} used in our \texttt{mPSU} implementation. The functionality of SS-PMT is given in Figure 3. SS-PMT is a two-party where $P_0$ have a set $X = \{x_1, \ldots, x_n\} \subset \{0,1\}^*$ and $P_1$ have an item $y \in \{0,1\}^*$. As the result, each $P_i \in \{0,1\}$ learns a bit $b_i$ so that $b_0 \oplus b_1 = 1$ if $y \in X$ and $b_0 \oplus b_1 = 0$ otherwise. The security requirement of SS-PMT is that the participants only learn the desired output and nothing else.

The SS-PMT protocol can be built from the usage of oblivious key-value pair (OKVS) \cite{GPR21} with the help of generic equality test techniques such as garble circuit and secret sharing. A Key Value Store (KVS) consists of two algorithms: i) \texttt{Encode} takes as input a set of $(k, v_i)$ key-value pairs from the key-value domain, $K \times V$, and outputs an object $S$ (or, with negligible probability, an error indicator $\perp$); ii) \texttt{Decode} takes as input an object $S$, a key $x$ and outputs a value $y$. A KVS is correct if, for all $A \subseteq K \times V$ with distinct keys: i) $Pr[\texttt{Encode}(A) = \perp]$ is negligible, and ii) if $\texttt{Encode}(A) = S \neq \perp$ and $(k, v) \in A$ then $\texttt{Decode}(S, k) = v$. We say that a KVS is oblivious, if the values $v_i$ are chosen uniformly then the output of $\texttt{Encode}$ hides the choice of the keys $k_i$. We refer to the \cite{GPR21} for more details about the security and other properties of OKVS.

The SS-PMT protocol is given in Figure 11. For each instance of SS-PMT execution, $P_0$ randomly chooses a secret value $s$ and encodes an OKVS structure with key-value pairs $\{(x_1, s), \ldots, (x_n, s)\}$. $P_0$ send this OKVS structure to $P_1$ and $P_1$ learns a decoded value $s'$ with input $x_1$. Then
$P_0$ and $P_1$ invoke a two-party computation protocol for equality check, which functionality is described in Figure 10. The protocol of equality check can be constructed by garbled circuit (GC) [Yao86, GMW87] with several optimizations such as point-and-permute [BNP08], Free-XOR [KS08], the half-gate [ZRE15], and fixed-key AES garbling optimizations [BHKR13].

To enable multi-point query which is desirable in our mPSU protocol. Naively, for $P_0$ with set $X = \{x_1, \ldots, x_m\}$ and $P_1$ with set $Y = \{y_1, \ldots, y_m\}$, we would like to learn the share of a sequence of bits $b_{1,0}, \ldots, b_{m,0}$ and $b_{1,1}, \ldots, b_{m,1}$ for $P_0$ and $P_1$ such that $b_{i,0} \oplus b_{i,1} = 1$ if $x_i \in Y$ and 0 otherwise. We execute the SS-PMT protocol following the methods in [PSTY19, GMR+21]. Instead of invoking $m$ times SS-PMT protocol between $P_0$ with $x_i \in X$ for $i \in [1, m]$ and $P_1$ with $Y$, we pre-process the set of with cuckoo and simple hashing scheme described in Section 2.4. Then the SS-PMT protocol is performed for each bin of the hash table. Very recently, [LG23] proposed a multi-query SS-PMT protocol. The performance of our mPSU protocol can be improved by using their SS-PMT protocol. For easy of implementation, we opt for the SS-PMT construction described above.

**Parameters:** The Boolean circuit $C$ computing the equality for two strings from $\{0,1\}^*$, with $I_1, I_2$ inputs and $O_1, O_2$ outputs associated with $P_1$ and $P_2$ resp.

**Functionality:**
- Wait for input $x_0 \in \{0,1\}^*$ from $P_0$ and $x_1 \in \{0,1\}^*$ from $P_1$.
- Give $b_0$ and $b_1$ to $P_0$ and $P_1$ respectively such that $b_0 \oplus b_1 = 1$ if $x_0 = x_1$ and 0 otherwise.

