Fast Unbalanced Private Set Union from Fully Homomorphic Encryption

Binbin Tu\textsuperscript{1}, Yu Chen\textsuperscript{1}, Qi Liu\textsuperscript{1}, and Cong Zhang\textsuperscript{2,3}

\textsuperscript{1} School of Cyber Science and Technology, Shandong University
\textsuperscript{2} State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
\textsuperscript{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)-linearly with 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 (FHE) and a newly introduced protocol called permuted matrix Private EQuality Test (pm-PEQT). 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. 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 $2^{19}$ with 128-bit length items, our PSU takes 2.242 MB of communication to compute the union. Compared with the state-of-the-art PSU proposed by Jia et al. (Usenix Security 2022), there are $300\times$ reduction in communication and roughly $30 - 120\times$ reduction in computational overhead in WAN/LAN settings.

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 \cite{34,27,33}, privacy-preserving data aggregation \cite{6}, and private ID \cite{23} etc. However, most PSU \cite{31,22,1,16,33,23} 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 linearly with 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. Most existing PSU protocols are not very efficient in dealing with the above unbalanced case.

Over the last decade, there has been a significant amount of work on private set intersection (PSI) including both balanced \cite{37,39,14,19,7,26,46,42,41,18,28,32} and unbalanced case \cite{40,10,15,45}, but little attention has been paid on PSU, especially in unbalanced case. Recently, Jia et al. \cite{30} propose an unbalanced PSU with shuffling technique, but their PSU\textsuperscript{*} suffers the following drawbacks. First, their PSU\textsuperscript{*} dose 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, the sender can get the intersection item if it inputs a one-element set. Second, the communication complexity of their PSU\textsuperscript{*} is linearly with the size of the larger set. Another closely related work is that of Chen et al. \cite{13}, which shows how to tweak FHE-based

\textsuperscript{4} In this paper, we use PSU\textsuperscript{*} to indicate a PSU protocol with information leakage.
PSI [12] 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 applying the optimization tricks due to [12] will compromise the semi-honest security of the receiver. They left the standardly secure 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. We summarize our contributions as follows:

1. We first propose a basic unbalanced PSU protocol based on leveled FHE. Then, we use an array of optimization techniques following [12,10,15] 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 Permute + Share 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 which have 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. 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^{19}) with 128-bit length items, our PSU protocol takes 2.242 MB of communication and 12 seconds of computation to compute the union with a single thread in LAN settings. Compared with the state-of-the-art PSU [30], there are roughly 300× reduction in communication and 30× reduction in computational overhead. In particular, the performance of our PSU protocols improve significantly in the case of low bandwidth. Our PSU requires 7.79 seconds which is about 120× faster than PSU [30] in 10Mbps bandwidth.

### 1.2 Related Works

We revisit recent PSU protocols [33,23,30,48] with good efficiency. Table 1 provides a brief comparison of our protocols to the prior highest-performing PSU protocols. We report in detail the performance results and comparisons in Section 7.

<table>
<thead>
<tr>
<th>Protocols</th>
<th>Communication</th>
<th>Computation</th>
<th>Security</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSU* [33]</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>Leaky</td>
</tr>
<tr>
<td>PSU [23]</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>Standard</td>
</tr>
<tr>
<td>PSU [48]</td>
<td>O(n)</td>
<td>O(n)</td>
<td>Standard</td>
</tr>
<tr>
<td>PSU* [30]</td>
<td>O(n log n)</td>
<td>O(n log n)</td>
<td>Standard</td>
</tr>
<tr>
<td>Our PSU</td>
<td>O(m log n)</td>
<td>O(n)</td>
<td>Standard</td>
</tr>
</tbody>
</table>

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. PSU* [33] leaks the information to the receiver (the receiver learns some subsets have the intersection items). PSU* [30] leaks the information to the sender (the sender learns the intersection size).
We provide the high-level intuition for our unbalanced PSU protocol. First, we propose a basic PSU protocol and two parties run OT protocol to let the information of the intersection. To remedy the leakage, we introduce a new cryptographic protocol called following \cite{12,10,15} to reduce the depth of homomorphic circuits. However, straightforward application leaks in homomorphic circuits is deep. Then, we try to improve the basic PSU by applying optimization techniques based on leveled FHE. Our basic protocol is easy to understand, but it is not efficient due to the depth of \(O(n^2 \log^2 n)\) computation. However, \cite{30} points out that the bucketing technique leaks the information to \(R\). More precisely, \(R\) learns that some subsets (size \(\rho\)) hold the intersection items with high probability.

Garimella et al. \cite{23} give a PSU protocol based on permuted characteristic functionality which in turn can be built from oblivious switching. Simply speaking, the sender \(S\) holding a set \(X\) interacts with the receiver \(R\) holding a set \(Y\). As a result, \(S\) gets a random permutation \(\pi\) and \(R\) gets a vector \(e \in \{0, 1\}^n\), where if \(e_i = 1, x_{\pi(i)} \in Y\), else \(x_{\pi(i)} \notin Y\). Then, \(R\) runs OT protocol with \(S\) to obtain the set union. Their protocol requires \(O(n \log n)\) communication and \(O(n \log n \log \log n)\) computation. However, \cite{30} gives a generic framework of PSU based on the multi-query reverse private membership test (RPMT). In this paper, we use \(O(n \log n)\) communication and \(O(n \log n)\) computation.

Zhang et al. \cite{48} recently give a generic framework of PSU based on the reverse private membership test (RPMT) and vector decryption-then-matching (VODM). By instantiating OKVS and VODM, they obtain two concrete mq-RPMT. The first is based on symmetric-key encryption and general 2PC. The second is based on re-randomizable public-key encryption. Both constructions achieve linear computation \(O(n)\) and communication \(O(n)\).

