# Fast Unbalanced Private Set Union from Fully Homomorphic Encryption

Binbin Tu[1], Yu Chen[1], Qi Liu[1], and Cong Zhang[2,3]

[1] School of Cyber Science and Technology, Shandong University
[2] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
[3] School of Cyber Security, University of Chinese Academy of Sciences
`{tubinbin,liuqicst}@mail.sdu.edu.cn;yuchen@sdu.edu.cn;zhangcong@iie.ac.cn`

**Abstract.** Private set union (PSU) allows two parties to compute the union of their sets without revealing anything else. It has found numerous applications in practice. Recently, some computationally efficient PSU protocols have been designed for the balanced case, but a limitation with these protocols is the communication complexity, which scales (super)-linear in the size of the larger set. This is of particular concern when performing PSU in the unbalanced case, where one party is a constrained device holding a small set, and another is a large service provider holding a large set.

In this work, we propose a generic construction of unbalanced PSU from leveled fully homomorphic encryption and a newly introduced protocol called permuted matrix private equality test. By instantiating the generic construction, we obtain two secure and fast unbalanced PSU protocols, whose communication complexity is linear in the size of the smaller set, and logarithmic in the larger set.

We implement our protocols and compare them with the state-of-the-art PSU. Experiments show that our protocols are more efficient than all previous protocols in the unbalanced case. Especially, the larger difference between the size of two sets, the better our protocols perform. For input sets of size $2^{10}$ and $2^{20}$ with 128-bit length items, our PSU takes 2.767 MB of communication to compute the union. Compared with the state-of-the-art PSU proposed by Zhang et al. (Usenix Security 2023), there are 37× reductions in communication and roughly $10-35\times$ reductions in computational overhead depending on network environments.

## 1 Introduction

PSU is a cryptographic protocol that allows two parties, a sender and a receiver with respective input sets $X$ and $Y$, to compute the union $X \cup Y$, without revealing anything else. It has become considerably efficient and has been deployed in practice, such as cyber risk assessment [LV04, HLS$^+$16, KRTW19], privacy-preserving data aggregation [BSMD10], and private ID [GMR$^+$21] etc. However, most PSU [KS05, Fri07, BA12, DC17, KRTW19, GMR$^+$21] are designed in the balanced case. These protocols typically perform only marginally better when one of the sets is much smaller than the other. In particular, their communication cost scales at least linear in the size of the larger set.

In most real world applications, the sender's set might be much smaller than the receiver's, such as the sender (client) might be a mobile device with limited battery, computing power, and storage, whereas the receiver (server) is a high-end computing device. Meanwhile, the bandwidth between two parties might be limited. As shown in [ZCL$^+$23, JSZ$^+$22, KRTW19], blacklist aggregation is a typical application of PSU. According to [RMY20], they collect around 176 million blacklisted IP addresses over 23,483 autonomous systems, where the size of IP blacklist varies greatly (large blacklists listing more than 500,000 IP addresses and small blacklists listing fewer than 1,000 IP addresses). Therefore, the aggregation of IP blacklists with large size differences constitutes a representative use case of unbalanced PSU. However, most existing PSU protocols are not very efficient in dealing with the unbalanced case.

Over the last decade, there has been a significant amount of work on private set intersection (PSI) including both balanced [PRTY19, PSTY19, CO18, FNO19, CCS18, HV17, RR17, PSZ18, PSZ14, DCW13, HEK12, KMP$^+$17] and unbalanced case [CLR17, KLS$^+$17, CHLR18, RA18, CMdG$^+$21], but little attention has been paid on PSU, especially in the unbalanced case. Recently, Jia et al. [JSZ$^+$22] propose an unbalanced

PSU$^{*4}$ that uses a shuffling technique, but their PSU$^*$ suffers from two primary drawbacks. Firstly, their PSU$^*$ does not satisfy standard security, since it *leaks the information of the intersection size* to the sender. Such information leakage could be critical for PSU. Consider the extreme case, in which the sender can get the intersection item if it inputs a one-element set. Secondly, the communication complexity of their PSU$^*$ is *linear in the size of the larger set.* Another closely related work is that of Chen et al. [CZZD22], which shows how to tweak FHE-based PSI [CLR17] to an unbalanced PSU protocol. As noted by the authors, their PSU protocol only serves as a proof of concept since it reveals intersection size to the sender, and straightforward application of the optimization tricks due to [CLR17] will compromise the semi-honest security of the receiver. They left the standard security and efficient FHE-based PSU protocol in the unbalanced setting as a challenging problem.

Motivated by the above discussions, we ask the following question:

*Is it possible to design a secure and fast unbalanced PSU protocol which has a communication overhead linear in the smaller set and logarithmic in the larger set?*

### 1.1 Contributions

In this paper, we give an affirmative answer to the above question. Our contributions are summarized as follows:

1. We first propose a basic unbalanced PSU protocol based on leveled fully homomorphic encryption (FHE). Then, we use an array of optimization techniques following [CLR17, CHLR18, CMdG$^+$21] to optimize the basic protocol, while the optimization might leak some information of the intersection.
2. We introduce a new cryptographic protocol named permuted matrix Private EQuality Test (pm-PEQT) to avoid the information leakage. Then, we give two constructions of pm-PEQT. The first is based on $\mathsf{Permute} + \mathsf{Share}$ [MS13, CGP20] and multi-point oblivious pseudorandom function (mp-OPRF). The second is based on the decisional Diffie-Hellman (DDH) assumption.
3. We present a generic construction of unbalanced PSU in the semi-honest model from leveled FHE and pm-PEQT. By instantiating the generic construction, we obtain two secure and fast unbalanced PSU protocols whose communication complexity *linear* in the size of the smaller set, and *logarithmic* in the larger set. Our protocols are particularly powerful when the set size of one party is much larger than that of the other.
4. We implement our PSU protocols and compare them with the state-of-the-art PSU. Experiments show that our protocols are more efficient than all previous protocols in the unbalanced case. For unbalanced sets size ($|X| = 2^{10}$, $|Y| = 2^{20}$) with 128-bit length items, our PSU protocol takes 2.767 MB of communication and about 18 seconds of computation to compute the union with a single thread in LAN settings. Compared with the state-of-the-art PSU [ZCL$^+$23], there are roughly 37× reduction in communication and 21× reduction in computational overhead. In particular, the performance of our PSU protocols improve significantly in the case of low bandwidth. Our PSU requires 24.75 seconds in 1Mbps bandwidth which is about 35× faster than PSU [ZCL$^+$23].

### 1.2 Related Works

We revisit recent PSU protocols [KRTW19, GMR$^+$21, JSZ$^+$22, ZCL$^+$23] with good efficiency. Table 1 provides a brief comparison of our protocols with the prior highest-performing PSU protocols. The performance results and comparisons are reported in Section 7.

Kolesnikov et al. [KRTW19] propose a PSU protocol based on the reverse private membership test (RPMT). In RPMT, a sender with input $x$ interacts with a receiver holding a set $Y$, and the receiver can learn a bit indicating whether $x \in Y$, while the sender learns nothing. Then, the receiver runs OT with the sender to obtain $\{x\} \cup Y$. For $|X| = |Y| = n$, the protocol runs $n$ RPMT independently and requires $O(n^2)$ communication and $O(n^2 \log^2 n)$ computation. By using the bucketing technique, two parties hash

---

[4] In this paper, we use PSU$^*$ to indicate a PSU protocol with information leakage.

| Protocols | Communication | Computation | Security |
|---|---|---|---|
| PSU$^*$ [KRTW19] | $O(n \log n)$ | $O(n \log n)$ | Leaky |
| PSU [GMR$^+$21] | $O(n \log n)$ | $O(n \log n)$ | Standard |
| PSU [JSZ$^+$22] | $O(n \log n)$ | $O(n \log n)$ | Standard |
| PSU$^*$ [JSZ$^+$22] | $O(n + m \log m)$ | $O(n)$ | Leaky |
| PSU [ZCL$^+$23] | $O(n)$ | $O(n)$ | Standard |
| Our PSU | $O(m \log n)$ | $O(n)$ | Standard |

Table 1: Comparisons of PSU in the semi-honest setting. $n$ and $m$ denote the size of the large set and the small set, respectively. The information of some subsets holding the intersection items is leaked to the receiver in PSU$^*$ [KRTW19]. The information of the intersection size is leaked to the sender in PSU$^*$ [JSZ$^+$22].

their sets into $\beta$ bins and each bin consists of $\rho$ items. A large $(n, n)$-PSU[5] is divided into $\beta$ small $(\rho, \rho)$-PSU. The complexity is reduced to $O(n \log n)$ communication and $O(n \log n \log \log n)$ computation. However, [JSZ$^+$22] points out that the bucketing technique leaks the information to $\mathcal{R}$. More precisely, $\mathcal{R}$ learns that some subsets (size $\rho$) hold the intersection items with high probability.

Garimella et al. [GMR$^+$21] give a PSU protocol based on permuted characteristic functionality which in turn can be built from oblivious switching. Simply speaking, a sender holding a set $X$ interacts with a receiver holding a set $Y$. As a result, the sender gets a random permutation $\pi$ and the receiver obtains a vector $\mathbf{e} \in \{0,1\}^n$, where if $e_i = 1$, $x_{\pi(i)} \in Y$, else $x_{\pi(i)} \notin Y$. Then, the receiver runs OT protocol with the sender to obtain the set union. Their protocol requires $O(n \log n)$ communication and $O(n \log n)$ computation.

Jia et al. [JSZ$^+$22] propose a PSU with the shuffling technique. Roughly speaking, a receiver hashes a set $Y$ into $Y_c$ by Cuckoo hash and a sender hashes a set $X$ by the simple hash. Two parties invoke a $\mathsf{Permute} + \mathsf{Share}$ functionality [MS13, CGP20]. The receiver shuffles $Y_c$ by a permutation $\pi$ chosen by the sender. The sender and the receiver obtain shuffled shares $\{s_{\pi(i)}\}$ and $\{s'_{\pi(i)}\}$, respectively, where $Y_c[\pi(i)] = s_{\pi(i)} \oplus s'_{\pi(i)}$. Two parties run mp-OPRF to compute all PRF values and the sender sends its PRF values to the receiver. The receiver tests which items belong to the union and runs OT with the sender to get the union. Their PSU requires $O(n \log n)$ communication and $O(n \log n)$ computation. They also consider the unbalanced case and give an unbalanced PSU$^*$ which requires $O(n + m \log m)$ communication and $O(n)$ computation.

Zhang et al. [ZCL$^+$23] recently give a generic framework of PSU based on the multi-query reverse private membership test (mq-RPMT). In mq-RPMT, a sender holding a set $X$ interacts with a receiver holding a set $Y$. As a result, the sender gets nothing and the receiver gets $\mathbf{b} \in \{0,1\}^n$, satisfying that $b_i = 1$ if and only if $x_i \in Y$. Then, two parties runs OT protocol to let the receiver get the set union. To construct mq-RPMT, they combine the oblivious key-value store (OKVS) [GPR$^+$21] and the vector decryption-then-matching (VODM). By instantiating OKVS and VODM, they obtain two concrete mq-RPMT. The first is based on the symmetric-key encryption and general 2PC. The second is based on the re-randomizable public-key encryption. Both constructions achieve $O(n)$ communication and $O(n)$ computation.

## 2 Overview of Our Techniques

We provide the high-level intuition for our unbalanced PSU protocol. First, we propose a basic PSU protocol based on leveled FHE. Our basic protocol is easy to understand, but it is not efficient due to the deep depth of homomorphic circuits. Then, we try to improve the basic PSU by applying optimization techniques following [CLR17, CHLR18, CMdG$^+$21] to reduce the depth of homomorphic circuits. However, straightforward application of these optimization techniques leaks the information of the intersection. To remedy the leakage, we introduce a new cryptographic protocol called permuted matrix private equality test (pm-PEQT). Finally,

---

[5] In this paper, we use $(m, n)$-PSU to indicate a PSU protocol where the sender's set size is $m$ and the receiver's set size is $n$.

we manage to give a generic construction of standardly secure unbalanced PSU from leveled FHE and pm-PEQT. By instantiating the generic construction, we obtain secure and fast unbalanced PSU protocols in the semi-honest model. We describe the ideal functionality of PSU in Figure 1.

**Parameters:** Set sizes $m$ and $n$ are public.
**Functionality:**

1. Wait for an input $X = \{x_1, \cdots, x_m\} \subseteq \{0,1\}^*$ from the sender, and an input $Y = \{y_1, \cdots, y_n\} \subseteq \{0,1\}^*$ from the receiver.
2. Give output $X \cup Y$ to the receiver.

Fig. 1: Ideal functionality $\mathcal{F}_{\mathrm{PSU}}^{m,n}$ for private set union

### 2.1 Notation

We denote the parties in our PSU as the sender $\mathcal{S}$ and the receiver $\mathcal{R}$, and their respective input sets as $X$ and $Y$ with $m = |X| \ll n = |Y|$. For $n \in \mathbb{N}$, let $[n]$ denote the set $\{1, 2, \cdots, n\}$. $1^\lambda$ denotes the string of $\lambda$ ones. We use $\kappa$ and $\lambda$ to indicate the computational and statistical security parameters, respectively. If $S$ is a set, $s \leftarrow S$ indicates sampling $s$ from $S$ at random. We denote vectors by lower-case bold letters, e.g., $\mathbf{s}$. We denote matrices by upper-case bold letters, e.g., $\mathbf{S}$. We write $\mathbf{S} = [s_{ij}]$ to denote the matrix $\mathbf{S}$ with $i$-th row and $j$-th column element $s_{ij}$. For a permutation $\pi$ over $n$ items, we write $\{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ to denote $\pi(\{s_1, \cdots, s_n\})$, where $s_{\pi(i)}$ indicates the $i$-th element after the permutation. For a column permutation $\pi_c$ (or, row permutation $\pi_r$) on a matrix $\mathbf{S} = [s_{ij}]$, we write $\mathbf{S}_{\pi_c}$ (or, $\mathbf{S}_{\pi_r}$) to denote that $\pi_c(\mathbf{S}) = [s_{\pi_c(ij)}]$ (or, $\pi_r(\mathbf{S}) = [s_{\pi_r(ij)}]$) is the permuted matrix, where $s_{\pi_c(ij)}$ (or, $s_{\pi_r(ij)}$) indicates the $i$-th row and $j$-th column element after the permutation.

### 2.2 Our Basic PSU Protocol

Our starting point is the FHE-based basic PSI protocol [CLR17]. New, we review the protocol as follows.