**Figure 10:** Secure Two-Party Computation for Equality Check

**Parameters:**
- $P_0$ and $P_1$, the receiver set size $n$, the PRF $F$

**Input:**
- $P_0$: $X = \{x_1, \ldots, x_m\} \subset (\{0,1\})^n$ and the PRF key $k$
- $P_1$: $y \in \{0,1\}$

**Protocol:**
1. $P_0$ chooses a random value $s \leftarrow \{0,1\}^*$ and constructs an OKVS $S \leftarrow \text{Encode}(\{(x_1,s), \ldots, (x_n,s)\})$ and sends $S$ to $P_1$.
2. The sender decode compute the value $s' = \text{Decode}(S, x)$.
3. $P_0$ and $P_1$ invoke the protocol of secure two-party equality check protocol (Functionality 10) with input $s$ and $s'$ and learns the bit $b_0$ and $b_1$.

**Figure 11:** Secret-shared Private Membership Test Protocol [GPR+21, PSTY19]

## B Our Original mPSU Protocol With Non-Collusion Assumption

In this section, we present our original mPSU protocol, which is built on mOT, Shuffle&Decrypt, and conditional OPRF (cOPRF). This protocol is secure when a subset of parties do not collude,
unlike our main protocol discussed in Section 4, which is secure in the dishonest majority setting.

We briefly introduce the cOPRF functionality and construction, which is not utilized in the main protocol (Figure 8). Next, we describe the security issue identified by Dong et al. [DCZB24]. Finally, we explain how to address this issue, starting from the original protocol.

**B.1 Conditional OPRF (cOPRF)**

An OPRF [FIPR05] enables the receiver to learn a PRF value on its input query \( q \) without knowing the sender’s PRF key \( k \). We should the functionalty of OPRF in Figure 12. In this section, we introduce a new notion of a conditional oblivious PRF (cOPRF). Intuitively, the functionality is similar to OPRF, with the additional feature that the sender has a set of elements \( X \), and the receiver obtains a designated PRF value depending on whether its query \( q \) is within the sender’s set \( X \).

We note that cOPRF is not required by the current mPSU protocol in Figure 8. However, considering its potential independent interest, we provide a complete description in this section.

**Definition 6.** Conditional OPRF (cOPRF) is a two-party protocol, in which a sender \( S \) has a PRF key \( k \in \{0, 1\}^\kappa \) and an associated set \( X = \{x_1, \ldots, x_m\} \in (\{0, 1\}^\ast)^m \), and the receiver learns \( \bar{F}(k, q||b) \) where \( b = 0 \) if \( q \in X \) and \( b = 1 \) otherwise. Here, \( \bar{F} \) is a PRF, and \( q \) is a query input chosen by the receiver.

From Definition 6, we see that a cOPRF remains secure when the underlying PRF function \( \bar{F} \) reveals nothing about the query, akin to the traditional OPRF. Moreover, the receiver learns nothing about the sender’s input from the cOPRF output, even when sending the same query or multiple queries. The formal description of a conditional oblivious PRF (cOPRF) functionality is given in Figure 13. The primary security goal of cOPRF is to enable the receiver to acquire a designated PRF value according to a defined condition. These PRF values can then be employed in the following computation phase tailored to that condition, as seen in our mPSU protocol.

**Parameters:** A PRF \( F \), and a bound \( m \) on the number of queries.

**Functionality:**

- Wait for input \( (q_1, \ldots, q_m) \) from the receiver where \( q_i \in \{0, 1\}^\kappa \).
- Sample a random PRF key \( k \) and give it to the sender.
- Give \( \{F(k, q_1), \ldots, F(k, q_m)\} \) to the receiver.

**Figure 12: OPRF Ideal Functionality**

**Our cOPRF Protocol.** We present the construction of an cOPRF, which is built on our mOT primitive. While many OPRF protocols exist such as the BaRK OPRF [KKRT16], we use the Diffie-Hellman OPRF protocol [DH76] in which the PRF value of \( x \) has a form \( H(x||0)^k \) for a random hash function \( H : \{0, 1\}^\ast \rightarrow \{0, 1\}^\kappa \).

The protocol starts with the receiver \( \mathcal{R} \) picking a random number \( \alpha \leftarrow \{0, 1\}^\kappa \) and computing \( v_i = H(q||i)^\alpha \) for \( i \in \{0, 1\} \). The goal is to allow the receiver \( \mathcal{R} \) obliviously send \( v_i \) to the sender \( S \) depending on whether \( q \in X \). This can be done using the mOT in which the receiver \( \mathcal{R} \) acts as
PARAMETERS: Sender $S$ and Receiver $R$, the receiver set size $m$, and the PRF $F$.