Jia et al. \cite{30} propose a PSU with the shuffling technique. Simply speaking, the receiver \(R\) hashes a set \(Y\) into \(Y_c\) by Cuckoo hash and the sender \(S\) hashes a set \(X\) by simple hash. \(R\) shuffles \(Y_c\) by a permutation \(\pi\) chosen by \(S\). \(S\) and \(R\) get shuffled shares \(\{s_{\pi(i)}\}\) and \(\{s'_{\pi(i)}\}\), where \(Y_c[\pi(i)] = s_{\pi(i)} \oplus s'_{\pi(i)}\), respectively. Two parties run mp-OPRF to compute all PRF values and \(S\) sends its PRF values to \(R\). \(R\) tests which items belong to the union and runs OT with \(S\) 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.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Set sizes (m) and (n) are public. Two parties: sender (S) and receiver (R).</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Functionality:</strong></td>
<td>1. Wait for an input (X = {x_1, \cdots, x_m} \subseteq {0, 1}^<em>) from (S), and an input (Y = {y_1, \cdots, y_n} \subseteq {0, 1}^</em>) from (R). 2. Give output (X \cup Y) to (R).</td>
</tr>
</tbody>
</table>

Fig. 1: Ideal functionality \(F_{PSU}^{m,n}\) for private set union

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 depth of homomorphic circuits is deep. Then, we try to improve the basic PSU by applying optimization techniques following \cite{12,10,15} to reduce the depth of homomorphic circuits. However, straightforward application leaks the information of the intersection. To remedy the leakage, we introduce a new cryptographic protocol called

\footnote{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 \(n\).}
permuted matrix private equality test (pm-PEQT). Finally, 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 a secure and fast unbalanced PSU protocol. We describe the ideal functionality of PSU in Figure 1.

2.1 Our Basic PSU Protocol

Our starting point is the FHE-based basic PSI protocol \cite{12} and we review the protocol as follows.

**Basic PSI protocol of Chen et al revisit.** Chen et al. \cite{12} give a basic unbalanced PSI protocol, in which the receiver (\(R\)) with input an item \(y\) interacts with the sender (\(S\)) holding a large set \(X\), and \(R\) can get the intersection \(\{y\} \cap X\). Informally, \(R\) encrypts its item \(y\), and sends the ciphertext \(c ← \text{FHE.Enc}(y)\) to \(S\); \(S\) chooses random non-zero plaintexts \(r\) and homomorphically computes \(c' ← \text{FHE.Enc}(r \cdot f(y))\), where the polynomial \(f(x) = \prod_{i=1}^{n} (x - x_i)\), and then returns \(c'\) to \(R\); \(R\) 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, because 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 \(R\). Meanwhile, \(R\) sends the result \(r + f(y)\) to \(S\) and \(S\) checks whether the item \(y\) belongs to the union by verifying \(r'S = r + f(y)\) and gets the union by OT protocol. In order to let the receiver output the results (requirements of the ideal
We first review optimized unbalanced PSI as follows. Chen et al. [12] 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, the receiver $R$ inserts the small set $Y$ into Cuckoo hash table $Y_c$ by Cuckoo hash and each bin $Y_c[i]$ consists of one item. The sender $S$ inserts the large set $X$ into hash table $\mathcal{H}$.

### 2.2 Optimized PSU with Leakage

We first review optimized unbalanced PSI as follows. Chen et al. [12] 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, the receiver $R$ inserts the small set $Y$ into Cuckoo hash table $Y_c$ by Cuckoo hash and each bin $Y_c[i]$ consists of one item. The sender $S$ inserts the large set $X$ into hash table $\mathcal{H}$.
Compute \( r'_{ij} = r_{ij} \cdot f_{ij}(y_j) \) from \( m\) into the output securely. However, if we divide a large \((n,m)\)-PSI into many small \((1,B')\)-PSI, each small PSI has a low homomorphic circuit. We review the basic PSI protocol and its optimizations in Figure 3.

According to the requirements of the ideal functionality of PSU, we consider the dual structure of \([12]\). 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 \([12]\) to improve our basic PSU. Roughly, the sender \( S \) hashes the small set \( X \) into \( X_c \) by Cuckoo hash and each bin \( X_c[i] \) consists of one item. The receiver \( R \) hashes the large set \( Y \) into hash table \( Y_b \) and each bin \( Y_b[i] \) consists of \( B \) items. Then \( R \) 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, \( R \) encodes each subset \( (B') \) into a polynomial and randomizes it by adding a random value, then it homomorphically computes and sends new ciphertexts to \( S \). \( S \) 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 show the comparison of PSI \([12]\) and our optimized PSU (omit OT) with leakage (right).

We emphasize that, unlike PSI \([12]\), 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 \([12]\) and our optimized PSU (omit OT) with leakage in Figure 4. Note that in the standardly secure \((m,n)\)-PSU, from the view of \( R \), any item in the set \( Y \) could be an item in \( X \cap Y \). However, in above optimized PSU*, \( R \) learns some subsets with size \( B' \) have the item in \( X \cap Y \). Moreover, if \( S \) returns its decrypted results \( r' \) to \( R \) directly, \( R \) 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, \( R \) gets \( f'(x) \neq 0 \) in other subsets, which causes \( R \) could compute the intersection items with sufficient polynomial values. For example, in Figure 4 (right), in the first column, if \( r_{11} = r'_{11} \), this means \( x_1 \in Y \) and \( x_1 \in \{y_{11}, \ldots, y_{B'+1}\} \), but \( x_1 \notin \{y_{B'+1}, \ldots, y_{B+1}\} \). \( R \) gets the rest nonzero polynomial values \( f_{21}(x_1), \ldots, f_{n1}(x_1) \) and it could compute \( x_1 \) from them.

---

\( X_b \) by simple hash, where the \( i \)-th bin indicates as \( X_b[i] \) and each bin consists of \( B \) items. \( S \) 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, \( S \) encodes each subset \((B') \) into a polynomial and randomizes it by multiplying a random value, then it homomorphically computes and sends new ciphertexts to \( R \). \( R \) decrypts the ciphertexts and gets the set intersection. We show the comparison of PSI \([12]\) (left) and optimized PSU (omit OT) with leakage (right).

---

\* In the PSI \([12]\), they use cuckoo hashing with no stash and three simple hash functions. For each same bins in hash tables \( Y \) and \( X_b \), if the item \( Y_b[i] \in X_b[i] \), that is, the item belongs to the intersection.

\& 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.
Based on above analysis, the main challenge is how to optimize our basic PSU without causing information leakage. More precisely, we need to overcome the following two difficulties:

- The receiver is able to check \( r_{ij} = f_{ij}(x) \) at all positions without knowing \( r_{ij} + f_{ij}(x) \).
- The receiver is able to check \( r_{ij} = f_{ij}(x) \) in same positions without knowing the positions \( i, j \).