**Basic PSI protocol of Chen et al revisit.** Chen et al. [CLR17] give a basic unbalanced PSI protocol, in which $\mathcal{R}$ holding an item $y$ interacts with $\mathcal{S}$ holding a large set $X$, and $\mathcal{R}$ can get the intersection $\{y\} \cap X$. Informally, $\mathcal{R}$ encrypts its item $y$, and sends the ciphertext $c \leftarrow \mathrm{FHE}.\mathsf{Enc}(y)$ to $\mathcal{S}$; $\mathcal{S}$ chooses random non-zero plaintexts $r$ and homomorphically computes $c' \leftarrow \mathrm{FHE}.\mathsf{Enc}(r \cdot f(y))$, where the polynomial $f(x) = \mathit{\Pi}_{x_i \in X}(x - x_i)$, and then returns $c'$ to $\mathcal{R}$; $\mathcal{R}$ decrypts $c'$: if $rf(y) = 0$, it knows $y \in X$ and outputs $\{y\}$, else, it gets a random value and outputs $\emptyset$. The protocol requires communication linear in the smaller set, achieving optimal communication that is on par with the naive solution, but it has high computational costs and deep homomorphic circuits, since the degree of $f(x)$ is related to the large set size.

**Basic PSU protocol.** The functionality adjustment (PSI → PSU) doesn't seem to be straightforward, since the randomized product $rf(y) = 0$ leaks the information of the intersection to the receiver. The main challenge is to find a new randomization method that hides the information of the intersection and admits to check which items belong to the union. We solve the problem by adding a random value $r$ to randomize the polynomial value. In this way, the randomized value $r + f(y)$ leaks nothing to $\mathcal{R}$. Meanwhile, $\mathcal{R}$ sends the result $r + f(y)$ to $\mathcal{S}$ and $\mathcal{S}$ checks whether the item $y$ belongs to the union by verifying $r \overset{?}{=} r + f(y)$. Finally, $\mathcal{S}$ obtains the union by OT protocol. In order to let the receiver output the results (requirements of the ideal functionality of PSU), we consider the dual structure of [CLR17]. In our PSU, the receiver holding a large set interacts with the sender holding a small set and gets the union.

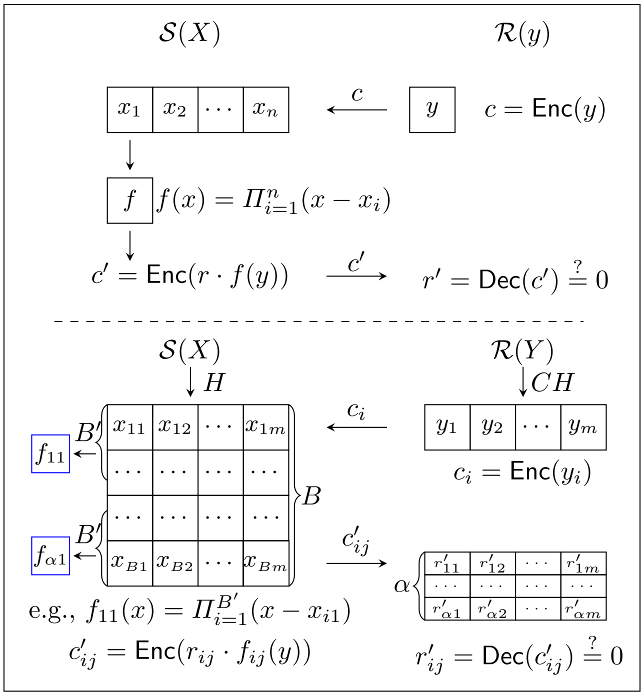

Fig. 2: The basic PSI and its optimizations [CLR17]

We start with a special case of our basic PSU. Suppose that $\mathcal{S}$ has only one item $x$ and $\mathcal{R}$ holding a large set $Y$ gets the resulting union $\{x\} \cup Y$. We show our basic unbalanced PSU based on leveled FHE as follows: $\mathcal{S}$ uses its public key to encrypt the item $x$ and sends the ciphertext $c = \text{FHE}.\mathsf{Enc}(x)$ to $\mathcal{R}$; $\mathcal{R}$ chooses random non-zero value $r$, and homomorphically computes $c' = \text{FHE}.\mathsf{Enc}(r + f(x))$, where the polynomial $f(x) = \Pi_{y_i \in Y}(x - y_i)$ and returns the new ciphertext to $\mathcal{S}$; $\mathcal{S}$ decrypts $c'$, gets the plaintext $r' = r + f(x)$ and returns $r'$ back to $\mathcal{R}$; $\mathcal{R}$ checks $r' \overset{?}{=} r$, if $r' = r$, sets $b = 0$ indicating $x \in Y$, else $b = 1$. Finally, $\mathcal{R}$ invokes the OT protocol with $\mathcal{S}$ to obtain the union $\{x\} \cup Y$. The detail of our basic PSU construction is given in Figure 9.

The key different step between our basic PSU and the basic PSI [CLR17] is that we use different randomization methods. We compute the sum of a random value $r$ and the polynomial value $f(x)$. $\mathcal{S}$ obtains the random plaintext $r' = r + f(x)$ which leaks nothing and $\mathcal{R}$ getting $r' = r + f(x)$ can checks whether $r' = r$ or not. If $r' = r$, we have $x \in Y$, else $x \notin Y$. As a result, $\mathcal{R}$ can get the union by OT protocol. Note that this protocol will leak some information of $x \notin Y$, but such leakage does not cause any harm to the PSU as shown in [KRTW19], since the PSU protocol releases that value at last.

### 2.3 Optimized PSU with Leakage

We first review optimized unbalanced PSI as follows. Chen et al. [CLR17] use an array of optimization techniques such as hashing, batching, windowing, partitioning, and modulus switching to optimize their basic protocol and obtain a fast unbalanced PSI. Informally, $\mathcal{R}$ inserts the small set $Y$ into Cuckoo hash table $Y_c$ by Cuckoo hash and each bin $Y_c[i]$ consists of one item. $\mathcal{S}$ inserts the large set $X$ into hash table $X_b$ by simple hash, where the $i$-th bin indicates $X_b[i]$ and each bin consists of $B$ items. $\mathcal{S}$ partitions each bin $X_b[i]$ into $\alpha$ subsets and each subset consists of $B' = B/\alpha$ items. Therefore, the large $(n, m)$-PSI is divided into many small $(B', 1)$-PSI. For each small PSI, $\mathcal{S}$ encodes each subset ($B'$ items) into a polynomial and randomizes it by multiplying a random value, then it homomorphically computes and sends the new

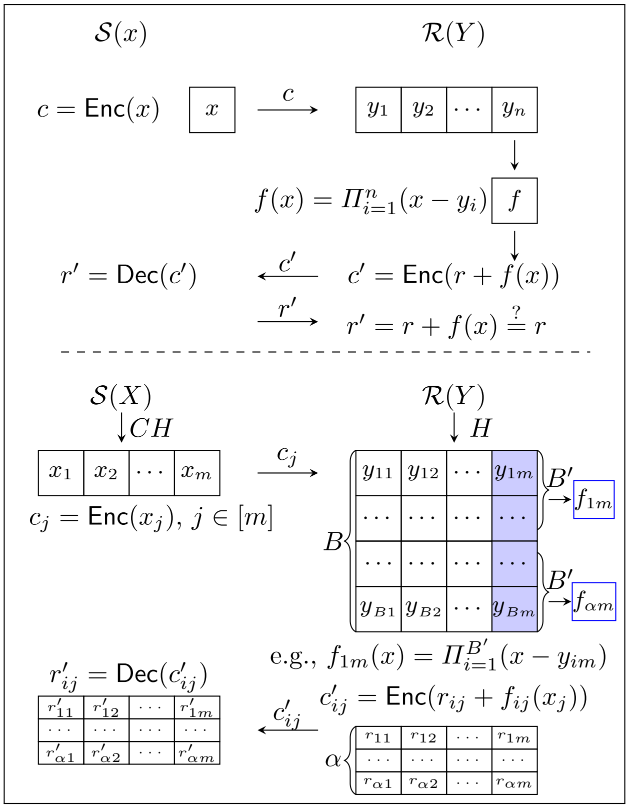

Fig. 3: The basic PSU (omit OT) and its optimizations

ciphertexts to $\mathcal{R}$. $\mathcal{R}$ then decrypts the ciphertexts and gets the set intersection. Since the degree of the polynomial is related to the small subset size $B'$, each small PSI has a low homomorphic circuit. We review the basic PSI protocol and its optimizations in Figure 2.

According to the requirements of the ideal functionality of PSU, we consider the dual structure of [CLR17]. In our PSU, the receiver holds a large set $Y$ and the sender holds a small set $X$. It is tempting to use the same optimization techniques [CLR17] to improve our basic PSU. Roughly, $\mathcal{S}$ hashes the small set $X$ into $X_c$ by Cuckoo hash and each bin $X_c[i]$ consists of one item. $\mathcal{R}$ hashes the large set $Y$ into hash table $Y_b$ and each bin $Y_b[i]$ consists of $B$ items. Then $\mathcal{R}$ partitions each bin $Y_b[i]$ into $\alpha$ subsets and each subset consists of $B' = B/\alpha$ items. The large $(m, n)$-PSU is divided into many small $(1, B')$-PSU. For each small PSU, $\mathcal{R}$ encodes each subset ($B'$ items) into a polynomial and randomizes it by adding a random value, then it homomorphically computes and sends new ciphertexts to $\mathcal{S}$. $\mathcal{S}$ decrypts the ciphertexts and sends the plaintexts back. $\mathcal{R}$ checks which items belong to the set union, and invokes OT with $\mathcal{S}$ to get them. We show our basic PSU (omit OT) and its optimizations in Figure 3.

We emphasize that, unlike PSI [CLR17], the optimization techniques for PSI is not suitable for our PSU. This is because a large PSI can be divided into many small PSI, and the receiver can combine all small set intersections into the output securely. However, if we divide a large $(m, n)$-PSU into many small $(1, B')$-PSU directly, this causes information leakage about the intersection. We show the comparison of PSI [CLR17] and our optimized PSU (omit OT) with leakage in Figure 4. Note that in the standardly secure $(m, n)$-PSU, from the view of $\mathcal{R}$, any item in the set $Y$ could be an item in $X \cap Y$. However, in the above optimized PSU$^*$, $\mathcal{R}$ learns some subsets with size $B'$ have the item in $X \cap Y$. Moreover, if $\mathcal{S}$ returns its decrypted results $r'$ to $\mathcal{R}$ directly. $\mathcal{R}$ can check which items of $X$ belong to the set union. This also leaks the information of $X \cap Y$. Because there are $\alpha$ subsets with size $B'$ in one bin, if $f(x) = 0$ in one subset, $\mathcal{R}$ gets $f'(x) \neq 0$ in other subsets, which causes $\mathcal{R}$ could compute the intersection items with sufficient polynomial values. For

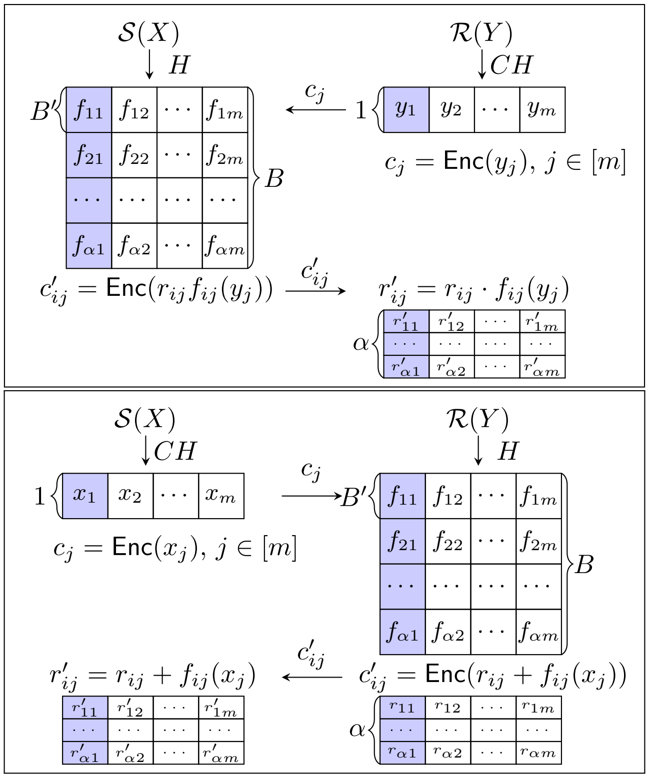

Fig. 4: Comparison of PSI [CLR17] (top) and optimized PSU (omit OT) with leakage (bottom)

example, in Figure 4 (bottom), in the first column, if $r_{11} = r'_{11}$, this means $x_1 \in Y$[6] and $x_1 \in \{y_{11}, \cdots, y_{B'1}\}$, but $x_1 \notin \{y_{(B'+1)1}, \cdots, y_{B1}\}$. $\mathcal{R}$ gets the rest nonzero polynomial values $f_{21}(x_1), \cdots, f_{\alpha 1}(x_1)$ and it could compute $x_1$ from them.

Based on the above analysis, the main challenge is how to optimize our basic PSU without causing information leakage. More precisely, for $\mathcal{S}$ holds a matrix $\mathbf{R}' = [r'_{ij}]$ and $\mathcal{R}$ holds a matrix $\mathbf{R} = [r_{ij}]$, $i \in [\alpha], j \in [m]$, we need to overcome the following two difficulties:

- The receiver is able to check $r_{ij} \overset{?}{=} r'_{ij}$ for all $i \in [\alpha], j \in [m]$ without knowing the value $r'_{ij}$ held by the sender.
- The receiver is able to check $r_{ij} \overset{?}{=} r'_{ij}$ for all $i \in [\alpha], j \in [m]$ without knowing the positions $i, j$ of $r_{ij}$.

We address the difficulties by introducing a new cryptographic protocol named permuted matrix private equality test (pm-PEQT) that enables the receiver to check whether the values in permuted positions are equal without knowing the values and permutation.

### 2.4 Permuted Matrix Private Equality Test

Here, we introduce a new cryptographic protocol named permuted matrix private equality test (pm-PEQT) which can be seen as an extension of private equality test (PEQT). In the PEQT, a receiver who has an input string $x$ interacts with a sender holding an input string $y$, and the result is that the receiver

---

[6] In $j$-th column, as long as there is a position $i$, such that $r_{ij} = r'_{ij}$, $x_j \in Y$. Meanwhile, at most one position is equal in each column.

learns a bit indicating whether $x = y$ merely, whereas the sender learns nothing. In our pm-PEQT, a sender holding a matrix $\mathbf{R}'_{\alpha\times m}$ and a matrix permutation $\pi = (\pi_c, \pi_r)$ interacts with a receiver holding a matrix $\mathbf{R}_{\alpha\times m}$. As a result, the receiver learns (only) the bit matrix $\mathbf{B}_{\alpha\times m}$ indicating that if $b_{ij} = 1$, $r_{\pi(ij)} = r'_{\pi(ij)}$, else, $r_{\pi(ij)} \neq r'_{\pi(ij)}$, $i \in [\alpha], j \in [m]$, while the sender learns nothing about $\mathbf{R}$. Compared with PEQT, pm-PEQT admits a matrix private equality test with *positions permutation*. We show the ideal functionality of pm-PEQT in Figure 5.