FUNCTIONALITY:

- Wait for input set $X = \{x_1, \ldots, x_m\}$, a PRF key $k$.
- Wait for input a query $q$ from $R$.
- Give $R$ the PRF value $\overline{F}(k, q||i)$ where $i = 0$ if $q \notin X$, and $i = 1$ otherwise.

Figure 13: Conditional OPRF (cOPRF) Ideal Functionality

PARAMETERS:

- Sender $S$ and Receiver $R$, the receiver set size $m$, the PRF $F$
- The $m$OT functionalities described in Figure 4
- The hash function $H : \{0,1\}^\ast \rightarrow \{0,1\}^\kappa$

INPUT:

- Sender $S$: $X = \{x_1, \ldots, x_m\} \subseteq (\{0,1\}^\ast)^m$ and the PRF key $k$
- Receiver $R$: $q \in \{0,1\}^\ast$

PROTOCOL:

1. The receiver chooses a random $\alpha \leftarrow Z$ and computes $v_i = H(q||i)^\alpha$ for $i \in \{0,1\}$.
2. The sender $S$ and the receiver $R$ invoke a $m$OT functionality where:
   - $R$ acts as the sender with input $(q, v_0, v_1)$.
   - $S$ acts as the receiver with input $X$, and obtains $v$.
3. The sender computes $w = v^k$ and sends it to the receiver who outputs $w^{1/\alpha}$.

Figure 14: Conditional OPRF (cOPRF) Construction

the $m$OT’s sender with input $(q, v_0, v_1)$ while the sender $S$ acts as the $m$OT’s receiver with input $X$. As a result, $S$ obtains $v$. Next, the $S$ raises $v$ to the $k$ power as $w = v^k$ and sends the result $w$ back to the $R$. Now, the receiver can raise the $w$ to the $1/\alpha$ to obtain the final output $y$. We formally present our cOPRF construction in Figure 14.

It is not hard to see that the output of the cOPRF protocol satisfies correctness. More precisely, if $q \notin X$, $v = v_1 = H(q||0)^\alpha$, thus, the protocol’s output $y = w^{1/\alpha} = H(q||0)^k$ as desired. In case the $q \in X$, the value $y = H(q||1)^k$ is a pseudorandom value which is computationally indistinguishable to $H(q||0)^k$ when the PRF key is unknown. In general, our cOPRF protocol is secure against the same query (i.e. the same query will always leads to the same pseudorandom value no matter its membership related to sender’s set).

**Theorem 7.** The cOPRF protocol described in Figure 14 securely implements the cOPRF functionality defined in Figure 13 in the semi-honest setting, given the $m$OT functionalities described in Section 4.

**Proof.** The security follows from the security of the $m$OT functionality and the fact the value $v_i = H(q||i)^\alpha$ and $y = w^{1/\alpha}$ is distributed uniformly.
More precisely, the corrupt sender $S$ learns nothing from the mOT execution as $v_0$ and $v_1$ are in the same distribution. The value $v_1$ reveals nothing about the receiver’s input $q$ due to the secret $α$ under the Diffie–Hellman assumption.

The corrupt receiver obtains $w = v^k$ from the honest sender. Due to the secret PRF key $k$, the receiver learns nothing from $v$. Thus, simulation is trivial, as the parties’ views in the protocol are exactly the cOPRF output.

B.2 The mPSU Protocol With Non-Collusion Assumption

We present our original mPSU protocol in Figure 15, which is secure if a subset of parties does not collude. The high-level idea of this framework is to allow a leader ($P_1$ in this case) to collect a set of encrypted element $x \in X_i \setminus \bigcup_{j=1}^{i-1} X_j$ by interacting with $P_i$ for $i \in [2, n]$ in round $i - 1$. Then all parties invoke the Shuffle&Decrypt functionality to reveal the final union.