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

### 2.3 Permuted Matrix Private Equality Test

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 learns a bit indicating whether \( x = y \) and nothing else, whereas the sender learns nothing. In our pm-PEQT, the sender holding a matrix \( R_{n \times m} \) and a matrix permutation \( \pi = (\pi_e, \pi_r) \) interacts with a receiver holding a matrix \( R_{n \times m} \). As a result, the receiver learns (only) the bit matrix \( B_{n \times m} \) indicating if \( b_{ij} = 1 \), \( r_{\pi(ij)} = r'_{\pi(ij)} \), else, \( r_{\pi(ij)} \neq r'_{\pi(ij)} \), \( i \in [n], j \in [m] \), while the sender learns nothing about \( R \). 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 \( R_{n \times m} \). The sender with a matrix \( R'_{n \times m} \) and a matrix permutation \( \pi = (\pi_e, \pi_r) \), where \( \pi_e \) (over \([m]\)) is a column permutation and \( \pi_r \) (over \([n]\)) is a row permutation. \( \alpha \) and \( m \) are public.

**Functionality:**

1. Wait for an input \( R' = [r'_{ij}], i \in [\alpha], j \in [m] \) and a permutation \( \pi = (\pi_e, \pi_r) \) from the sender, and an input \( R = [r_{ij}], i \in [\alpha], j \in [m] \) from the receiver.
2. Give the bit matrix \( B_{n \times m} \) to the receiver, where \( r_{\pi(ij)} = r'_{\pi(ij)} \); \( b_{ij} = 1 \), else, \( b_{ij} = 0 \), for all \( i \in [\alpha], j \in [m] \).

**Constructions of pm-PEQT.** pm-PEQT can not be easily built from PEQT by running many PEQT instances in parallel. This is because it is difficult 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 [23,30] and mp-OPRF [38]. Informally, \( S \) and \( R \) invoke the ideal Permute + Share functionality \( F_{ps} \) twice: First, both parties permute and share the columns of \( R \). \( R \) inputs each column of \( R \) and \( S \) inputs a permutation \( \pi_e \) over \([m]\). As a result, \( R \) gets shuffled share \( S_{\pi_e} = [s_{\pi_e(ij)}] \) and \( S \) gets \( S'_{\pi_e} = [s'_{\pi_e(ij)}] \), where \( s_{\pi_e(ij)} \oplus s'_{\pi_e(ij)} = r_{\pi_e(ij)} \). Then both parties permute and share the rows of \( S_{\pi_e} \). \( R \) inputs each row of \( S_{\pi_e} \) and \( S \) inputs a permutation \( \pi_r \) over \([\alpha]\). As a result, \( R \) gets \( S_{\pi_r} = [s_{\pi_r(ij)}] \) and \( S \) gets \( S'_{\pi_r} = [s'_{\pi_r(ij)}] \), where \( s_{\pi_r(ij)} \oplus s'_{\pi_r(ij)} = r_{\pi_r(ij)} \). \( R \) defines the shuffled matrix shares \( S_x = S_{\pi_r} \) and \( S \) defines the shuffled matrix shares \( S'_x = \pi_r(S_{\pi_e}) \oplus S'_{\pi_e} \), where \( s_{\pi(ij)} \oplus s'_{\pi_e(ij)} = r_{\pi_r(ij)} \), \( i \in [\alpha], j \in [m] \). Then, both parties invoke mp-OPRF functionality \( F_{mp-oprf} \). \( R \) inputs shuffled shares \( S_x \) and obtains \( F_k(s_{\pi(ij)}), i \in [\alpha], j \in [m] \), and \( S \) gets the key \( k \) of a PRF. Furthermore, \( S \) permutes the matrix \( R' \) by \( \pi = (\pi_e, \pi_r) \) and gets \( R'_\pi = [r'_{\pi(ij)}] \), then \( S \) computes all PRF values \( F_k(r'_{\pi(ij)} \oplus s'_{\pi(ij)}) \), \( i \in [\alpha], j \in [m] \) and sends them to \( R \). Finally, \( R \) sets \( b_{ij} = 1 \), if \( F_k(s_{\pi(ij)}) = F_k(r'_{\pi(ij)} \oplus s'_{\pi(ij)}) \), else, sets \( b_{ij} = 0 \). \( R \) lets
the bit matrix $B = [b_{ij}], i \in [\alpha], j \in [m]$. Note that the Permute + Share \cite{5,8} and mp-OPRF \cite{38,9} 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 \cite{2}. Let $G$ be a cyclic group with order $q$, where the DDH problem is hard. Informally, $\mathcal{R}$ and $\mathcal{S}$ choose random value $a, b \leftarrow \mathbb{Z}_q$ 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 $H()$ is a group element in $G$. Let $V = [v_{ij}]$ and $V' = [v_{ij}', i \in [\alpha], j \in [m]]. \mathcal{R}$ sends $V$ to $\mathcal{S}$. Then $\mathcal{S}$ computes $v_{ij}'' = (v_{ij})^a$ and lets $V'' = [v_{ij}''], i \in [\alpha], j \in [m]$. $\mathcal{S}$ shuffles $V''$ and $V'$ by same permutation $\pi = (\pi_c, \pi_r)$ and gets $V''_\pi = \pi(V'')$. $V'_\pi = \pi(V')$, where $v_{\pi(i)}'' = \pi(\pi(v_{ij}))''$ and $v_{\pi(r(i))}' = \pi(v_{ij})'$. $\mathcal{S}$ sends permuted matrices $V''_\pi$ and $V'_\pi$ to $\mathcal{R}$. Finally, for $i$-th row and $j$-column in $V''_\pi$ and $V'_\pi$, if $v_{\pi(i)}''(j) = v_{\pi(i)}'(j)$, $\mathcal{R}$ lets $b_{ij} = 1$, else, $b_{ij} = 0$. $\mathcal{R}$ gets a bit matrix $B = [b_{ij}], i \in [\alpha], j \in [m]$. The communication complexity of our DDH-based pm-PEQT is $O(\alpha m)$.

2.4 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 in Figure 6 and the details are as follows.