**Parameters:** Two parties: The receiver with a matrix $\mathbf{R}_{\alpha\times m}$. The sender with a matrix $\mathbf{R}'_{\alpha\times m}$ and a matrix permutation $\pi = (\pi_c, \pi_r)$, where $\pi_c$ (over $[m]$) is a column permutation and $\pi_r$ (over $[\alpha]$) is a row permutation. $\alpha$ and $m$ are public.
**Functionality:**

1. Wait for an input $\mathbf{R}' = [r_{ij}]$, $i \in [\alpha], j \in [m]$ and a permutation $\pi = (\pi_c, \pi_r)$ from $\mathcal{S}$, and an input $\mathbf{R} = [r_{ij}]$, $i \in [\alpha], j \in [m]$ from $\mathcal{R}$.
2. Give the bit matrix $\mathbf{B}_{\alpha\times m} = [b_{ij}]$ to $\mathcal{R}$, where $b_{ij} = 1$, if $r_{\pi(ij)} = r'_{\pi(ij)}$ and $b_{ij} = 0$ otherwise, for all $i \in [\alpha], j \in [m]$.

Fig. 5: Permuted matrix private equality test $\mathcal{F}_{\text{pm-PEQT}}$

**Constructions of pm-PEQT.** pm-PEQT can not be easily built from PEQT by running many PEQT instances in parallel. The main difficulty is to shuffle the receiver's items without knowing the permutation of the sender. We give two constructions of pm-PEQT as follows.

The first construction is based on Permute + Share [MS13, CGP20] and mp-OPRF [CM20]. Informally, $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal Permute + Share functionality $\mathcal{F}_{\text{PS}}$ twice: First, both parties permute and share the columns of $\mathbf{R}$. $\mathcal{R}$ inputs each column of $\mathbf{R}$ and $\mathcal{S}$ inputs a permutation $\pi_c$ over $[m]$. As a result, $\mathcal{R}$ gets shuffled share $\mathbf{S}_{\pi_c} = [s_{\pi_c(ij)}]$ and $\mathcal{S}$ gets $\mathbf{S}'_{\pi_c} = [s'_{\pi_c(ij)}]$, where $s_{\pi_c(ij)} \oplus s'_{\pi_c(ij)} = r_{\pi_c(ij)}$. Then both parties permute and share the rows of $\mathbf{S}_{\pi_c}$. $\mathcal{R}$ inputs each row of $\mathbf{S}_{\pi_c}$ and $\mathcal{S}$ inputs a permutation $\pi_r$ over $[\alpha]$. As a result, $\mathcal{R}$ gets $\mathbf{S}_{\pi_r} = [s_{\pi_r(ij)}]$ and $\mathcal{S}$ gets $\mathbf{S}'_{\pi_r} = [s'_{\pi_r(ij)}]$, where $s_{\pi_r(ij)} \oplus s'_{\pi_r(ij)} = s_{\pi_c(ij)}$. $\mathcal{R}$ defines the shuffled matrix shares $\mathbf{S}_{\pi} = \mathbf{S}_{\pi_r}$ and $\mathcal{S}$ defines the shuffled matrix shares $\mathbf{S}'_{\pi} = \pi_r(\mathbf{S}'_{\pi_c}) \oplus \mathbf{S}'_{\pi_r}$, where $s_{\pi(ij)} \oplus s'_{\pi(ij)} = r_{\pi(ij)}$, $i \in [\alpha], j \in [m]$. Then, both parties invoke mp-OPRF functionality $\mathcal{F}_{\text{mp-OPRF}}$. $\mathcal{R}$ inputs shuffled shares $\mathbf{S}_{\pi}$ and obtains $F_k(s_{\pi(ij)})$, $i \in [\alpha], j \in [m]$, and $\mathcal{S}$ gets the key $k$ of a PRF. Furthermore, $\mathcal{S}$ permutes the matrix $\mathbf{R}'$ by $\pi = (\pi_c, \pi_r)$ and gets $\mathbf{R}'_{\pi} = [r'_{\pi(ij)}]$, then $\mathcal{S}$ computes all PRF values $F_k(r'_{\pi(ij)} \oplus s'_{\pi(ij)})$, $i \in [\alpha], j \in [m]$ and sends them to $\mathcal{R}$. Finally, $\mathcal{R}$ sets $b_{ij} = 1$, if $F_k(s_{\pi(ij)}) = F_k(r'_{\pi(ij)} \oplus s'_{\pi(ij)})$, else, sets $b_{ij} = 0$. $\mathcal{R}$ lets the bit matrix $\mathbf{B} = [b_{ij}]$, $i \in [\alpha], j \in [m]$. The details of this construction are deferred to Figure 10. Note that the Permute + Share [MS13, CGP20] and mp-OPRF [CM20] are fast cryptographic tools. The communication complexity of our pm-PEQT based on Permute + Share and mp-OPRF is $O(\alpha m \log \alpha m)$.

The second construction is based on the DDH assumption [Bon98]. Let $\mathbb{G}$ be a cyclic group with order $q$. $\mathcal{R}$ and $\mathcal{S}$ choose random values $a$, $b \leftarrow \mathbb{Z}_q$ and compute $v_{ij} = H(r_{ij})^a$, $v'_{ij} = H(r'_{ij})^b$ for all $i \in [\alpha]$, $j \in [m]$, respectively, where the output of $H(\cdot)$ is a group element in $\mathbb{G}$. Let $\mathbf{V} = [v_{ij}]$ and $\mathbf{V}' = [v'_{ij}]$, $i \in [\alpha]$, $j \in [m]$. $\mathcal{R}$ sends $\mathbf{V}$ to $\mathcal{S}$. Then $\mathcal{S}$ computes $v''_{ij} = (v_{ij})^b$ and defines $\mathbf{V}'' = [v''_{ij}]$, $i \in [\alpha]$, $j \in [m]$. $\mathcal{S}$ shuffles $\mathbf{V}''$ and $\mathbf{V}'$ by same permutation $\pi = (\pi_c, \pi_r)$ and gets $\mathbf{V}''_{\pi} = \pi(\mathbf{V}'')$, $\mathbf{V}'_{\pi} = \pi(\mathbf{V}')$, where $v''_{\pi(ij)} = \pi(v''_{ij})$, $v'_{\pi(ij)} = \pi(v'_{ij})$. $\mathcal{S}$ sends permuted matrices $\mathbf{V}''_{\pi}$ and $\mathbf{V}'_{\pi}$ to $\mathcal{R}$. Finally, for $i$-th row and $j$-column in $\mathbf{V}''_{\pi}$ and $\mathbf{V}'_{\pi}$, if $v''_{\pi(ij)} = v'^{a}_{\pi(ij)}$, $\mathcal{R}$ defines $b_{ij} = 1$ and $b_{ij} = 0$ otherwise. $\mathcal{R}$ gets a bit matrix $\mathbf{B} = [b_{ij}]$, $i \in [\alpha]$, $j \in [m]$. The details of this construction are deferred to Figure 11. The communication complexity of our DDH-based pm-PEQT is $O(\alpha m)$.

### 2.5 Our Full PSU Protocol

Now, we are ready to describe our full PSU protocol. We provide the high-level technical overview for our generic construction of PSU as follows.

First, we construct a semi-finished FHE-based optimized PSU$^*$ in which $\mathcal{S}$ does not send its decrypted results $\mathbf{R}'_{\alpha\times m}$ to $\mathcal{R}$. In this step, $\mathcal{S}$ holding a small set $X$ interacts with $\mathcal{R}$ holding a large set $Y$. The result is that $\mathcal{R}$ outputs a matrix $\mathbf{R}_{\alpha\times m} = [r_{ij}]$, and $\mathcal{S}$ outputs a matrix $\mathbf{R}'_{\alpha\times m} = [r'_{ij}]$, where $\alpha$ denotes the number of partitions, $m$ denotes the number of bins, $r_{ij}$ denotes the random values used to hide each polynomial and $r'_{ij}$ denotes the decrypted results, $i \in [\alpha]$, $j \in [m]$. Note that for all $i \in [\alpha]$ in same $j$-th column, if all $r'_{ij} \neq r_{ij}$, $x_j \notin Y$, else, $x_j \in Y$.

Then, two parties invoke the pm-PEQT functionality, $\mathcal{S}$ inputs $\mathbf{R}'$ and a permutation $\pi = (\pi_c, \pi_r)$[7] and $\mathcal{R}$ inputs $\mathbf{R}$. As a result, $\mathcal{R}$ gets a bit matrix $\mathbf{B} = [b_{ij}]$, where $b_{ij} = 1$, if and only if $r_{\pi(ij)} = r'_{\pi(ij)}$ for $i \in [\alpha], j \in [m]$. $\mathcal{R}$ computes a bit vector $\mathbf{b} = [b_j]$, $j \in [m]$, for all $i \in [\alpha]$, if $b_{ij} = 0$, sets $b_j = 1$, else, sets $b_j = 0$. $\mathcal{S}$ permutes the Cuckoo hash table $X_c$ by $\pi_c$ and gets $\pi_c(X_c) = [x_{\pi_c(1)}, \cdots, x_{\pi_c(m)}]$. We note that if $b_j = 1$, we have $x_{\pi_c(j)} \notin Y$, else $x_{\pi_c(i)} \in Y$, $j \in [m]$.

Finally, for $j \in [m]$, two parties execute OT protocol as follows. In the $j$-th OT instance, $\mathcal{S}$ inputs $(\perp, x_{\pi_c(j)})$, and $\mathcal{R}$ inputs $b_j$, $j \in [m]$. As a result, $\mathcal{R}$ obtains all $x_{\pi_c(j)}$ for $b_j = 1$. $\mathcal{R}$ outputs the union $Y \cup \{x_{\pi_c(j)}\}$.

In this way, we complete our constructions of secure and fast unbalanced PSU protocols.

## 3 Preliminaries

### 3.1 Building Blocks

We briefly review the main cryptographic tools including simple hashing, Cuckoo hashing, leveled fully homomorphic encryption, oblivious transfer, multi-point oblivious PRF, and Permute + Share.

**Simple hashing.** In the simple hashing [PSZ14], the hash table consists of $m$ bins $B_1, \cdots, B_m$. Hashing is done by mapping each element $x$ to a bin $B_{h(x)}$ using a hash function $h : \{0,1\}^* \to [m]$ that was chosen uniformly at random and independently of the input elements. According to the following inequality [MR95], the maximum bin size $B$ can be set to ensure that no bin will contain more than $B$ items except with probability $2^{-\lambda}$ when hashing $n$ items into $m$ bins.

$$\Pr[\exists \text{ bin size} \geq B] \leq m \left[ \sum_{i=B}^{n} \binom{n}{i} \cdot \left(\frac{1}{m}\right)^{i} \cdot \left(1 - \frac{1}{m}\right)^{n-i} \right]$$

**Cuckoo hashing.** Cuckoo hashing [PR01, DM03, FPSS03, PSZ18] can be used to build dense hash tables by many hash functions. Following [CLR17, CHLR18], we use three hash functions and adjust the number of items and table size to reduce the stash size to 0 while achieving a hashing failure probability of $2^{-\lambda}$.

**Leveled fully homomorphic encryption**. The leveled fully homomorphic encryption supports circuits of a certain bounded depth. Following [CLR17], our protocols require the leveled FHE satisfies IND-CPA secure with circuit privacy [BPMW16]. We use an array of optimization techniques of FHE as [CLR17, CHLR18, CMdG$^+$21], such as batching, windowing, partitioning, and modulus switching to significantly reduce the depth of the homomorphic circuit. We review these optimizations in appendix A. For the implementation, we use the homomorphic encryption library SEAL which implements the BFV scheme [FV12] following [CLR17, CHLR18, CMdG$^+$21].

---

[7] Each column of the matrix $\mathbf{R}_{\alpha\times m}$ corresponds to the same item, so the permutation for the matrix requires that the columns are consistent.

**Oblivious transfer.** Oblivious transfer [Rab05] is a central cryptographic primitive in the area of secure computation. In the 1-out-of-2 OT, a sender with two input strings $(x_0, x_1)$ interacts with a receiver who has an input choice bit $b$. The result is that the receiver learns $x_b$ without learning anything about $x_{1-b}$, while the sender learns nothing about $b$. Ishai et al. [IKNP03] introduced the OT extension that allows for a large number of OT executions at the cost of computing a small number of public-key operations. We recall the 1-out-of-2 oblivious transfer functionality $\mathcal{F}_{\mathrm{OT}}$ in Figure 6.

**Parameters:** Two parties: $\mathcal{S}$ and $\mathcal{R}$.
**Functionality:**

1. Wait for input $\{x_0, x_1\}$ from $\mathcal{S}$. Wait for input $b \in \{0, 1\}$ from $\mathcal{R}$.
2. Give $x_b$ to $\mathcal{R}$.

Fig. 6: 1-out-of-2 oblivious transfer functionality $\mathcal{F}_{\mathrm{OT}}$

**Multi-point oblivious pseudorandom function.** An oblivious pseudorandom function (OPRF) allows a receiver to input $x$ and learns the PRF value $F_k(x)$, and the key $k$ is known to a sender. Pinkas et al. [PRTY19] propose multi-point OPRF (mp-OPRF) and realize efficient PSI protocols. Recently, Chase and Miao [CM20] propose a more efficient mp-OPRF based on oblivious transfer extension. In the mp-OPRF, the receiver inputs $\{x_1, x_2, \cdots, x_n\}$ and learns all PRF values $\{F_k(x_1), F_k(x_2), \cdots, F_k(x_n)\}$, and the sender gets the key $k$. We recall the mp-OPRF functionality $\mathcal{F}_{\text{mp-OPRF}}$ in Figure 7.

**Parameters:** A PRF $F$. Two parties: $\mathcal{S}$ and $\mathcal{R}$.
**Functionality:**

1. Wait for input $\{x_1, \cdots, x_n\}$ from $\mathcal{R}$.
2. Sample a random PRF key $k$ and give it to $\mathcal{S}$. Give $\{F_k(x_1), \cdots, F_k(x_n)\}$ to $\mathcal{R}$.