In round $i - 1$ of Step (3) show in Figure 15, there are four sub-steps are performed:

- Step (3,a): $P_i$ choose a key and invoke OPRF with $P_1$ to update the PRF value for both parties.
- Step (3,b): For each element $x \in X_i$, $P_i$ invoke mOT with $P_1$ to send the encryption depending on the membership of $x$ with respect to $\bigcup_{j=1}^{i-1} X_j$.
- Step (3,c): $P_i$ choose a key and invoke cOPRF with $P_{t \in [i+1,n]}$ to update the PRF value for the rest parties.
- Step (3,4): $P_{t \in [i+1,n]}$ invoke mOT with $P_i$ to update the encryption set.

We highlight the usage of PRF in the protocol figure.

B.3 Security Issue Identified by [DCZB24]

Dong et al. [DCZB24] pointed out that the mPSU protocol (Figure 15) is not secure when $P_1$ colludes with other parties, as the collusion can learn the intersection items. We describe this attack here, refer to their paper for a more detailed analysis.

Consider a three-party case where $P_1$ and $P_3$ each possesses a same single element and $P_2$ possesses a set $X_2$, i.e., $X_1 = \{x_1\}$, $X_3 = \{x_3\}$ and $x_1 = x_3$. According to the mPSU protocol described in Figure 15, in step (3,a), $P_1$ and $P_2$ invoke the OPRF where $P_1$ acts as the receiver with $x_1$ and $P_2$ acts as the sender with a PRF key $k_2$. $P_1$ receives the PRF value for $x_1$ as $F_{k_2}(x_1)$. In step (3,c), $P_2$ and $P_3$ invoke the cOPRF where $P_3$ acts as the receiver with $x_3$, and $P_2$ acts as the sender with the PRF key $k_2$ and the set $X_2$. $P_3$ receives the output $w$. By the definition of cOPRF functionality, if $x_3 \notin X_2$, $w = F_{k_2}(x_3)$, otherwise $w$ is a random value.

If $P_1$ and $P_3$ collude, They can check the equality of $F_{k_2}(x_1)$ and $w$ to learn extra information. If $F_{k_2}(x_1) = w$, it implies that $P_3$ receives $F_{k_2}(x_3)$ from the cOPRF indicating that $x_3 \notin X_2$. If $F_{k_2}(x_1) \neq w$, it implies that $P_3$ receives a random value from the cOPRF, indicating that $x_3 \in X_2$.

This attack can be applied to the general case where parties have more than one element. So the mPSU protocol described in Figure 15 is not secure against arbitrary colluding participants.
Parameters:

- $n$ parties $P_i \in \{\alpha\}$ for $n > 1$.
- The mOT, OPRF, Shuffle&Decrypt functionalities described in Figures 4&12&6, respectively.
- A multi-key cryptosystem (KeyGen, Enc, ParDec, FuDec, ReRand) defined in Section 2.5.
- Hashing parameters: a number of bins $\mu$, maximum bin sizes $\beta : \mathbb{Z} \rightarrow \mathbb{Z}$ for simple-hashing bins, the $h$ hash functions $H_{\lambda\kappa} : \{0,1\}^* \rightarrow [\mu]$.

Input:

- Party $P_i \in \{\alpha\}$ has $X_i = \{x_{i,1}, \ldots, x_{i,m}\}$.

Protocol:

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

2. Local Execution:

   - $P_i \in \{\alpha\}$ hashes items $X_i$ into $\mu$ bins using the Cuckoo hashing. Let $C_{b,i}^0$ denote the items in the $P_i$'s $b$th bin. $P_i$ computes the encryption $e_{b,i} = \text{Enc}(pk, C_{b,i}^0)$, for $b \in [\mu]$.
   - $P_i \in \{\alpha\}$ hashes $X_i$ into $\mu$ bins under $k$ hash functions. Let $S_{b,i}^0$ denote the set of items in the $P_i$'s $b$th bin. $P_i$ pads $S_{b,i}^0$ with dummy values to the maximum bin size $\beta(m)$.

3. $P_i$ sequentially interacts with $P_j$ for $i \in [2, n]$ as follow.