First, we construct a semi-finished FHE-based optimized PSU\footnote{Each column of the matrix $R_{\alpha \times m}$ corresponds to the same item, so the permutation for the matrix requires that the columns are consistent.} in which the sender $\mathcal{S}$ does not send its decrypted results $R'_{\alpha \times m}$ to the receiver $\mathcal{R}$. $\mathcal{S}$ holding a small set $X$ interacts with $\mathcal{R}$ holding a large set $Y$. The result is $\mathcal{R}$ outputs a matrix $R_{\alpha \times m} = [r_{ij}]$, and $\mathcal{S}$ outputs a matrix $R'_{\alpha \times m} = [r'_{ij}]$, where $\alpha$ denotes the number of partitions, $m$ denotes the number of bins, $r_{ij}$ denotes the random value 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, by using pm-PEQT, $\mathcal{S}$ inputs $R'$ and a permutation $\pi = (\pi_c, \pi_r)$ and $\mathcal{R}$ inputs $R$. As a result, $\mathcal{R}$ gets a bit vector $b = [b_{ij}], j \in [m], \forall i \in [\alpha], b_{ij} = 0, \forall i \in [\alpha], j \in [m], r_{\pi(i)} = r'_{\pi(i)}; \mathcal{R}$ computes a bit vector $b_{ij} = 1, \forall i \in [\alpha], j \in [m], b_{\pi(i)} = 1, \forall i \in [\alpha], b_{\pi(i)} = 0, \mathcal{R}$ permutes the cuckoo hash table $X_c$ by $\pi_c$ and gets $\pi_c(X_c) = [x_{\pi(1)}, \cdots, x_{\pi(m)}]$. We note that if $b_{ij} = 1, x_{\pi(i)} \notin Y$, else, $x_{\pi(i)} \in Y, j \in [m]$.

Finally, by using OT protocol, $\mathcal{S}$ inputs $(\bot, x_{\pi(i)}), j \in [m], \mathcal{R}$ inputs $b_{ij}, j \in [m], \mathcal{R}$ gets $x_{\pi(i)},$ if $b_{ij} = 1$, else, gets $\bot$. After that, $\mathcal{R}$ outputs the union $Y \cup \{x_{\pi(i)}\}$.

In this way, we complete our construction of secure and fast unbalanced PSU protocol.

Fig. 6: Core design idea of our full PSU protocol
3 Preliminaries

3.1 Notation

We denote the parties in our PSU as sender (S) and receiver (R), and their respective input sets as X and Y with \( m = |X| \ll n = |Y| \). For \( n \in \mathbb{N} \), \([n]\) denotes the set \( \{1, 2, \ldots, n\} \). \( 1^\lambda \) 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., \( s \). We denote matrices by upper-case bold letters, e.g., \( S \). We write \( S = [s_{ij}] \) to denote each element of \( S \), where \( s_{ij} \) denotes the element in \( i \)-th row and \( j \)-th column. For a permutation \( \pi \) over \( n \) items, we write \( \{s_{\pi(1)}, \ldots, s_{\pi(n)}\} \) to denote \( \pi(\{s_1, \ldots, s_n\}) \), 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 \( S = [s_{ij}] \), we write \( S_{\pi_c} \) (or, \( S_{\pi_r} \)) to denote \( \pi_c(S) = [s_{\pi_c(ij)}] \) (or, \( \pi_r(S) = [s_{\pi_r(ij)}] \)) be 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.

3.2 Building Blocks

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

Cuckoo hashing. Cuckoo hashing [36,17,21,43] can be used to build dense hash tables by many hash functions. Following [12], 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 [12], our protocols require the leveled FHE satisfies IND-CPA secure with circuit privacy [3], but we can use oblivious pseudorandom function (OPRF) to avoid the requirement of circuit privacy as [10]. We refer the reader to [12,10,15,3] for more details. We use an array of optimization techniques of FHE as [12,10,15], such as batching, windowing, partitioning and modulus switching to significantly reduce the depth of the homomorphic circuit. We review the optimizations in appendix A. For the implementation, we use the homomorphic encryption library SEAL which implements the BFV scheme [20] following [12,10,15].

Oblivious transfer. Oblivious transfer [44] 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. [29] 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 \( F_{\text{OT}} \) in Figure 7.

Parameters: Two parties: sender \( S \) and receiver \( R \).
Functionality:
1. Wait for input \( \{x_0, x_1\} \) from \( S \). Wait for input \( b \in \{0, 1\} \) from \( R \).
2. Give \( x_b \) to \( R \).

Fig. 7: 1-out-of-2 oblivious transfer functionality \( F_{\text{OT}} \)
Multi-point oblivious pseudorandom function. An oblivious pseudorandom function (OPRF) allows the receiver to input \(x\) and learns the PRF value \(F_k(x)\), where \(F\) is a PRF, and \(k\) is known to the sender. Pinkas et al. [38] propose multi-point OPRF (mp-OPRF) and realize efficient PSI protocols. Recently, Chase and Miao [9] 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 \(F_{\text{mp-OPRF}}\) in Figure 8.

### Parameters: A PRF \(F\). Two parties: sender \(S\) and receiver \(R\).

### Functionality:

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

Fig. 8: \(F_{\text{mp-OPRF}}\) functionality

### Permute + Share. We recall the Permute + Share (PS) functionality \(F_{\text{PS}}\) defined by Chase et al. [8] in Figure 9. Roughly speaking, in the Permute + 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 \(x_\pi(i) = s_\pi(i) \oplus s'_\pi(i), i \in [n]\).

### 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. 9: \(F_{\text{PS}}\) functionality

### 4 The Basic PSU Protocol

We describe our basic PSU protocol in Figure 10 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.

**Theorem 1.** The PSU protocol described in Figure 10 is secure in the \(F_{\text{OT}}\)-hybrid model, in the presence of semi-honest security adversaries, provided that the fully homomorphic encryption scheme is IND-CPA secure with circuit privacy.

**Proof.** We construct \(\text{Sim}_S\) and \(\text{Sim}_R\) to simulate the views of corrupted sender \(S\) and corrupted receiver \(R\) respectively, and argue the indistinguishability of the produced transcript from the real execution.
We give two efficient constructions of pm-PEQT in the semi-honest model. The functionality is specified in Figure 11.

Corrupt Sender. Sim$_S(X)$ simulates the view of corrupt $S$ as follows: It encrypts $m$ random values. Then, it invokes Sim$_{OT}^S(\perp,x_i), i \in [m]$ and appends the output to the view. Now we argue that the view output by Sim$_S$ 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. Sim$_R(Y,X\cup \perp)$ simulates the view of corrupt $R$ as follows: It simulates the ciphertexts by encrypting $m$ random value. Sim$_R$ sets $X = (X \cup \perp)$ and pads $X$ with $\perp$ into $m$ items and permutes all items randomly. It computes the polynomial $f(y) = \Pi_{y_i \in Y}(y-y_i)$ and the random values $r = [r_i], i \in [m]$ used to randomize the polynomial. Then, for $x_i \neq \perp$, Sim$_R$ defines $r'_i := f(x_i) + r_i$, else, it defines $r'_i := r_i$, and appends $r'_i$ to the view. If $x_i = \perp$, it sets $b_i = 0$, else, $b_i = 1$. Then Sim$_R$ invokes Sim$_{OT}^R(b_i,x_i)$ for $i \in [m]$ and appends the output to the view.