Fig. 7: mp-OPRF functionality $\mathcal{F}_{\text{mp-OPRF}}$

**Permute + Share.** We recall the $\mathsf{Permute} + \mathsf{Share}$ functionality $\mathcal{F}_{\mathrm{PS}}$ [MS13, CGP20] in Figure 8. Roughly speaking, in the $\mathsf{Permute} + \mathsf{Share}$ protocol, $P_0$ inputs a set $X = \{x_1, \cdots, x_n\}$ of size $n$ and $P_1$ chooses a permutation $\pi$ on $n$ items. The result is that $P_0$ learns the shuffled shares $\{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ and $P_1$ learns the other shuffled shares $\{s'_{\pi(1)}, \cdots, s'_{\pi(n)}\}$, where $x_{\pi(i)} = s_{\pi(i)} \oplus s'_{\pi(i)}$, $i \in [n]$.

## 4 The Basic PSU Protocol

We describe our basic PSU protocol in Figure 9 as a strawman protocol. In this protocol, if $r + f(x) \neq r$, the receiver can get $f(x)$ which leaks some information of $x \notin Y$, but this leakage does not cause any harm to the PSU, since the PSU protocol releases that value at last. We prove its semi-honest security in the following theorem.

**Parameters:** Two parties: $P_0$ and $P_1$. Set size $n$ for $P_0$.
**Functionality:**

1. Wait for input $X = \{x_1, \cdots, x_n\}$ from $P_0$, abort if $|X| \neq n$. Wait for input a permutation $\pi$ from $P_1$, abort if $\pi$ is not a permutation on $n$ items.
2. Give output shuffled shares $\{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ to $P_0$, and another shuffled shares $\{s'_{\pi(1)}, \cdots, s'_{\pi(n)}\}$ to $P_1$, where $x_{\pi(i)} = s_{\pi(i)} \oplus s'_{\pi(i)}$, $i \in [n]$.

Fig. 8: Permute + Share functionality $\mathcal{F}_{\mathrm{PS}}$

**Input**: The sender inputs set $X$ of size $m = |X|$ and the receiver inputs set $Y$ of size $n = |Y|$. $m$ and $n$ are public.
**Output**: The receiver outputs $X \cup Y$. The sender outputs $\perp$.

1. [**Setup**] $\mathcal{S}$ generates a public-secret key pair and keeps the secret key itself.
2. [**Set encryption**] $\mathcal{S}$ encrypts each item $x_i \in X$, $c_i = \mathrm{FHE}.\mathsf{Enc}(x_i), i \in [m]$ and sends $(c_1, \cdots, c_m)$ to $\mathcal{R}$.
3. [**Computation**] For each $c_i$, $\mathcal{R}$
   (a) samples a random non-zero value $r_i$;
   (b) homomorphically computes $c'_i = \mathrm{FHE}.\mathsf{Enc}\ (f(x_i) + r_i)$, where $f(x) = \mathit{\Pi}_{y_i \in Y}(x - y_i)$.
   (c) sends $c'_i, i \in [m]$ to $\mathcal{S}$.
4. [**Decryption**] $\mathcal{S}$ decrypts $c'_i, i \in [m]$ to $r'_i = f(x_i) + r_i$ and sends them to $\mathcal{R}$.
5. [**Output**] $\mathcal{R}$ checks all plaintexts and sets a bit vector $\mathbf{b} = [b_i]$, $i \in [m]$. If $r'_i = r_i$, it sets $b_i = 0$, otherwise, sets $b_i = 1$. Then, both parties invoke OT protocol, $\mathcal{R}$ inputs the bit $b_i$ and $\mathcal{S}$ inputs $(\perp, x_i)$, $i \in [m]$. For all $i \in [m]$, $\mathcal{R}$ gets $x_i$, if $b_i = 1$, else gets $\perp$. Finally, $\mathcal{R}$ outputs $X \cup Y$.

Fig. 9: Basic PSU protocol

**Theorem 1.** *The PSU protocol described in Figure 9 is secure in the $\mathcal{F}_{OT}$-hybrid model, in the presence of semi-honest adversaries, provided that the fully homomorphic encryption scheme is IND-CPA secure with circuit privacy.*

*Proof.* We construct $\mathrm{Sim}_{\mathcal{S}}$ and $\mathrm{Sim}_{\mathcal{R}}$ to simulate the views of corrupt $\mathcal{S}$ and corrupt $\mathcal{R}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

**Corrupt Sender.** $\mathrm{Sim}_{\mathcal{S}}(X)$ simulates the view of corrupt $\mathcal{S}$ as follows: It encrypts $m$ random values. Then, it invokes $\mathrm{Sim}^{\mathcal{S}}_{\mathrm{OT}}(\perp, x_i)$, $i \in [m]$ and appends the output to the view. Now we argue that the view output by $\mathrm{Sim}_{\mathcal{S}}$ is indistinguishable from the real one. The plaintexts are randomized in the real view which is indistinguishable from the random values in the simulated view. The FHE satisfies the circuit privacy which hides the computational circuit in step 3. The view produced by the underlying OT simulator is indistinguishable from the real view. Thus, the simulation is indistinguishable from the real view.

**Corrupt Receiver.** $\mathrm{Sim}_{\mathcal{R}}(Y, X \cup Y)$ simulates the view of corrupt $\mathcal{R}$ as follows: It simulates the ciphertexts by encrypting $m$ random values. $\mathrm{Sim}_{\mathcal{R}}$ sets $\hat{X} = (X \cup Y) \backslash Y$ and pads $\hat{X}$ with $\perp$ into $m$ items and permutes all items randomly. It computes the polynomial $f(y) = \mathit{\Pi}_{y_i \in Y}(y - y_i)$ and the random values $\mathbf{r} = [r_i]$, $i \in [m]$ used to randomize the polynomial. Then, for $\hat{x}_i \neq \perp$, $\mathrm{Sim}_{\mathcal{R}}$ defines $r'_i := f(\hat{x}_i) + r_i$, else, it defines $r'_i := r_i$, and appends $\mathbf{r'} = [r'_i]$,$i \in [m]$ to the view. If $\hat{x}_i = \perp$, it sets $b_i = 0$, else, $b_i = 1$. Then $\mathrm{Sim}_{\mathcal{R}}$ invokes $\mathrm{Sim}^{\mathcal{R}}_{\mathrm{OT}}(b_i, \hat{x}_i)$ for $i \in [m]$ and appends the output to the view.

We argue that the outputs of $\mathrm{Sim}_{\mathcal{R}}$ are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:
$\mathsf{Hyb}_0$: $\mathcal{R}$'s view in the real protocol.
$\mathsf{Hyb}_1$: Same as $\mathsf{Hyb}_0$ except that the ciphertexts in the step 2 are replaced by encrypting $m$ random values generated by $\mathrm{Sim}_{\mathcal{R}}$. Since the fully homomorphic encryption scheme is IND-CPA secure, the above simulation is indistinguishable from the real view.

$\mathsf{Hyb}_2$: Same as $\mathsf{Hyb}_1$ except that $\text{Sim}_{\mathcal{R}}$ runs the $\mathcal{F}_{\text{OT}}$ simulator to produce the simulated view for $\mathcal{R}$. The security of OT protocol guarantees the view in simulation is computationally indistinguishable from the view in the real protocol. The hybrid is the view output by $\text{Sim}_{\mathcal{R}}$.

## 5 Permuted Matrix Private Equality Test

We give two efficient constructions of pm-PEQT in the semi-honest model. The functionality is specified in Figure 5.

### 5.1 pm-PEQT from Permute + Share and mp-OPRF

The first construction of pm-PEQT is based on the Permute + Share [CGP20] and mp-OPRF [CM20] as described in Figure 10. For notational convenience, we use pm-PEQT$_{\text{sym}}$ to denote pm-PEQT based on the Permute + Share and mp-OPRF, since the main computational costs of this protocols are symmetric-key operations.

**Input**: The receiver inputs a matrix $\mathbf{R}_{\alpha\times m}$. The sender inputs a matrix $\mathbf{R}'_{\alpha\times m}$, and a permutation $\pi = (\pi_c, \pi_r)$, where $\pi_c$ (over $[m]$) and $\pi_r$ (over $[\alpha]$) are two sub-permutations for columns and rows.
**Output**: The receiver outputs a bit matrix $\mathbf{B}$. The sender outputs $\perp$.

1. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal Permute + Share functionality $\mathcal{F}_{\text{PS}}$ twice. First, both parties permute and share the columns of $\mathbf{R}$, where each column of $\mathbf{R}$ can be seen as an item. $\mathcal{R}$ inputs each column of $\mathbf{R}$ and $\mathcal{S}$ inputs the permutation $\pi_c$. As a result, $\mathcal{R}$ gets $\mathbf{S}_{\pi_c} = [s_{\pi_c(ij)}]$ and $\mathcal{S}$ gets $\mathbf{S}'_{\pi_c} = [s'_{\pi_c(ij)}]$, where $s_{\pi_c(ij)} \oplus s'_{\pi_c(ij)} = r_{\pi_c(ij)}$. Then, both parties permute and share the rows of $\mathbf{S}_{\pi_c}$, where each row of $\mathbf{S}_{\pi_c}$ can be seen as an item. $\mathcal{R}$ inputs each row of $\mathbf{S}_{\pi_c}$ and $\mathcal{S}$ inputs the permutation $\pi_r$. As a result, $\mathcal{R}$ gets $\mathbf{S}_{\pi_r} = [s_{\pi_r(ij)}]$ and $\mathcal{S}$ gets $\mathbf{S}'_{\pi_r} = [s'_{\pi_r(ij)}]$, where $s_{\pi_r(ij)} \oplus s'_{\pi_r(ij)} = s_{\pi_c(ij)}$. Finally, $\mathcal{R}$ defines the shuffled matrix shares $\mathbf{S}_\pi = \mathbf{S}_{\pi_r}$ and $\mathcal{S}$ defines the shuffled matrix shares $\mathbf{S}'_\pi = \pi_r(\mathbf{S}'_{\pi_c}) \oplus \mathbf{S}'_{\pi_r}$, where $s_{\pi(ij)} \oplus s'_{\pi(ij)} = r_{\pi(ij)}$, $i \in [\alpha], j \in [m]$.
2. Both parties invoke mp-OPRF functionality $\mathcal{F}_{\text{mp-OPRF}}$. $\mathcal{R}$ inputs shuffled shares $\mathbf{S}_\pi$ and obtains the outputs $F_k(s_{\pi(ij)})$, $i \in [\alpha], j \in [m]$. $\mathcal{S}$ obtains the key $k$.
3. $\mathcal{S}$ computes $F_k(r'_{\pi(ij)} \oplus s'_{\pi(ij)})$, $i \in [\alpha], j \in [m]$ and sends them to $\mathcal{R}$.
4. $\mathcal{R}$ sets $b_{ij} = 1$, if $F_k(s_{\pi(ij)}) = F_k(r'_{\pi(ij)} \oplus s'_{\pi(ij)})$, else, $b_{ij} = 0$, and gets a bit matrix $\mathbf{B} = [b_{ij}]$, $i \in [\alpha], j \in [m]$.

Fig. 10: pm-PEQT from Permute + Share and mp-OPRF

**Theorem 2.** *The construction of Figure 10 securely implements functionality $\mathcal{F}_{\textit{pm-PEQT}}$ in the ($\mathcal{F}_{\textit{PS}}$, $\mathcal{F}_{\textit{mp-OPRF}}$)-hybrid model, in the presence of semi-honest adversaries.*

*Proof.* We exhibit simulators $\text{Sim}_{\mathcal{R}}$ and $\text{Sim}_{\mathcal{S}}$ for simulating corrupt $\mathcal{R}$ and $\mathcal{S}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

**Corrupt Sender.** $\text{Sim}_{\mathcal{S}}(\mathbf{R}', \pi = (\pi_c, \pi_r))$ simulates the view of corrupt $\mathcal{S}$ as follows: $\text{Sim}_{\mathcal{S}}$ randomly chooses $\mathbf{S}'_{\pi_c}$ and invokes $\text{Sim}^{\mathcal{S}}_{\text{PS}}(\pi_c, \mathbf{S}'_{\pi_c})$ and appends the output to the view. $\text{Sim}_{\mathcal{S}}$ randomly chooses $\mathbf{S}'_{\pi_r}$ and invokes $\text{Sim}^{\mathcal{S}}_{\text{PS}}(\pi_r, \mathbf{S}'_{\pi_r})$ and appends the output to the view. Then, $\text{Sim}_{\mathcal{S}}$ randomly selects a key $k$ of PRF and invokes $\text{Sim}^{\mathcal{S}}_{\text{mp-OPRF}}(k)$ and appends the output to the view.

We argue that the outputs of $\text{Sim}_{\mathcal{S}}$ are indistinguishable from the real view of $\mathcal{S}$ by the following hybrids:
$\mathsf{Hyb}_0$: $\mathcal{S}$'s view in the real protocol.
$\mathsf{Hyb}_1$: Same as $\mathsf{Hyb}_0$ except that the output of $\mathcal{F}_{\text{PS}}$ is replaced by $\mathbf{S}'_{\pi_c}$, $\mathbf{S}'_{\pi_r}$ chosen by $\text{Sim}_{\mathcal{S}}$, and $\text{Sim}_{\mathcal{S}}$ runs the $\mathcal{F}_{\text{PS}}$ simulator to produce the simulated view for $\mathcal{S}$. The security of Permute + Share guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

$\mathsf{Hyb}_2$: Same as $\mathsf{Hyb}_1$ except that the output key of $\mathcal{F}_{\text{mp-OPRF}}$ is replaced by the $k$ chosen by $\text{Sim}_{\mathcal{S}}$, and $\text{Sim}_{\mathcal{S}}$ runs the $\mathcal{F}_{\text{mp-OPRF}}$ simulator to produce the simulated view for $\mathcal{S}$. The security of mp-OPRF guarantees the view in simulation is computationally indistinguishable from the view in the real protocol. The hybrid is the view output by $\text{Sim}_{\mathcal{S}}$.

**Corrupt Receiver.** $\text{Sim}_{\mathcal{R}}(\mathbf{R}, \mathbf{B} = [b_{ij}])$ simulates the view of corrupt $\mathcal{R}$ as follows: $\text{Sim}_{\mathcal{R}}$ chooses $\mathbf{S}_{\pi_c}$ and invokes $\text{Sim}^{\mathcal{R}}_{\text{PS}}(\mathbf{R}, \mathbf{S}_{\pi_c})$ and appends the output to the view. $\text{Sim}_{\mathcal{R}}$ chooses $\mathbf{S}_{\pi_r}$ and invokes $\text{Sim}^{\mathcal{R}}_{\text{PS}}(\mathbf{S}_{\pi_c}, \mathbf{S}_{\pi_r})$ and appends the output to the view. $\text{Sim}_{\mathcal{R}}$ randomly selects $u_{ij}$, $i \in [\alpha]$, $j \in [m]$ and invokes $\text{Sim}^{\mathcal{R}}_{\text{mp-OPRF}}(u_{ij})$ and appends the output to the view. Finally, for all $i \in [\alpha]$, $j \in [m]$, $\text{Sim}_{\mathcal{R}}$ sets $v_{ij} = u_{ij}$ if $b_{ij} = 1$, else, it chooses $v_{ij}$ randomly and appends all $v_{ij}$ to the view.