   - For each bin $b \in [\mu]$, the $P_i$ and $P_j$ invoke the functionality of OPRF where:
     - $P_i$ acts as the receiver with input the set $S_{b,i-1}^0$.
     - $P_i$ acts as the sender with input the random-chosen PRF key $k_i$.
     - $P_i$ obtains a set $S_{b,i}$ of the PRF values $F(k_i, y)$ for $y \in S_{b,i-1}^0$. Note that $F(k_i, y) = H(k_i, y || 1^k)$ for a random hash function $H$.
   - For each bin $b \in [\mu]$, $P_i$ and $P_j$ invoke a mOT instance where:
     - $P_i$ acts as the receiver with input $S_{b,i}^0$.
     - $P_i$ acts as the sender with input $(y_{b,i}, r_{b,i}) || \text{Enc}(pk, 0), y_{b,i} || e_{b,i})$. Here, $r_{b,i}$ is a random value; $y_{b,i} = F(k_i, C_{b,i-1}^0) \text{ if } C_{b,i-1} \neq \emptyset \text{ and random otherwise; } e_{b,i} = \text{Enc}(pk, 0) \text{ if } C_{b,i-1} = \emptyset$,
     - otherwise, $e_{b,i} = e_{b,i-1}$.
     - $P_i$ obtains $\bar{t}_b || e_b$.
     - $P_i$ appends $\bar{t}_b$ to $E_b$. $P_i$ hashes $\bigcup_{b=1}^\mu \{ S_{b,i}^0 \cup \bar{t}_b \}$ into $\mu$ bins under $b$ hash functions. The $P_i$ redefines $S_{b,i}^0$ to be the set of items in its $b$th bin, and then pads $S_{b,i}^0$ with dummy values to the maximum bin size $\beta(m)$.
   - For each bin $b \in [\mu]$, $P_i$ and $P_j \in \{\alpha\}$ invoke the cOPRF where:
     - $P_i$ acts as the receiver with input $C_{b,i-1}^0$ or a dummy if $C_{b,i-1} = \emptyset$.
     - $P_i$ acts as the sender with input the PRF key $k_i$ and the set $S_{b,i-1}^0$.
     - $P_i$ obtains $w_b$ and sets $w_b = \emptyset$ if $C_{b,i-1} = \emptyset$.
     - $P_i$ hashes $W = \{ w_b \mid b \in [\mu] \text{ and } w_b \neq \emptyset \}$ into $\mu$ bins using the Cuckoo and Simple hashing. Let $S_{b,i}^0$ and $C_{b,i}^0$ denote the items in the Simple and Cuckoo $b$-th bin, respectively. $P_i$ pads $S_{b,i}^0$ with dummy to maximum bin size $\beta(m)$.
   - The $P_i$ and $P_j \in \{\alpha\}$ invoke a mOT instance where:
     - $P_i$ acts as the receiver with input $\{ F(k_i, y) \mid y \in S_{b,i-1}^0 \}$
     - $P_i$ acts as the sender with input $(C_{b,i}^0, \text{Enc}(pk, 0), e_{b,i-1})$.
     - $P_i$ obtains $c$ and sends $\bar{c}' = \text{ReRand}(c, pk)$ to $P_i$.
     - $P_i$ computes $e_{b,i} = \text{ReRand}(c', pk)$

4. All the parties invoke the Shuffle&Decrypt functionality where:

   - $P_i$ inputs $E = \bigcup_{b=2}^\mu E_b$, the $sk_i$ and a random permutation $\pi_i : [m] \rightarrow [m]$.
   - $P_i$ inputs the private key $sk_i$ and a random permutation $\pi_i : [m] \rightarrow [m]$.
   - $P_i$ obtains a set $U$.

5. $P_i$ removes all zero from $U$, and outputs $U \cup X_i$.

Figure 15: Our mPSU Protocol: A Subset of Parties Do Not Collude
B.4 The Fix from the Original Design

We find that the security issue is rooted in the usage of PRFs. The fix is straightforward by removing the step that computes and uses PRF (highlighted in Figure 15). We present our secure protocol in Figure 8, where the collusion learns nothing, including the intersection items.

B.5 Discussion about the PK-MPSU in [DCZB24]

The PK-MPSU protocol in [DCZB24] is similar to our secure protocol (Figure 8). The similarity stems from the core idea of allowing the leader to collect ciphertexts and apply Shuffle&Decrypt to reveal the final result. Our contribution lies in initiating the mPSU design within this framework. [DCZB24] further improved our work by proposing an efficient batch SS-PMT protocol. It is worth mentioning that they also proposed an efficient mPSU protocol based primarily on symmetric techniques, which is secure in the standard semi-honest model.