We argue that the outputs of Sim$_R$ are indistinguishable from the real view of $R$ by the following hybrids: Hyb$_0$: $R$’s view in the real protocol.

Hyb$_1$: Same as Hyb$_0$ except that the ciphertexts in the step 2 are replaced by encrypting $m$ random values generated by Sim$_R$. Since the fully homomorphic encryption scheme is IND-CPA secure, the above simulation is indistinguishable from the real view.

Hyb$_2$: Same as Hyb$_1$ except that Sim$_R$ runs the $F_{OT}$ simulator to produce the simulated view for $R$. 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 Sim$_R$.

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 [7]

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

The first construction of pm-PEQT is based on the Permute + Share [23,30] and mp-OPRF [9] as described in Figure [11].

Theorem 2. The construction of Figure [11] securely implements functionality $F_{pm-PEQT}$ in the $(F_{PS}, F_{mp-OPRF})$-hybrid model, in the presence of semi-honest adversaries.
The second construction of pm-PEQT is based on DDH as described in Figure 12.

Output: The receiver outputs a bit matrix B. The sender outputs ⊥.

1. S and R invoke the ideal Permute + Share functionality F_{PS} twice. First, both parties permute and share the columns of R, where each column of R can be seen as an item. R inputs each column of R and S inputs the permutation π_r. As a result, R gets S_{π_r} = [s_{π_r(i,j)}] and S gets S’_{π_r} = [s’_{π_r(i,j)}], where s_{π_r(i,j)} ⊕ s’_{π_r(i,j)} = r_{π_r(i,j)}.

Then, both parties permute and share the rows of S_{π_r}, where each row of S_{π_r} can be seen as an item. R inputs each row of S_{π_r} and S inputs the permutation π_. As a result, R gets S_π = [s_{π(i,j)}] and S gets S’_π = [s’_{π(i,j)}], where s_{π(i,j)} ⊕ s’_{π(i,j)} = r_{π(i,j)}. Finally, R defines the shuffled matrix shares S_π = S_{π_r} and S defines the shuffled matrix shares S’_π = π_r(S_{π_r}) ⊕ S_{π_r}, where s_{π(i,j)} ⊕ s’_{π(i,j)} = r_{π(i,j)}, i ∈ [α], j ∈ [m].

2. Both parties invoke mp-OPRF functionality F_{mp-OPRF}. R inputs shuffled shares S_π and obtains the outputs F_k(s_{π(i,j)}), i ∈ [α], j ∈ [m]. S obtains the key k.

3. S computes F_k(r’_{π(i,j)} ⊕ s’_{π(i,j)}), i ∈ [α], j ∈ [m] and sends them to R.

4. R sets b_{ij} = 1, if F_k(s_{π(i,j)}) = F_k(r’_{π(i,j)} ⊕ s’_{π(i,j)}), else, b_{ij} = 0, and gets a bit matrix B = [b_{ij}], i ∈ [α], j ∈ [m].

Fig. 11: pm-PEQT from Permute + Share and mp-OPRF

Proof. We exhibit simulators Sim_R and Sim_S for simulating corrupt R and S respectively, and argue the indistinguishability of the produced transcript from the real execution.

Corrupt Sender. Sim_S(R’, π = (π_c, π_r)) simulates the view of corrupt S as follows: Sim_S randomly chooses S’_{π_r} and invokes Sim_{PS}^S(π_c, S’_{π_r}) and appends the output to the view. Sim_S randomly chooses S’_{π_r} and invokes Sim_{PS}^S(π_r, S’_{π_r}) and appends the output to the view. Then, Sim_S randomly selects a key k of PRF and invokes Sim_{mp-OPRF}^S(k) and appends the output to the view.

We argue that the outputs of Sim_S are indistinguishable from the real view of S by the following hybrids:

Hyb_0: S’s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of F_{PS} is replaced by S’_{π_r}, S’_{π_r} chosen by Sim_S, and Sim_S runs the F_{PS} simulator to produce the simulated view for S. The security of Permute + Share guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_2: Same as Hyb_1 except that the output key of F_{mp-PEQT} is replaced by the k chosen by Sim_S, and Sim_S runs the F_{mp-PEQT} simulator to produce the simulated view for 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 Sim_S.

Corrupt Receiver. Sim_R(R, B = [b_{ij}]) simulates the view of corrupt R as follows: Sim_R chooses S_{π_r} and invokes Sim_{PS}^R(R, S_{π_r}) and appends the output to the view. Sim_R chooses S_{π_r} and invokes Sim_{PS}^R(S_{π_r}, S_{π_r}) and appends the output to the view. Sim_R randomly selects u_{ij}, i ∈ [α], j ∈ [m] and invokes Sim_{mp-OPRF}^R(u_{ij}) and appends the output to the view. Finally, for all i ∈ [α], j ∈ [m], Sim_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 Sim_R is indistinguishable from a real view of R by the following hybrids:

Hyb_0: R’s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of F_{PS} is replaced by S_{π_r}, S_{π_r} chosen by Sim_R, and Sim_R runs the F_{PS} simulator to produce the simulated view for R. The security of Permute + Share guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_2: Same as Hyb_1 except that the output PRF values of F_{mp-OPRF} is replaced by u_{ij}, i ∈ [α], j ∈ [m], and all PRF values in the last step is replaced by the u_{ij}, chosen by Sim_R randomly, and Sim_R runs the F_{mp-OPRF} simulator to produce the simulated view for 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 12.
Input: The receiver $\mathcal{R}$ inputs a matrix $R_{m \times n}$. The sender $S$ inputs a matrix $R_{m \times n}$, and a permutation $\pi = (\pi_c, \pi_r)$ where $\pi_c$ (over $[m]$) and $\pi_r$ (over $[n]$) are two sub-permutation for columns and rows. $\mathbb{G}$ is a cyclic group with order $q$.