The view generated by $\text{Sim}_{\mathcal{R}}$ is indistinguishable from a real view of $\mathcal{R}$ by the following hybrids:
$\mathsf{Hyb}_0$: $\mathcal{R}$'s view in the real protocol.
$\mathsf{Hyb}_1$: Same as $\mathsf{Hyb}_0$ except that the output of $\mathcal{F}_{\text{PS}}$ is replaced by $\mathbf{S}_{\pi_c}$, $\mathbf{S}_{\pi_r}$ chosen by $\text{Sim}_{\mathcal{R}}$, and $\text{Sim}_{\mathcal{R}}$ runs the $\mathcal{F}_{\text{PS}}$ simulator to produce the simulated view for $\mathcal{R}$. The security of $\mathsf{Permute} + \mathsf{Share}$ guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.
$\mathsf{Hyb}_2$: Same as $\mathsf{Hyb}_1$ except that the output PRF values of $\mathcal{F}_{\text{mp-OPRF}}$ is replaced by $u_{ij}$, $i \in [\alpha]$, $j \in [m]$, and all PRF values in the last step is replaced by the $v_{ij}$, chosen by $\text{Sim}_{\mathcal{R}}$ randomly, and $\text{Sim}_{\mathcal{R}}$ runs the $\mathcal{F}_{\text{mp-OPRF}}$ simulator to produce the simulated view for $\mathcal{R}$. The security of mp-OPRF and PRF guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

### 5.2 pm-PEQT based on DDH

The second construction of pm-PEQT is based on DDH as described in Figure 11. For notational convenience, we use $\text{pm-PEQT}_{\text{pub}}$ to denote DDH-based pm-PEQT, since the main computational costs are public-key operations.

**Input**: The receiver inputs a matrix $\mathbf{R}_{\alpha\times m}$. The sender inputs a matrix $\mathbf{R}'_{\alpha\times m}$, and a permutation $\pi = (\pi_c, \pi_r)$ where $\pi_c$ (over $[m]$) and $\pi_r$ (over $[\alpha]$) are two sub-permutation for columns and rows. $\mathbb{G}$ is a cyclic group with order $q$.
**Output**: The receiver outputs a bit matrix $\mathbf{B}$. The sender outputs $\perp$.

1. $\mathcal{R}$ chooses a random value $a \leftarrow \mathbb{Z}_q$ and computes $v_{ij} = H(r_{ij})^a$ for all $i \in [\alpha]$, $j \in [m]$, where $H(\cdot)$ denotes hash functions which output the elements of group $\mathbb{G}$. Let $\mathbf{V} = [v_{ij}]$, $i \in [\alpha]$, $j \in [m]$. $\mathcal{R}$ sends $\mathbf{V}$ to $\mathcal{S}$.
2. $\mathcal{S}$ chooses a random value $b \leftarrow \mathbb{Z}_q$ and computes $v'_{ij} = H(r'_{ij})^b$ for all $i \in [\alpha]$, $j \in [m]$, where $H(\cdot)$ denotes hash functions which output the elements of group $\mathbb{G}$. Let $\mathbf{V}' = [v'_{ij}]$, $i \in [\alpha]$, $j \in [m]$. Then $\mathcal{S}$ computes $v''_{ij} = (v_{ij})^b$ and lets $\mathbf{V}'' = [v''_{ij}]$. $\mathcal{S}$ shuffles $\mathbf{V}''$ and $\mathbf{V}'$ by same permutation $\pi = (\pi_c, \pi_r)$ and gets $\mathbf{V}''_\pi = \pi(\mathbf{V}'')$, $\mathbf{V}'_\pi = \pi(\mathbf{V}')$, where $v''_{\pi(ij)} = \pi(v''_{ij})$, $v'_{\pi(ij)} = \pi(v'_{ij})$. $\mathcal{S}$ sends $\mathbf{V}''_\pi$, $\mathbf{V}'_\pi$ to $\mathcal{R}$.
3. For $i$-th row and $j$-column in $\mathbf{V}''_\pi$ and $\mathbf{V}'_\pi$, if $v''_{\pi(ij)} = v'^{a}_{\pi(ij)}$, $\mathcal{R}$ sets $b_{ij} = 1$, else, $b_{ij} = 0$. $\mathcal{R}$ sets a bit matrix $\mathbf{B} = [b_{ij}]$, $i \in [\alpha]$, $j \in [m]$.

Fig. 11: Instantiation of pm-PEQT based on DDH

**Theorem 3.** *The construction of Figure 11 securely implements functionality $\mathcal{F}_{pm\text{-}PEQT}$ based on DDH in the random oracle model, in the presence of semi-honest adversaries.*

*Proof.* We exhibit simulators $\text{Sim}_{\mathcal{R}}$ and $\text{Sim}_{\mathcal{S}}$ for simulating corrupt $\mathcal{R}$ and corrupt $\mathcal{S}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

**Corrupt Sender.** $\text{Sim}_{\mathcal{S}}(\mathbf{R}', \pi = (\pi_c, \pi_r))$ simulates the view of corrupt $\mathcal{S}$ as follows: It chooses random group elements $v_{ij}$, $i \in [\alpha]$, $j \in [m]$ to simulate the view. We argue that the outputs of $\text{Sim}_{\mathcal{S}}$ are indistinguishable from the real view of $\mathcal{S}$ by the following hybrids:

$\mathsf{Hyb}_0$: $\mathcal{S}$'s view in the real protocol consists of $H(r_{ij})^a$, $i \in [\alpha]$, $j \in [m]$, where $a \leftarrow \mathbb{Z}_q$.
$\mathsf{Hyb}_1$: Same as $\mathsf{Hyb}_0$ except that $\mathrm{Sim}_{\mathcal{S}}$ chooses random group elements $v_{ij}$, $i \in [\alpha]$, $j \in [m]$ instead of $H(r_{ij})^a$, $i \in [\alpha]$, $j \in [m]$, where $a \leftarrow \mathbb{Z}_q$. The hybrid is the view output by $\mathrm{Sim}_{\mathcal{S}}$.

We argue that the views in $\mathsf{Hyb}_0$ and $\mathsf{Hyb}_1$ are computationally indistinguishable. Let $\mathcal{A}$ be a probabilistic polynomial-time (PPT) adversary against the DDH assumption. Given the DDH challenge $g^x, g^{y_{ij}}, g^{z_{ij}}$, where $x, y_{ij} \leftarrow \mathbb{Z}_q$, $\mathcal{A}$ is asked to distinguish if $z_{ij} = x \cdot y_{ij}$ or random values. $\mathcal{A}$ implicitly sets $a = x$, and simulates (with the knowledge of $\mathbf{R}$) the view as below:

- RO queries: $\mathrm{Sim}_{\mathcal{S}}$ honestly emulates random oracle (RO) $H$. For queries $r_{ij}$, if $r_{ij} \notin \mathbf{R}$, it picks a random group element to assign $H(r_{ij})$, otherwise, it assigns $H(r_{ij}) = g^{y_{ij}}$.
- Outputs $g^{z_{ij}}$, $i \in [\alpha]$, $j \in [m]$.

Clearly, if $z_{ij} = x \cdot y_{ij}$, $\mathcal{A}$ simulates $\mathsf{Hyb}_0$. Else, it simulates $\mathsf{Hyb}_1$ (without the knowledge of $\mathbf{R}$), because it responds to all RO queries with random group elements without knowing that the inputs belong to $\mathbf{R}$ or not. Therefore, the outputs of $\mathrm{Sim}_{\mathcal{S}}$ are computationally indistinguishable from the real view based on the DDH assumption.

**Corrupt Receiver.** $\mathrm{Sim}_{\mathcal{R}}(\mathbf{R},\ \mathbf{B} = [b_{ij}])$ simulates the view of corrupt $\mathcal{R}$ as follows: $\mathrm{Sim}_{\mathcal{R}}$ chooses $a \leftarrow \mathbb{Z}_q$ randomly and simulates the first round message as real protocol. For $b_{ij} = 0$, $i \in [\alpha]$, $j \in [m]$, it chooses random group elements $v_{ij}$ and $u_{ij}$ to simulate the view. For $b_{ij} \neq 0$, $i \in [\alpha]$, $j \in [m]$, it chooses random group elements $v_{ij}$ and sets $u_{ij} = v_{ij}^a$ to simulate the view.

We argue that the outputs of $\mathrm{Sim}_{\mathcal{R}}$ are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:
$\mathsf{Hyb}_0$: $\mathcal{R}$'s view in the real protocol consists of $H(r'_{\pi(ij)})^b$ and $H(r_{\pi(ij)})^{ab}$, $i \in [\alpha]$, $j \in [m]$, where $a$, $b \leftarrow \mathbb{Z}_q$.
$\mathsf{Hyb}_1$: Same as $\mathsf{Hyb}_0$ except that for $b_{ij} = 0$, that is $r_{\pi(ij)} \neq r'_{\pi(ij)}$, $\mathrm{Sim}_{\mathcal{R}}$ chooses random group elements $v_{ij}$ and $u_{ij}$ instead of $H(r'_{\pi(ij)})^b$ and $H(r_{\pi(ij)})^{ab}$.
$\mathsf{Hyb}_2$: Same as $\mathsf{Hyb}_1$ except that for $b_{ij} = 1$, that is $r_{\pi(ij)} = r'_{\pi(ij)}$, $\mathrm{Sim}_{\mathcal{R}}$ chooses random group elements $v_{ij}$ and sets $u_{ij} = v_{ij}^a$, $i \in [\alpha]$, $j \in [m]$ instead of $H(r'_{\pi(ij)})^b$ and $H(r_{\pi(ij)})^{ab}$. The hybrid is the view output by $\mathrm{Sim}_{\mathcal{R}}$.

We argue that the view in $\mathsf{Hyb}_0$ and $\mathsf{Hyb}_1$ are computationally indistinguishable based on the DDH assumption. Given the DDH challenge $g^x$, $g^{y_{ij}}$, $g^{y'_{ij}}$, $g^{z_{ij}}$, $g^{z'_{ij}}$, where $x$, $y_{ij}$, $y'_{ij} \leftarrow \mathbb{Z}_q$, $\mathcal{A}$ is asked to distinguish if $z_{ij} = x \cdot y_{ij}$, $z'_{ij} = x \cdot y'_{ij}$ or random values. $\mathcal{A}$ implicitly sets $b = x$, and simulates (with the knowledge of $\mathbf{R}'$ and $\pi$) the view as below:

- RO queries: $\mathrm{Sim}_{\mathcal{R}}$ honestly emulates random oracle $H$. For queries $r_{ij}$ and $r'_{ij}$, if $r_{ij} \notin \mathbf{R}$, $r'_{ij} \notin \mathbf{R}'$, it assigns $H(r_{\pi(ij)})$, $H(r'_{\pi(ij)})$ with random group elements. If $r_{ij} \in \mathbf{R}$, $r'_{ij} \in \mathbf{R}'$, it assigns $H(r_{ij}) = g^{a^{-1} y_{ij}}$, $H(r'_{ij}) = g^{y'_{ij}}$.
- Outputs $g^{z_{ij}}$, $g^{z'_{ij}}$.

Clearly, if $z_{ij} = x \cdot y_{ij}$, $z'_{ij} = x \cdot y'_{ij}$, $\mathcal{A}$ simulates $\mathsf{Hyb}_0$. Else, it simulates $\mathsf{Hyb}_1$. In the $\mathsf{Hyb}_1$, $\mathrm{Sim}_{\mathcal{R}}$ needs not to know the $\mathbf{R}'$ and $\pi$ in these positions with $b_{ij} = 0$, because in these positions, it responds to all random oracle queries with random group elements.

We argue that the view in $\mathsf{Hyb}_1$ and $\mathsf{Hyb}_2$ are computationally indistinguishable based on the DDH assumption. Given the DDH challenge $g^x, g^{y_{ij}}, g^{z_{ij}}$ where $x, y_{ij} \leftarrow \mathbb{Z}_q$, $\mathcal{A}$ is asked to distinguish if $z_{ij} = x \cdot y_{ij}$ or random values. $\mathcal{A}$ implicitly sets $b = x$, and simulates (with the knowledge of $\mathbf{R}'$ and $\pi$ for all positions with $b_{ij} = 1$) the view as below:

- RO queries: $\mathrm{Sim}_{\mathcal{R}}$ honestly emulates random oracle $H$. For queries $r_{ij}$, $r'_{ij}$, if $r_{ij} \notin \mathbf{R}$, $r'_{ij} \notin \mathbf{R}'$, it assigns $H(r_{ij})$, $H(r'_{ij})$ with random group elements. If $r_{ij} = r'_{ij} \in \mathbf{R}$, it assigns $H(r_{ij}) = H(r'_{ij}) = g^{y_{ij}}$.
- Outputs $g^{a \cdot z_{ij}}$, $g^{z_{ij}}$.

Clearly, if $z_{ij} = x \cdot y_{ij}$, $\mathcal{A}$ simulates $\mathsf{Hyb}_1$. Else, it simulates $\mathsf{Hyb}_2$. In the $\mathsf{Hyb}_2$, $\mathrm{Sim}_{\mathcal{R}}$ needs not to know the $\mathbf{R}'$ and $\pi$ in these positions with $b_{ij} = 1$, because in these positions, it responds to all random oracle queries with random group elements. Therefore, the outputs of $\mathrm{Sim}_{\mathcal{R}}$ are computationally indistinguishable from the real view based on the DDH assumption.

**Comparisons of two pm-PEQT instantiations.** pm-PEQT$_{\text{pub}}$ based on DDH assumption is not post-quantum secure, while pm-PEQT$_{\text{sym}}$ based on symmetric cryptographic primitives is potentially post-quantum secure. The comprehensive comparison of the two pm-PEQT instantiations is given in Table 2.

*Remark.* We note that our pm-PEQT can be generalized to multi-query private equality test with permutation, which in turn can be built from permuted OPRF [CZZD22] in a general manner.

| Protocols | Communication | Computation | Post-quantum security |
|---|---|---|---|
| pm-PEQT$_{\text{pub}}$ | $O(m)$ | $O(m)$ | × |
| pm-PEQT$_{\text{sym}}$ | $O(m \log m)$ | $O(m \log m)$ | √ |

Table 2: Comparisons of two pm-PEQT instantiations. $m$ denotes the number of elements in the matrix.