Output: The receiver outputs a bit matrix $B$. The sender outputs $\bot$.

1. $\mathcal{R}$ choose random value $a \leftarrow \mathbb{Z}_q$ and compute $v_{ij} = H(r_{ij})^a$ for all $i \in [\alpha]$, $j \in [m]$, where $H(\cdot)$ denotes hash functions which output the elements of group $\mathbb{G}$. Let $V = [v_{ij}]$, $i \in [\alpha]$, $j \in [m]$.

2. $\mathcal{S}$ choose random value $b \leftarrow \mathbb{Z}_q$ and compute $v'_{ij} = H(r'_{ij})^b$ for all $i \in [\alpha]$, $j \in [m]$, where $H(\cdot)$ denotes hash functions which output the elements of group $\mathbb{G}$. Let $V' = [v'_{ij}]$, $i \in [\alpha]$, $j \in [m]$. Then $\mathcal{S}$ computes $v''_{ij} = (v''_{ij})^b$ and lets $V'' = [v''_{ij}]$. $\mathcal{S}$ shuffles $V''$ and $V'$ by same permutation $\pi = (\pi_c, \pi_r)$ and gets $V'' = \pi(V'')$, $V' = \pi(V')$, where $v''_{ij} = \pi(v_{ij})$, $v'_{ij} = \pi(v''_{ij})$. $\mathcal{S}$ sends $V''$, $V'$ to $\mathcal{R}$.

3. For $i$-th row and $j$-column in $V''$ and $V'$, if $v''_{ij} = v''_{ij}$, $\mathcal{R}$ lets $b_{ij} = 1$, else, $b_{ij} = 0$. $\mathcal{R}$ sets a bit matrix $B = [b_{ij}]$, $i \in [\alpha]$, $j \in [m]$.

Fig. 12: Instantiation of pm-PEQT based on DDH

**Theorem 3.** The construction of Figure 12 securely implements functionally $F_{\text{pm-PEQT}}$ based on DDH in the random oracle model, in the presence of semi-honest security 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}($$\mathcal{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:

- **Hyb$_0$:** $\mathcal{S}$’s view in the real protocol consists of $H(r_{ij})^a$, $i \in [\alpha]$, $j \in [m]$, where $a \leftarrow \mathbb{Z}_q$.
- **Hyb$_1$:** Same as $\text{Hyb}_0$ except that $\text{Sim}_\mathcal{S}$ 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 $a \leftarrow \mathbb{Z}_q$.

We argue that the views in $\text{Hyb}_0$ and $\text{Hyb}_1$ are computationally indistinguishable. Let $A$ be a probabilistic polynomial-time (PPT) adversary against the DDH assumption. Given the DDH challenge $g^x$, $g^{y_1}$, $g^{z_1}$, where $x, y_1, z_1 \leftarrow \mathbb{Z}_q$, $A$ is asked to distinguish if $z_1 = x \cdot y_1$ or random values. $A$ implicitly sets $a = x$, and simulates (with the knowledge of $\mathcal{R}$) the view as below:

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

**Corrupt Receiver.** $\text{Sim}_\mathcal{R}($$\mathcal{B} = [b_{ij}]$) simulates the view of corrupt $\mathcal{R}$ as follows: $\text{Sim}_\mathcal{R}$ 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 $\text{Sim}_\mathcal{R}$ are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:

- **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 $a, b \leftarrow \mathbb{Z}_q$.
- **Hyb$_1$:** Same as $\text{Hyb}_0$ except that for $b_{ij} = 0$, that is $r_{\pi_{ij}} = r'_{\pi_{ij}}$, $\text{Sim}_\mathcal{R}$ chooses random group elements $v_{ij}$ and $u_{ij}$ instead of $H(r'_{\pi_{ij}})^b$ and $H(r'_{\pi_{ij}})^{ab}$.
- **Hyb$_2$:** Same as $\text{Hyb}_1$ except that for $b_{ij} = 1$, that is $r_{\pi_{ij}} = r'_{\pi_{ij}}$, $\text{Sim}_\mathcal{R}$ 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 $\text{Sim}_\mathcal{R}$.
We argue that the view in \( \text{Hyb}_1 \) and \( \text{Hyb}_2 \) are computationally indistinguishable based on the DDH assumption. Given the DDH challenge \( g^x, g^{y_{ij}}, g^{z_{ij}}, g^{z_{ij}^2} \), where \( x, y_{ij}, z_{ij} \leftarrow \mathbb{Z}_q \). \( A \) is asked to distinguish if \( z_{ij} = x \cdot y_{ij}, z_{ij}' = x \cdot y_{ij}' \) or random values. \( A \) implicitly sets \( b = x \), and simulates (with the knowledge of \( R' \) and \( \pi \)) the view as below:

- RO queries: \( \text{Sim}_R \) honestly emulates random oracle \( H \). For queries \( r_{ij} \) and \( r_{ij}' \), if \( r_{ij} \notin R, r_{ij}' \notin R' \), it assigns \( H(r_{\pi(ij)}), H(r_{\pi(ij)}') \) with random group elements. If \( r_{ij} \in R, r_{ij}' \in R' \), it assigns \( H(r_{ij}) = g^{a^{-1}y_{ij}}, H(r_{ij}') = g^{y_{ij}} \).
- Outputs \( g^{z_{ij}}, g^{z_{ij}^2} \).

Clearly, if \( z_{ij} = x \cdot y_{ij}, z_{ij}' = x \cdot y_{ij}' \), \( A \) simulates \( \text{Hyb}_2 \). Else, it simulates \( \text{Hyb}_1 \). In the \( \text{Hyb}_1 \), \( \text{Sim}_R \) needs not to know the \( R' \) 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 \( \text{Hyb}_1 \) and \( \text{Hyb}_2 \) are computationally indistinguishable based on the DDH assumption. Given the DDH challenge \( g^x, g^{y_{ij}}, g^{z_{ij}}, g^{z_{ij}^2} \) where \( x, y_{ij} \leftarrow \mathbb{Z}_q \), \( A \) is asked to distinguish if \( z_{ij} = x \cdot y_{ij} \) or random values. \( A \) implicitly sets \( b = x \), and simulates (with the knowledge of \( R' \) and \( \pi \) for all positions with \( b_{ij} = 1 \)) the view as below:

- RO queries: \( \text{Sim}_R \) honestly emulates random oracle \( H \). For queries \( r_{ij} \), \( r_{ij}' \), if \( r_{ij} \notin R, r_{ij}' \notin R' \), it assigns \( H(r_{ij}), H(r_{ij}') \) with random group elements. If \( r_{ij} \in R, r_{ij}' \in R' \), it assigns \( H(r_{ij}) = g^{a^{-1}y_{ij}}, H(r_{ij}') = g^{y_{ij}} \).
- Outputs \( g^{a^{-2}z_{ij}}, g^{z_{ij}^2} \).