## 6 Full PSU Protocol

In this section, we detail our full PSU protocol in Figure 12. The main optimization idea of our protocol is as follows.

**Offline/online.** Following [CLR17, CHLR18, CMdG$^+$21], the pre-processing of the receiver in our PSU can be done entirely offline without involving the sender. Specifically, given an upper bound on the sender's set size, the receiver can locally choose parameters and perform the pre-processing. Upon learning the sender's actual set size, the receiver sends the parameters to the sender, and the sender pads the same dummy items like $\perp$ which are known two both parties.

**OPRF pre-processing.** We use the dual structure of [CLR17] in our PSU and prove the security based on FHE with circuit privacy following [CLR17]. This leads to performing a noise flooding operation on the result ciphertexts, as is necessary in [CLR17]. Following [CHLR18, CMdG$^+$21], we can use an OPRF to compute the items on both sides before engaging in the PSU, which prevents the sender from learning anything about the original items. Thus, our PSU can be proven secure without circuit privacy and utilize more efficient FHE parameters as [CHLR18, CMdG$^+$21], which improves our performance and adds flexibility to the parametrization.

**Low circuit depth.** The steps 1-6 in our full PSU protocol 12 can be seen as the dual structure of unbalanced PSI [CLR17, CHLR18, CMdG$^+$21]. Therefore, the optimizations used in [CLR17, CHLR18, CMdG$^+$21], such as batching, windowing, partitioning, and modulus switching, are suitable for the steps 1-6 to significantly reduce the depth of the homomorphic circuit. We review the optimizations in appendix A.

**Communication.** We show our communication complexity as follow. In step 1-6, the communication requires $O(m \log n)$; the communication of the pm-PEQT requires $O(m \log m)$ (based on $\mathsf{Permute} + \mathsf{Share}$ and mp-OPRF) or $O(m)$ (based on DDH), since we omit the parameter $\alpha$ which is used to make trade-off between the computation and communication of our PSU; the communication of OT requires $O(m)$. In summary, the communication of our PSU is $O(m \log n)$.

**Correctness.** The correctness of our PSU protocol is conditioned on the hashing succeeding, which happens with overwhelming probability $1 - 2^{-\lambda}$.

**Theorem 4.** *The protocol in Figure 12, is a secure protocol for $\mathcal{F}_{PSU}$ in the ($\mathcal{F}_{pm\text{-}PEQT}$, $\mathcal{F}_{OT}$)-hybrid model, in the presence of semi-honest adversaries, provided that the fully homomorphic encryption scheme is IND-CPA secure with circuit privacy.*

**Input**: The receiver inputs set $Y \subset \{0,1\}^*$ of size $n = |Y|$ and the sender inputs set $X \subset \{0,1\}^*$ of size $m = |X|$, where $m \ll n$. $m$ and $n$ are public.
**Output**: The receiver outputs $X \cup Y$. The sender outputs $\perp$.

1. [**Setup**] $\mathcal{R}$ and $\mathcal{S}$ agree on the hashing, FHE, mp-PEQT and OT parameters.
2. [**Hashing**] $\mathcal{S}$ hashes $X$ into table $X_c$ by Cuckoo hash, where $X_c$ consists of $m_c$ bins and each bin has only one item. $\mathcal{R}$ uses the same hash function to hash $Y$ into table $\mathbf{Y}_{B\times m_c}$, where $\mathbf{Y}_{B\times m_c}$ consists of $m_c$ bins and each bin has $B$ items.
3. [**Pre-process Y**]
   (a) [Partitioning] $\mathcal{R}$ partitions $\mathbf{Y}_{B\times m_c}$ by rows into $\alpha$ subtables $\mathbf{Y}_1, \cdots, \mathbf{Y}_\alpha$. Each subtable has $B' = B/\alpha$ rows and $m$ columns.
   (b) [Computing coefficients] For $j$-th columns of $i$-th subtable $\mathbf{y}_{i,j} = [y_{i,j,1}, \cdots, y_{i,j,B'}]^T$, $i \in [\alpha], j \in [m_c]$, $\mathcal{R}$ computes the coefficients of the polynomial $f_{i,j}(y) = \Pi_{k=1}^{B'}(y - y_{i,j,k}) = a'_{i,j,0} + a_{i,j,1}y + \cdots + a_{i,j,B'}y^{B'}$. $\mathcal{R}$ computes the coefficient matrix $\mathbf{A}$ as follows: $\mathcal{R}$ chooses a random matrix $\mathbf{R}_{\alpha\times m_c} = [r_{i,j}]$, and sets $j$-th columns of $i$-th subtable $\mathbf{A}_{i,j} = [a_{i,j,0}, a_{i,j,1}, \cdots, a_{i,j,B'}]^T$, $i \in [\alpha], j \in [m_c]$, where $a_{i,j,0} = a'_{i,j,0} + r_{i,j}$.
   (c) [Batching] For each subtable obtained from the previous step, $\mathcal{R}$ interprets each of its row as a vector of length $m_c$ with elements in $\mathbb{Z}_t$. Then $\mathcal{R}$ batches each vector into $\beta = m_c/\gamma$ plaintext polynomials. As a result, each row of $i$-th subtable $\mathbf{A}_i$ is transformed into $\beta$ polynomials denoted $\hat{\mathbf{A}}_{i,j}$, $i \in [\alpha]$, $j \in [\beta]$.
4. [**Encrypt X**]
   (a) [Batching] $\mathcal{S}$ interprets $X_c$ as a vector of length $m_c$ with items in $\mathbb{Z}_t$. It batches this vector into $\beta = m_c/\gamma$ plaintext polynomials $\hat{X}_1, \cdots, \hat{X}_\beta$.
   (b) [Windowing] For each batched plaintext polynomial $\hat{X}$, $\mathcal{S}$ computes the component-wise $i \cdot 2^j$-th powers $\hat{X}^{i\cdot 2^{lj}}$, for $i \in [2^l - 1]$ and $0 \le j \le \lceil \log_2(B')/l \rceil$.
   (c) [Encrypt] $\mathcal{S}$ uses FHE scheme to encrypt each such power, obtaining $\beta$ collections of ciphertexts $\mathbf{C}_j, j \in [\beta]$, and each collection consists of the ciphertexts $[c_{i,j}]$, $i \in [2^l - 1]$, $0 \le j \le \lceil \log_2(B')/l \rceil$. $\mathcal{S}$ sends these ciphertexts to $\mathcal{R}$.
5. [**Computation**]
   (a) [Homomorphically compute encryptions of all powers] For each collection $\mathbf{C}_j, j \in [\beta]$, $\mathcal{R}$ homomorphically computes encryptions of all powers $\mathbf{C}_j = [\mathbf{c}_{j,0}, \cdots, \mathbf{c}_{j,B'}]$, where $\mathbf{c}_{j,k}, 0 \le k \le B'$ is a ciphertext encrypted $\hat{X}_j^k$.
   (b) [Homomorphically evaluate the dot product] $\mathcal{R}$ homomorphically evaluates $\mathbf{C}'_{i,j} = \mathbf{C}_j \hat{\mathbf{A}}_{i,j}$, $i \in [\alpha]$, $j \in [\beta]$, performs modulus switching on $\mathbf{C}'_{i,j}$ to reduce sizes, and sends the ciphertexts to $\mathcal{S}$.
6. [**Decrypt**] $\mathcal{S}$ decrypts $\mathbf{C}'_{i,j}$ and concatenates the results into the matrix $\mathbf{R}'_{\alpha\times m_c}$.
7. [**pm-PEQT**] $\mathcal{R}$ inputs the matrix $\mathbf{R}_{\alpha\times m_c}$, and $\mathcal{S}$ inputs the permutation $\pi = (\pi_c, \pi_r)$ and the matrix $\mathbf{R}'_{\alpha\times m_c}$. Both parties invoke the pm-PEQT functionality. As a result, $\mathcal{R}$ gets a bit matrix $\mathbf{B}_{\alpha\times m_c}$, where $b_{ij} = 1$ if $r_{\pi(ij)} = r'_{\pi(ij)}$, and $r_{\pi(ij)} \ne r'_{\pi(ij)}$ otherwise.
8. [**Output**] $\mathcal{R}$ sets a bit vector $\mathbf{b} = [b_j]$, $j \in [m_c]$, where $b_j = 1$ if for all $i \in [\alpha]$, $b_{ij} = 0$ and $b_j = 0$ otherwise. Then, $\mathcal{R}$ and $\mathcal{S}$ invoke the OT functionality, in which $\mathcal{R}$ inputs $b_j$, $j \in [m_c]$ and $\mathcal{S}$ inputs $(\perp, X_c[\pi_c(j)])$. If $b_j = 1$, $\mathcal{R}$ gets $X_c[\pi_c(j)]$, else, it gets $\perp$. Finally, $\mathcal{R}$ outputs the set union $Y \cup \{X_c[\pi_c(j)]\}$, for all $j \in [m_c]$.

Fig. 12: Full PSU protocol

*Proof.* The correctness of our PSU protocol is conditioned on the hashing succeeding, which happens with overwhelming probability $1 - 2^{-\lambda}$.

For ease of exposition, we will assume that all parameters are fixed and public. We exhibit simulators $\mathsf{Sim}_{\mathcal{S}}$ and $\mathsf{Sim}_{\mathcal{R}}$ for simulating corrupt $\mathcal{S}$ and $\mathcal{R}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

**Corrupt Sender.** $\mathrm{Sim}_{\mathcal{S}}(X)$ simulates the view of corrupt $\mathcal{S}$ as follows. $\mathsf{Sim}_{\mathcal{S}}$ hashes $X$ into $X_c$ as the real protocol, and encrypts random values in place of the ciphertexts in step 5. Then it decrypts the ciphertexts as $\mathbf{R}'$ and chooses randomly permutation $\pi = (\pi_c, \pi_r)$. It invokes $\mathrm{Sim}^{\mathcal{S}}_{\text{pm-PEQT}}(\mathbf{R}', \pi)$ and $\mathrm{Sim}^{\mathcal{S}}_{\mathrm{OT}}(\perp, X_c[\pi_c(j)])$, $j \in [m_c]$ appends the output to the view. Now we argue that the view output by $\mathrm{Sim}_{\mathcal{S}}$ is indistinguishable from the real one. The plaintexts are randomized in the real view which is indistinguishable from the random values in the simulated view. The FHE satisfies the circuit privacy which hides the computational circuit. The views of the underlying pm-PEQT and OT simulator are indistinguishable. Thus, the simulation is indistinguishable from the real view.

**Corrupt Receiver.** $\mathsf{Sim}_{\mathcal{R}}\ (Y, X \cup Y)$ simulates the view of corrupt receiver as follows: $\mathrm{Sim}_{\mathcal{R}}$ encrypts random value in place of the ciphertexts in step 4. It chooses the random matrix $\mathbf{R}$ in step 3. $\mathsf{Sim}_{\mathcal{R}}$ computes $\hat{X} = (X \cup Y)\backslash Y$ and pads $\hat{X}$ with $\perp$ to $m_c$ items and permutes these items randomly. For all items in $\hat{X}$, if $\hat{x}_i \neq \perp$, it sets $b_i = 1$, else $b_i = 0$. And then it generates $\mathbf{B}_{\alpha \times m_c}$, for all columns $\mathbf{b}_i$, if $b_i = 1$, all items in $\mathbf{b}_i$ are set to 0, else, one random position in $\mathbf{b}_i$ is set to 1 and all other positions are set to 0. $\mathrm{Sim}_{\mathcal{R}}$ invokes $\mathrm{Sim}^{\mathcal{R}}_{\text{pm-PEQT}}(\mathbf{R}, \mathbf{B})$ appends the output to the view. Then, for all $i \in [m_c]$, it invokes $\mathrm{Sim}^{\mathcal{R}}_{\mathrm{OT}}(b_i, \hat{x}_i)$ and appends the output to the view.

The view generated by $\mathrm{Sim}_{\mathcal{R}}$ is indistinguishable from a real view of $\mathcal{R}$ by the following hybrids:
$\mathsf{Hyb}_0$: $\mathcal{R}$'s view in the real protocol.
$\mathsf{Hyb}_1$: Same as $\mathsf{Hyb}_0$ except that the ciphertexts are replaced by encrypting random values generated by $\mathrm{Sim}_{\mathcal{R}}$. Since the fully homomorphic encryption scheme is IND-CPA secure, the simulation is indistinguishable from the real view.
$\mathsf{Hyb}_2$: Same as $\mathsf{Hyb}_1$ except that the output of $\mathcal{F}_{\text{pm-PEQT}}$ is replaced by $\mathbf{B}$ generated by $\mathrm{Sim}_{\mathcal{R}}$, and $\mathrm{Sim}_{\mathcal{R}}$ runs the $\mathcal{F}_{\text{pm-PEQT}}$ simulator to produce the simulated view for $\mathcal{R}$. The security of the pm-PEQT protocol guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.
$\mathsf{Hyb}_3$: Same as $\mathsf{Hyb}_2$ except that $\mathrm{Sim}_{\mathcal{R}}$ runs the $\mathcal{F}_{\mathrm{OT}}$ simulator to produce the simulated view for $\mathcal{R}$. The security of OT protocol guarantees the view in simulation is computationally indistinguishable from the view in the real protocol. The hybrid is the view output by $\mathrm{Sim}_{\mathcal{R}}$.

## 7 Implementation and Performance

In this section, we experimentally evaluate our two PSU protocols:

- $\mathrm{PSU}_{\mathrm{sym}}$: PSU protocol based on FHE, OT, and pm-PEQT, where pm-PEQT is built from $\mathsf{Permute} + \mathsf{Share}$ and mp-OPRF.
- $\mathrm{PSU}_{\mathrm{pub}}$: PSU protocol based on FHE, OT, and DDH-based pm-PEQT.

We give our experimental environment at first, then compare our protocols with the state-of-the-art works in terms of communication and runtime in different network environments. Our source code is available upon request.

### 7.1 Experimental Setup

We run our experiments on a single Intel Core i7-11700 CPU @ 2.50GHz with 16 threads and 16GB of RAM. We simulate network latency and bandwidth by using the Linux `tc` command. Specifically, we consider the following LAN setting, where the two parties are connected via a local host with 10Gbps throughput, and a 0.2ms round-trip time (RTT). We also consider two WAN settings with 100Mbps and 10Mbps bandwidth, each with an 80ms RTT.

### 7.2 Implementation Details

We use the FHE scheme in [FV12], $\mathsf{Permute} + \mathsf{Share}$ in [MS13, CGP20], mp-OPRF in [CM20] and OT extension in [IKNP03]. For concrete analysis, we set the computational security parameter $\kappa = 128$ and the statistical security parameter $\lambda = 40$ following [JSZ$^+$22, ZCL$^+$23]. Our implementation is written in C++. The following libraries are used in our implementation.