Clearly, if \( z_{ij} = x \cdot y_{ij}, z_{ij}' = x \cdot y_{ij}' \), \( A \) simulates \( \text{Hyb}_1 \). Else, it simulates \( \text{Hyb}_2 \). In the \( \text{Hyb}_2 \), \( \text{Sim}_R \) needs not to know the \( R' \) 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 \( \text{Sim}_R \) are computationally indistinguishable from the real view based on the DDH assumption.

**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 [13] in a general manner.

### 6 Full PSU Protocol

In this section, we detail our full PSU protocol in Figure [13]. The main optimization idea of our protocol is as follows.

**Offline/online.** Following [12][10][15], 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 can send the parameters to the sender, and the sender can pad the same dummy items like \( \bot \) which are known two both parties.

**OPRF Pre-processing.** We use dual structure of [12] in our PSU and prove the security based on FHE with circuit privacy following [12]. This leads to perform a noise flooding operation on the result ciphertexts, as was necessary in [12]. Following [10][15], we can use an OPRF to compute the items on both sides before engaging in the PSU. This ensures that the receiver’s items \( Y \setminus X \) are psorandom in the sender’s view, preventing the sender from learning anything about the original items, even if it learns full PRF values. Thus, our PSU can be proven security without circuit privacy and utilize more efficient FHE parameters as [10][15], which improves our performance and adds flexibility to the parametrization.

**Low circuit depth.** The steps 1-6 in our full PSU protocol can be seen as the dual structure of unbalanced PSI [12][10][15]. Therefore, the optimizations used in [12][10][15], 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].
Input: The receiver \( R \) inputs set \( Y \subset \{0,1\}^n \) of size \( n = |Y| \) and the sender \( S \) inputs set \( X \subset \{0,1\}^m \) of size \( m = |X| \), where \( m \ll n \) and \( n \) are public.

Output: The receiver outputs \( X \cup Y \). The sender outputs \( \bot \).

1. **[Setup]** \( R \) and \( S \) agree on the hashing, FHE scheme, mp-PEQT and OT parameters.

2. **[Hashing]** \( S \) hashes the set \( X \) into table \( X_c \) by Cuckoo hash, where \( X_c \) consists of \( m_c \) bins and each bin has only one item. \( R \) hashes the set \( Y \) into table \( Y_{B \times m_c} \) by same simple hash, where \( Y_{B \times m_c} \) consists of \( m_c \) bins and each bin has \( B \) items.

3. **[Pre-process \( Y \)]**
   (a) **[Partitioning]** \( R \) partitions \( Y_{B \times m_c} \) by rows into \( \alpha \) subtables \( Y_1, \cdots, Y_\alpha \). Each subtable has \( B' = B/\alpha \) rows and \( m \) columns. Let \( i \)-th subtable be \( Y_i = [y_{i,1}, \cdots, y_{i,B'}]^T, i \in [\alpha] \), where \( y_{i,k}, k \in [B'] \) indicates \( k \)-th row of \( Y_i \).
   (b) **[Computing coefficients]** For \( j \)-th columns of \( i \)-th subtable \( y'_{i,j} = [y_{i,j,1}, \cdots, y_{i,j,B'}]^T, i \in [\alpha], j \in [m_c] \), where \( y_{i,j,k}, k \in [B'] \) indicates \( k \)-th item of \( y'_{i,j} \), \( R \) computes the coefficients of the polynomial \( f_{i,j}(y) = \Pi_{k=1}^{B'} (y-y_{i,j,k}) = a'_{i,j,0} + a'_{i,j,1}y + \cdots + a'_{i,j,B'}y^{B'} \). \( R \) computes the coefficient matrix \( A \) as follows: \( R \) chooses a random matrix \( A_{m \times m_c} = [r_{i,j}] \) and sets \( j \)-th columns of \( i \)-th subtable \( A_{i,j} = [a_{i,j,0}, a_{i,j,1}, \cdots, a_{i,j,B'}]^T, i \in [\alpha], j \in [m_c] \), where \( a_{i,j,0} = a'_{i,j,0} + r_{i,j} \).
   (c) **[Batching]** For each subtable obtained from the previous step, \( R \) interprets each of its row as a vector of length \( m_c \) with elements in \( \mathbb{Z}_t \). Then \( R \) batches each vector into \( \beta = m_c/\gamma \) plaintext polynomials. As a result, each row of \( i \)-th subtable \( A_i \) is transformed into \( \beta \) polynomials denoted \( A_{i,j}, i \in [\alpha], j \in [\beta] \).

4. **[Encrypt \( X \)]**
   (a) **[Batching]** \( S \) 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 \( \hat{X} \), \( S \) computes the component-wise \( i \cdot 2^j \)-th powers \( \hat{X}^{i \cdot 2^j} \), for \( 1 \leq i \leq 2^j - 1 \) and \( 0 \leq j < \lceil \log_2(B')/\gamma \rceil \).
   (c) **[Encrypt]** \( S \) uses FHE scheme to encrypt each such power, obtaining \( \beta \) collections of ciphertexts \( C_j, j \in [\beta] \), and each collection consists of the ciphertexts \([c_{i,j}], 1 \leq i \leq 2^j - 1 \) and \( 0 \leq j < \lceil \log_2(B')/\gamma \rceil \). \( S \) sends these ciphertexts to \( R \).

5. **[Computation]**
   (a) **[Homomorphically compute encryptions of all powers]** For each collection \( C_j, j \in [\beta] \), \( R \) homomorphically computes encryptions of all powers \( C_j = [c_{j,0}, \cdots, c_{j,B'}] \), which \( c_{j,k}, 0 \leq k \leq B' \) is a homorphic ciphertext encrypting \( \hat{X}_k \).
   (b) **[Homomorphically evaluate the dot product]** \( R \) homomorphically evaluates \( C'_{i,j} = C_j \hat{A}_{i,j}, i \in [\alpha], j \in [\beta] \), performs modulus switching on \( C'_{i,j} \) to reduce sizes, and sends the ciphertexts to \( S \).