- FHE: SEAL https://github.com/microsoft/SEAL
  and APSI https://github.com/microsoft/APSI
- $\mathsf{Permute} + \mathsf{Share}$: https://github.com/dujiajun/PSU
- mp-OPRF: https://github.com/peihanmiao/OPRF-PSI
- OT: https://github.com/osu-crypto/libOTe

### 7.3 Performance Comparisons

In this section, We compare our $\text{PSU}_{\text{sym}}$ and $\text{PSU}_{\text{pub}}$ with PSU [JSZ$^+$22] and PSU [ZCL$^+$23] in terms of runtime and communication, and the results are reported in Table 3 and Figure 13.

We stress that all reported costs are computed in the same environment. For comparisons of other works [JSZ$^+$22, ZCL$^+$23], we use the parameters recommended in their open-source code and fix the item length to 128-bit.

- PSU [JSZ$^+$22]: https://github.com/dujiajun/PSU
- PSU [ZCL$^+$23]: https://github.com/alibaba-edu/mpc4j

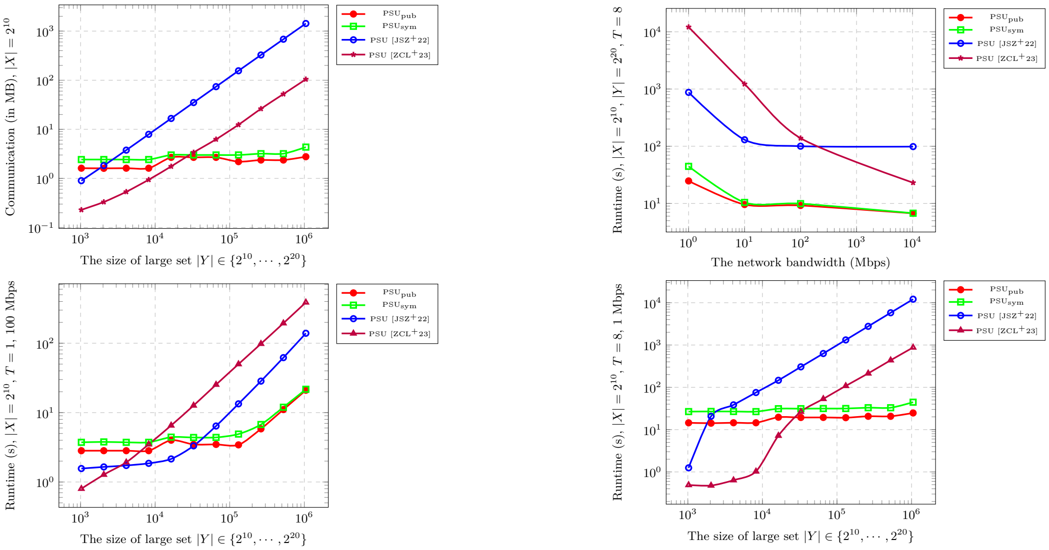

Fig. 13: Comparisons of communication (in MB) and runtime (in seconds) between PSU [JSZ$^+$22], PSU [ZCL$^+$23], $\text{PSU}_{\text{pub}}$ and $\text{PSU}_{\text{sym}}$. Both $x$ and $y$-axis are in log scale. The top left figure shows the communication cost increases as the large set size increases. The top right figure shows the runtime decreases as the bandwidth increases. The bottom two figures show the runtime increases as the large set size increases.

| Parameters | | Protocols | Comm. (MB) | Total running time (s) | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | 10Gbps | | | | 100Mbps | | | | 10Mbps | | | | 1Mbps | | | |
| $\|X\|$ | $\|Y\|$ | | | $T=1$ | $T=2$ | $T=4$ | $T=8$ | $T=1$ | $T=2$ | $T=4$ | $T=8$ | $T=1$ | $T=2$ | $T=4$ | $T=8$ | $T=1$ | $T=2$ | $T=4$ | $T=8$ |
| $2^{10}$ | $2^{10}$ | PSU [ZCL+23] | 0.229 | 0.63 | 0.36 | 0.29 | 0.24 | 0.8 | 0.593 | 0.538 | 0.492 | 0.795 | 0.589 | 0.53 | 0.497 | 0.798 | 0.59 | 0.528 | 0.491 |
| | | PSU [JSZ+22] | 0.899 | 0.047 | 0.041 | 0.042 | 0.057 | 1.562 | 1.476 | 1.397 | 1.245 | 1.565 | 1.48 | 1.405 | 1.247 | 1.565 | 1.425 | 1.399 | 1.249 |
| | | Our $\text{PSU}_{\text{pub}}$ | 1.614 | 0.916 | 0.716 | 0.615 | 0.565 | 2.824 | 2.624 | 2.525 | 2.475 | 2.825 | 2.625 | 2.524 | 2.524 | 6.392 | 14.482 | 14.084 | 14.489 |
| | | Our $\text{PSU}_{\text{sym}}$ | 2.423 | 0.866 | 0.716 | 0.666 | 0.665 | 3.731 | 3.48 | 3.429 | 3.378 | 3.731 | 3.479 | 3.329 | 3.278 | 22.369 | 22.474 | 24.676 | 26.681 |
| | $2^{11}$ | PSU [ZCL+23] | 0.331 | 1.12 | 0.486 | 0.379 | 0.306 | 1.276 | 0.648 | 0.537 | 0.465 | 1.188 | 0.652 | 0.542 | 0.471 | 1.178 | 0.643 | 0.537 | 0.477 |
| | | PSU [JSZ+22] | 1.823 | 0.056 | 0.046 | 0.046 | 0.06 | 1.649 | 1.686 | 1.483 | 1.33 | 1.566 | 1.686 | 1.481 | 1.366 | 13.983 | 17.065 | 18.322 | 20.684 |
| | | Our $\text{PSU}_{\text{pub}}$ | 1.604 | 0.916 | 0.716 | 0.616 | 0.565 | 2.826 | 2.625 | 2.525 | 2.526 | 2.825 | 2.625 | 2.525 | 2.525 | 14.183 | 14.432 | 14.186 | 14.185 |
| | | Our $\text{PSU}_{\text{sym}}$ | 2.423 | 0.866 | 0.716 | 0.666 | 0.615 | 3.781 | 3.48 | 3.42 | 3.38 | 3.731 | 3.48 | 3.429 | 3.379 | 22.367 | 22.768 | 24.333 | 26.893 |
| | $2^{12}$ | PSU [ZCL+23] | 0.534 | 1.723 | 0.755 | 0.584 | 0.475 | 1.921 | 0.918 | 0.737 | 0.643 | 1.918 | 0.927 | 0.747 | 0.648 | 1.916 | 0.909 | 1.189 | 0.638 |
| | | PSU [JSZ+22] | 3.766 | 0.073 | 0.06 | 0.053 | 0.066 | 1.73 | 2.05 | 1.809 | 1.571 | 3.064 | 3.121 | 3.124 | 3.473 | 33.102 | 33.818 | 34.793 | 38.5 |
| | | Our $\text{PSU}_{\text{pub}}$ | 1.614 | 0.967 | 0.716 | 0.616 | 0.565 | 2.825 | 2.625 | 2.525 | 2.525 | 2.825 | 2.526 | 2.526 | 2.525 | 14.132 | 14.432 | 14.185 | 14.538 |
| | | Our $\text{PSU}_{\text{sym}}$ | 2.423 | 0.867 | 0.716 | 0.666 | 0.616 | 3.73 | 3.479 | 3.429 | 3.379 | 3.731 | 3.581 | 3.379 | 3.279 | 21.97 | 22.827 | 24.184 | 26.838 |
| | $2^{13}$ | PSU [ZCL+23] | 0.941 | 3.268 | 1.357 | 1.015 | 0.818 | 3.491 | 1.577 | 1.199 | 0.981 | 3.51 | 1.55 | 1.198 | 0.996 | 5.423 | 1.515 | 1.18 | 1.027 |
| | | PSU [JSZ+22] | 7.904 | 0.103 | 0.081 | 0.071 | 0.08 | 1.857 | 2.299 | 2.381 | 2.055 | 6.599 | 6.615 | 6.823 | 7.252 | 67.678 | 68.388 | 70.609 | 75.37 |
| | | Our $\text{PSU}_{\text{pub}}$ | 1.614 | 0.968 | 0.717 | 0.617 | 0.567 | 2.825 | 2.625 | 2.525 | 2.525 | 2.825 | 2.625 | 2.625 | 2.476 | 14.538 | 14.137 | 14.124 | 14.525 |
| | | Our $\text{PSU}_{\text{sym}}$ | 2.423 | 0.817 | 0.717 | 0.667 | 0.617 | 3.731 | 3.53 | 3.38 | 3.279 | 3.93 | 3.528 | 3.38 | 3.279 | 13.731 | 22.83 | 24.538 | 26.653 |
| | $2^{14}$ | PSU [ZCL+23] | 1.755 | 6.299 | 2.729 | 2.008 | 1.617 | 6.53 | 2.987 | 2.379 | 1.948 | 6.744 | 3.126 | 2.424 | 1.978 | 10.41 | 7.147 | 7.209 | 7.194 |
| | | PSU [JSZ+22] | 16.638 | 0.177 | 0.132 | 0.114 | 0.124 | 2.149 | 2.634 | 2.475 | 2.398 | 14.104 | 14.15 | 14.19 | 14.798 | 141.584 | 142.459 | 144.862 | 146.415 |
| | | Our $\text{PSU}_{\text{pub}}$ | 2.681 | 1.22 | 0.818 | 0.718 | 0.768 | 4 | 3.548 | 3.44 | 3.39 | 3.58 | 3.329 | 3.179 | 3.179 | 11.136 | 19.072 | 19.679 | 19.629 |
| | | Our $\text{PSU}_{\text{sym}}$ | 3 | 1.169 | 0.818 | 0.718 | 0.818 | 4.43 | 4.03 | 3.93 | 3.83 | 4.58 | 4.18 | 3.98 | 3.93 | 27.21 | 27.75 | 29.41 | 31.21 |
| | $2^{15}$ | PSU [ZCL+23] | 3.382 | 12.229 | 4.963 | 3.63 | 2.898 | 12.665 | 5.262 | 3.895 | 3.087 | 12.862 | 5.284 | 3.934 | 3.277 | 21.578 | 26.489 | 26.572 | 26.519 |
| | | PSU [JSZ+22] | 35.07 | 0.331 | 0.25 | 0.208 | 0.197 | 3.318 | 3.333 | 3.188 | 3.057 | 29.574 | 29.664 | 29.894 | 30.802 | 296.704 | 296.927 | 299.866 | 304.638 |
| | | Our $\text{PSU}_{\text{pub}}$ | 2.681 | 1.272 | 0.971 | 0.87 | 0.77 | 3.48 | 3.23 | 3.034 | 3.035 | 3.632 | 3.331 | 3.181 | 3.132 | 18.807 | 19.113 | 19.367 | 19.364 |
| | | Our $\text{PSU}_{\text{sym}}$ | 3 | 1.117 | 0.92 | 0.87 | 0.86 | 4.38 | 4.03 | 3.93 | 3.88 | 4.53 | 4.23 | 4.09 | 4.03 | 27.1 | 28.01 | 29.7 | 31.27 |
| | $2^{16}$ | PSU [ZCL+23] | 6.253 | 24.093 | 9.675 | 7.075 | 5.623 | 25.299 | 10.253 | 7.64 | 6.155 | 24.813 | 10.248 | 7.464 | 6.098 | 44.336 | 53.332 | 53.333 | 53.319 |
| | | PSU [JSZ+22] | 73.859 | 0.665 | 0.486 | 0.405 | 0.38 | 6.41 | 6.406 | 6.256 | 6.173 | 62.288 | 62.35 | 62.511 | 63.384 | 622.804 | 654.807 | 626.968 | 629.705 |
| | | Our $\text{PSU}_{\text{pub}}$ | 2.681 | 1.475 | 1.176 | 1.076 | 1.026 | 3.481 | 3.181 | 3.031 | 2.981 | 3.634 | 3.383 | 3.235 | 3.184 | 18.957 | 19.661 | 19.407 | 19.412 |
| | | Our $\text{PSU}_{\text{sym}}$ | 3 | 1.374 | 1.126 | 1.076 | 1.126 | 4.39 | 4.088 | 3.888 | 3.888 | 4.54 | 4.33 | 4.09 | 3.98 | 26.7 | 27.45 | 29.71 | 31.52 |
| | $2^{17}$ | PSU [ZCL+23] | 12.388 | 48.778 | 19.997 | 16.774 | 12.152 | 50.007 | 20.123 | 14.865 | 12.437 | 55.58 | 21.656 | 16.09 | 13.954 | 115.579 | 108.053 | 107.998 | 108.044 |
| | | PSU [JSZ+22] | 155.589 | 1.401 | 1.043 | 0.874 | 0.807 | 13.383 | 13.296 | 13.26 | 13.192 | 131.162 | 131.819 | 131.609 | 131.947 | 1311.533 | 1311.735 | 1315.518 | 1319.699 |
| | | Our $\text{PSU}_{\text{pub}}$ | 2.195 | 1.99 | 1.489 | 1.337 | 1.237 | 3.429 | 3.179 | 3.029 | 3.029 | 4.149 | 3.749 | 3.547 | 3.549 | 19.302 | 19.354 | 19.411 | 19.16 |
| | | Our $\text{PSU}_{\text{sym}}$ | 3 | 1.889 | 1.439 | 1.386 | 1.337 | 4.905 | 4.404 | 4.3 | 4.25 | 5.15 | 4.604 | 4.45 | 4.3 | 18.92 | 27.77 | 29.43 | 31.54 |
| | $2^{18}$ | PSU [ZCL+23] | 26.167 | 96.827 | 39.012 | 28.268 | 22.787 | 98.22 | 39.853 | 29.768 | 23.785 | 105.787 | 42.572 | 31.508 | 30.798 | 218.057 | 213.494 | 213.427 | 213.465 |
| | | PSU [JSZ+22] | 326.313 | 3.227 | 2.486 | 2.186 | 2.221 | 28.446 | 28.304 | 28.321 | 28.262 | 276.011 | 275.668 | 276.009 | 276.486 | 2746.869 | 2769.363 | 2752.628 | 2763.214 |
| | | Our $\text{PSU}_{\text{pub}}$ | 2.367 | 3.723 | 2.584 | 2.172 | 1.972 | 5.833 | 4.833 | 4.179 | 3.931 | 6.137 | 5.134 | 4.534 | 4.232 | 19.384 | 20.095 | 20.851 | 20.661 |
| | | Our $\text{PSU}_{\text{sym}}$ | 3.188 | 3.625 | 2.72 | 2.17 | 1.87 | 6.738 | 5.789 | 5.035 | 4.684 | 7.14 | 6.886 | 5.837 | 5.185 | 26.787 | 29.261 | 31.569 | 32.987 |
| | $2^{19}$ | PSU [ZCL+23] | 52.21 | 195.069 | 78.656 | 61.553 | 50.208 | 194.284 | 80.609 | 62.76 | 50.097 | 213.3 | 87.526 | 68.734 | 64.247 | 436.767 | 436.923 | 437.291 | 436.934 |
| | | PSU [JSZ+22] | 683.002 | 9.416 | 8.126 | 7.184 | 6.804 | 62 | 62 | 61.71 | 61.76 | 579.853 | 580.128 | 580.262 | 580.492 | 5840.408 | 5840.151 | 5754.497 | 5790.476 |
| | | Our $\text{PSU}_{\text{pub}}$ | 2.367 | 9.043 | 5.833 | 4.384 | 3.534 | 11.056 | 7.895 | 6.344 | 5.645 | 11.353 | 8.197 | 6.645 | 5.995 | 19.538 | 20.655 | 20.646 | 20.658 |
| | | Our $\text{PSU}_{\text{sym}}$ | 3.188 | 8.939 | 5.835 | 4.232 | 3.632 | 11.956 | 8.749 | 7.146 | 6.496 | 12.255 | 9.153 | 7.452 | 6.649 | 27.134 | 29.006 | 30.368 | 32.929 |
| | $2^{20}$ | PSU [ZCL+23] | 104.28 | 394.92 | 160.811 | 125.446 | 98.564 | 387.994 | 160.841 | 126.949 | 100.213 | 427.788 | 224.759 | 157.808 | 129.907 | 874.951 | 875.095 | 875.139 | 874.854 |
| | | PSU [JSZ+22] | 1426.855 | 28.137 | 25.432 | 23.716 | 23.039 | 139.148 | 138.854 | 138.346 | 138.324 | 1222.204 | 1221.153 | 1221.029 | 1220.607 | 12450.719 | 12210.308 | 12025.49 | 12115.414 |
| | | Our $\text{PSU}_{\text{pub}}$ | 2.767 | 18.465 | 11.682 | 8.321 | 6.71 | 20.995 | 14.19 | 10.879 | 9.234 | 21.297 | 14.488 | 10.98 | 9.577 | 32.253 | 25.792 | 24.746 | 24.751 |
| | | Our $\text{PSU}_{\text{sym}}$ | 4.341 | 18.428 | 11.513 | 8.263 | 6.756 | 21.696 | 14.884 | 11.375 | 9.924 | 22.146 | 15.39 | 11.783 | 10.429 | 46.729 | 40.956 | 40.681 | 44.508 |

Table 3: Comparisons of communication (in MB) and runtime (in seconds) between PSU [JSZ+22], PSU [ZCL+23], $\text{PSU}_{\text{sym}}$ and $\text{PSU}_{\text{pub}}$ for sets size ($|X| = 2^{10}$, $|Y| \in \{2^{10}, \cdots, 2^{20}\}$) with threads $T \in \{1, 2, 4, 8\}$, and 10Gbps bandwidth, 0.2ms RTT; 100Mbps, 10Mbps and 1Mbps bandwidth, 80ms RTT. The best results are marked in cyan.

**Communication comparison.** Our PSU protocol can achieve the lowest communication among all protocols [JSZ$^+$22, ZCL$^+$23] in the unbalanced case. Our analysis of the unbalanced border is as below: the communication complexity of our PSU is $O(m \log n)$, while the communication complexity of the state-of-the-art PSU protocol due to Zhang et al. [ZCL$^+$23] is $O(n)$. Therefore, in the asymptotic sense the performance border between our protocol and Zhang et al.'s protocol can be deduced from the inequality $m < \frac{n}{\log n}$. Our experimental results shown in Table 3 is consistent with our asymptotic analysis: when ($n \geq 2^{15}$, $m = 2^{10}$), our PSU protocols is superior than [ZCL$^+$23]. Especially, as shown in Figure 13, the larger difference between two set sizes, the better our protocols perform. For set sizes ($|X| = 2^{10}, |Y| = 2^{20}$), the communication of our $\mathrm{PSU_{pub}}$ requires 2.767 MB, which is about 37× lower than PSU [ZCL$^+$23] requiring 104.28 MB, about 515× lower than PSU [JSZ$^+$22] requiring 1426.855MB.

**Runtime comparison.** Our $\mathrm{PSU_{sym}}$ and $\mathrm{PSU_{pub}}$ are faster than PSU [JSZ$^+$22, ZCL$^+$23] in the unbalanced case depending on network environments. As shown in Figure 13, the larger difference between two set sizes, the better our protocols perform. For set sizes ($|X| = 2^{10}$, $|Y| = 2^{20}$) with $T = 1$ thread in LAN setting, the runtime of our $\mathrm{PSU_{pub}}$ requires 18.465 seconds, while PSU [ZCL$^+$23] requires 394.92 seconds, about 21× improvement, PSU [JSZ$^+$22] requires 28.137 seconds, about 1.5× improvement. The performance of our protocols improves significantly in the case of low bandwidth. For set sizes ($|X| = 2^{10}$, $|Y| = 2^{20}$) with $T = 8$ thread in 1 Mbps bandwidth, our $\mathrm{PSU_{pub}}$ requires 24.751 seconds, while PSU [ZCL$^+$23] requires 874.854 seconds, about 35× improvement, PSU [JSZ$^+$22] requires 12115.414 seconds, about 489× improvement.

## References

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In *ASIACCS*, 2012.

[BGH13] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *Public-Key Cryptography - PKC 2013*, pages 1–13, 2013.

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Innovations in Theoretical Computer Science 2012*, pages 309–325, 2012.

[Bon98] Dan Boneh. The decision diffie-hellman problem. In *Algorithmic Number Theory*, pages 48–63, 1998.

[BPMW16] Florian Bourse, Rafaël Del Pino, Michele Minelli, and Hoeteck Wee. FHE circuit privacy almost for free. In *CRYPTO*, 2016.

[BSMD10] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: privacy-preserving aggregation of multi-domain network events and statistics. In *19th USENIX Security Symposium*, pages 223–240, 2010.

[CCS18] Andrea Cerulli, Emiliano De Cristofaro, and Claudio Soriente. Nothing refreshes like a repsi: Reactive private set intersection. In *Applied Cryptography and Network Security ACNS 2018*, pages 280–300, 2018.

[CGP20] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *Advances in Cryptology - ASIACRYPT 2020*, pages 342–372, 2020.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1223–1237, 2018.

[CLP17] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - SEAL v2.1. In *Financial Cryptography and Data Security - FC 2017*, pages 3–18, 2017.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1255, 2017.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In *Advances in Cryptology - CRYPTO 2020*, pages 34–63, 2020.

[CMdG$^+$21] Kelong Cong, Radames Cruz Moreno, Mariana Botelho da Gama, Wei Dai, Ilia Iliashenko, Kim Laine, and Michael Rosenberg. Labeled PSI from homomorphic encryption with reduced computation and communication. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1135–1150, 2021.

[CO18] Michele Ciampi and Claudio Orlandi. Combining private set-intersection with secure two-party computation. In *Security and Cryptography for Networks - 11th International Conference, SCN 2018*, pages 464–482, 2018.

[CZZD22] Yu Chen, Min Zhang, Cong Zhang, and Minglang Dong. Private set operations from multi-query reverse private membership test. Cryptology ePrint Archive, Paper 2022/652, 2022.

[DC17] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In *Information Security and Privacy - 22nd Australasian Conference, ACISP 2017*, pages 261–278, 2017.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13*, pages 789–800, 2013.

[DM03] Luc Devroye and Pat Morin. Cuckoo hashing: Further analysis. *Inf. Process. Lett.*, 86(4):215–219, 2003.

[FNO19] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private set intersection with linear communication from general assumptions. In *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, pages 14–25, 2019.

[FPSS03] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul G. Spirakis. Space efficient hash tables with worst case constant access time. In *STACS 2003*, pages 271–282, 2003.

[Fri07] Keith B. Frikken. Privacy-preserving set union. In *Applied Cryptography and Network Security, ACNS 2007*, pages 237–252, 2007.

[FV12] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, page 144, 2012.

[GDL$^+$16] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *Proceedings of the 33nd International Conference on Machine Learning, ICML*, pages 201–210, 2016.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *Advances in Cryptology - CRYPTO 2012*, pages 850–867, 2012.

[GMR$^+$21] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In *Public-Key Cryptography - PKC 2021*, pages 591–617, 2021.

[GPR$^+$21] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In *Advances in Cryptology - CRYPTO 2021*, pages 395–425, 2021.

[HEK12] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *19th Annual Network and Distributed System Security Symposium, 2012*, 2012.

[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 *IEEE Cybersecurity Development, SecDev 2016*, pages 75–76, 2016.

[HV17] Carmit Hazay and Muthuramakrishnan Venkitasubramaniam. Scalable multi-party private set-intersection. In *Public-Key Cryptography - PKC 2017*, pages 175–203, 2017.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *Advances in Cryptology - CRYPTO 2003*, Lecture Notes in Computer Science, 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. *IACR Cryptol. ePrint Arch.*, page 157, 2022.

[KLS$^+$17] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *Proc. Priv. Enhancing Technol.*, pages 177–197, 2017.

[KMP$^+$17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *ACM SIGSAC Conference on Computer and Communications Security*, pages 1257–1272, 2017.

[KRTW19] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In *Advances in Cryptology - ASIACRYPT 2019*, pages 636–666, 2019.

[KS05] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *Advances in Cryptology - CRYPTO 2005*, pages 241–257, 2005.

[LV04] Arjen K. Lenstra and Tim Voss. Information security risk assessment, aggregation, and mitigation. In *Information Security and Privacy: 9th Australasian Conference, ACISP*, pages 391–401, 2004.

[MR95] Rajeev Motwani and Prabhakar Raghavan. Randomized algorithms. Cambridge university press, 1995.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *Advances in Cryptology - EUROCRYPT 2013*, pages 557–574, 2013.

[PR01] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms - ESA 2001*, pages 121–133, 2001.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *Advances in Cryptology - CRYPTO 2019*, pages 401–431, 2019.

[PSTY19] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In *Advances in Cryptology - EUROCRYPT 2019*, pages 122–153, 2019.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *Proceedings of the 23rd USENIX Security Symposium*, pages 797–812, 2014.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur.*, pages 7:1–7:35, 2018.

[RA18] Amanda Cristina Davi Resende and Diego F. Aranha. Faster unbalanced private set intersection. pages 203–221, 2018.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptol. ePrint Arch.*, page 187, 2005.

[RMY20] Sivaramakrishnan Ramanathan, Jelena Mirkovic, and Minlan Yu. BLAG: improving the accuracy of blacklists. In *NDSS*. The Internet Society, 2020.

[RR17] Peter Rindal and Mike Rosulek. Improved private set intersection against malicious adversaries. In *Advances in Cryptology - EUROCRYPT 2017*, pages 235–259, 2017.

[SV14] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, pages 57–81, 2014.

[ZCL$^+$23] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. Optimal private set union from multi-query reverse private membership test. In *USENIX 2023*, 2023. `https://eprint.iacr.org/2022/358`.

## A Optimization techniques

Our PSU use the dual structure of unbalanced PSI [CLR17, CHLR18, CMdG$^+$21]. Thus, we can take advantage of the same optimization techniques of FHE following [CLR17, CHLR18, CMdG$^+$21], such as batching, windowing, partitioning, and modulus switching, to significantly reduce the depth of the homomorphic circuit. We review the optimization techniques as follows.

**Batching.** Batching is a well-known and powerful technique in fully homomorphic encryption to enable Single Instruction, Multiple Data (SIMD) operations on ciphertexts [GHS12, BGH13, SV14, CLP17, GDL$^+$16]. The batching technique allows the receiver to operate on $\gamma$ items from the sender simultaneously, resulting in $\gamma$-fold improvement in both the computation and communication. As an example, The sender groups its items into vectors of length $\gamma$, encrypts them, and sends $m/\gamma$ ciphertexts to the receiver. Upon receiving each ciphertext $c_i$, the receiver samples a vector $\mathbf{r}_i = (r_{i1}, \cdots, r_{i\gamma}) \in (\mathbb{Z}_t \backslash \{0\})^n$ at random, homomorphically computes $r_i + \mathit{\Pi}_{y \in Y}(c_i - y)$, and sends it back to the sender. Note that these modifications do not affect correctness or security, since the exact same proof can be applied to per each vector coefficient.

**Windowing.** We use a standard windowing technique [CLR17], to lower the depth of the circuit. In our PSU, the receiver needs to evaluate the sender's encrypted data. If the receiver only has an encryption $c \leftarrow \text{FHE}.\mathsf{Enc}(x)$, it samples a random $r$ in $\mathbb{Z}_t \backslash \{0\}$ and homomorphically computes $r + \mathit{\Pi}_{y_i \in Y}(c - y_i)$. The receiver computes at worst the product $x^n$, which requires a circuit of depth $\lceil \log_2 n \rceil$. To see this, we write $r + \mathit{\Pi}_{y_i \in Y}(x - y_i) = r + a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + x^n$. If the sender sends encryptions of extra powers of $x$, the receiver could use these powers to evaluate the same computation with a much lower-depth circuit. More precisely, for a window size of $l$ bits, the sender computes and sends $c_{ij} = \text{FHE}.\mathsf{Enc}(x^{i \cdot 2^{lj}})$ to the receiver for all $1 \le i \le 2^l - 1$, $0 \le j \le \lfloor \log_2(n)/l \rfloor$. For example, when $l = 1$, the receiver sends encryptions of $x, x^2, \cdots, x^{2^{\lfloor \log_2 n \rfloor}}$. This technique results in a significant reduction in the circuit depth.

**Partitioning.** Another way to reduce circuit depth is to let the receiver partition its set into $\alpha$ subsets [CLR17, CHLR18, CMdG$^+$21]. In our PSU, the receiver needs to compute encryptions of all powers $x, \cdots, x^n$ for each of the sender's items $x$. With partitioning, the receiver only needs to compute encryptions of $x, \cdots, x^{n/\alpha}$, which can be reused for each of the $\alpha$ partitions.

**Modulus switching.** We employ modulus switching to effectively reduce the size of the response ciphertexts as [CLR17, CHLR18, CMdG$^+$21, BGV12]. Modulus switching is a well-known operation in lattice-based fully homomorphic encryption schemes. It is a public operation, which transforms a ciphertext with encryption parameter $q$ into a ciphertext encrypting the same plaintext, but with a smaller parameter $q' < q$. Note that the security of the protocol is trivially preserved as long as the smaller modulus $q'$ is determined at setup.