6. **[Decrypt]** \( S \) gets and decrypts all ciphertexts and concatenates the results into the matrix \( B_{\alpha \times m_c} \).

7. **[mp-PEQT]** \( R \) inputs the matrix \( R_{\alpha \times m_c} \), and \( S \) inputs the permutation \( \tau = (\pi_c, \pi_r) \) and the matrix \( R_{\alpha \times m_c} \). Both parties invoke the mp-PEQT functionality. As a result, \( R \) gets a bit matrix \( B_{\alpha \times m_c} \), where if \( b_{ij} = 1 \), \( r_{\pi(i,j)} = r'_{\pi(i,j)}, \) else \( r_{\pi(i,j)} \neq r'_{\pi(i,j)}, i \in [\alpha], j \in [m_c] \).

8. **[Output]** \( R \) sets a bit vector \( b = [b_j], j \in [m_c] \), where if for all \( i \in [\alpha], b_{ij} = 0 \), it sets \( b_j = 1 \), else \( b_j = 0 \). Then, \( R \) and \( S \) invoke the OT functionality, in which \( R \) inputs \( b_j, j \in [m_c] \) and \( S \) inputs \( \langle \bot, X_c[\pi_c(j)] \rangle \). If \( b_j = 1 \), \( R \) gets \( X_c[\pi_r(j)] \), else, it gets \( \bot \). Finally, \( R \) outputs the set union \( Y \cup \{ X_c[\pi_c(j)] \}, \) for all \( j \in [m_c] \).

Fig. 13: Full PSU protocol
We give 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 Permute + 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 $O(m \log n)$.

We show the correctness of our PSU protocol and prove its security as follows.

**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 13 is a secure protocol for $F_{PSU}$ in the ($F_{pm-PEQT}$, $F_{OT}$)-hybrid model, in the presence of semi-honest adversaries, provided that the fully homomorphic encryption scheme is IND-CPA secure with circuit privacy.

**Proof.** For ease of exposition, we will assume that all parameters are fixed and public. We exhibit simulators $Sim_S$ and $Sim_R$ for simulating corrupt $S$ and $R$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

**Corrupt Sender.** $Sim_S(X)$ simulates the view of corrupt $S$ as follows. $Sim_S$ 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 $R'$ and chooses randomly permutation $\pi = (\pi_c, \pi_r)$. It invokes $Sim^{S}_{pm-PEQT}(R', \pi)$ and $Sim^{S}_{OT}(\bot, X_c[\pi_c(j)])$, $j \in [m_c]$ appends the output to the view. Now we argue that the view output by $Sim_S$ 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.** $Sim_R(Y, X \cup Y)$ simulates the view of corrupt receiver $R$ as follows: $Sim_R$ encrypts random value in place of the ciphertexts in step 4. It chooses the random matrix $R$ in step 3. $Sim_R$ computes $X = (X \cup Y) \setminus Y$ and pads $X$ with $\bot$ to $m_c$ items and permutes these items randomly. For all items in $X$, if $\hat{x}_i \neq \bot$, it sets $b_i = 1$, else, it sets 0. And then it generates $B_{\alpha \times m_c}$, for all columns $b_i$, if $b_i = 1$, it sets all items in $b_i$ be 0, else, it sets one random position in $b_i$ be 1 and all other positions are 0. $Sim_R$ invokes $Sim^{R}_{pm-PEQT}(R, B)$ appends the output to the view. Then, for all $i \in [m_c]$, it invokes $Sim^{R}_{OT}(b_i, \hat{x}_i)$ and appends the output to the view.

The view generated by $Sim_R$ is indistinguishable from a real view of $R$ by the following hybrids:

$Hyb_0$: $R$’s view in the real protocol.

$Hyb_1$: Same as $Hyb_0$ except that the ciphertexts are replaced by encrypting random values generated by $Sim_R$. Since the fully homomorphic encryption scheme is IND-CPA secure, the simulation is indistinguishable from the real view.

$Hyb_2$: Same as $Hyb_1$ except that the output of $F_{pm-PEQT}$ is replaced by $B$ generated by $Sim_R$, and $Sim_R$ runs the $F_{pm-PEQT}$ simulator to produce the simulated view for $R$. The security of the pm-PEQT protocol guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

$Hyb_3$: Same as $Hyb_2$ except that $Sim_R$ runs the $F_{OT}$ simulator to produce the simulated view for $R$. 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 $Sim_R$.

### 7 Implementation and Performance

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

- **PSU_{PS}:** PSU protocol based on FHE, OT and pm-PEQT, where pm-PEQT is built from Permute + Share and mp-OPRF.
- **PSU_{DDH}:** PSU protocol based on FHE, OT and DDH-based pm-PEQT.
<table>
<thead>
<tr>
<th>Parameters</th>
<th>Protocols</th>
<th>Comm. (MB)</th>
<th>Runtime (s), $T = 1$</th>
<th>Runtime (s), $T = 4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>X</td>
<td></td>
<td>Y</td>
<td>$</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>$2^{18}$</td>
<td>PSUDDH</td>
<td>2.026</td>
<td>0.216</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSUFS</td>
<td>2.14</td>
<td>0.93</td>
</tr>
<tr>
<td></td>
<td>$2^{20}$</td>
<td>PSUDDH</td>
<td>2.223</td>
<td>0.428</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSUFS</td>
<td>2.45</td>
<td>1.848</td>
</tr>
<tr>
<td></td>
<td>$2^{22}$</td>
<td>PSUDDH</td>
<td>3.24</td>
<td>1.33</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSUFS</td>
<td>4.11</td>
<td>6.82</td>
</tr>
<tr>
<td>$2^{11}$</td>
<td>$2^{18}$</td>
<td>PSUDDH</td>
<td>3.063</td>
<td>0.436</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSUFS</td>
<td>3.56</td>
<td>1.85</td>
</tr>
<tr>
<td></td>
<td>$2^{20}$</td>
<td>PSUDDH</td>
<td>3.343</td>
<td>0.436</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSUFS</td>
<td>3.56</td>
<td>1.85</td>
</tr>
<tr>
<td></td>
<td>$2^{22}$</td>
<td>PSUDDH</td>
<td>4.51</td>
<td>1.55</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSUFS</td>
<td>5.69</td>
<td>7.64</td>
</tr>
</tbody>
</table>

Table 2: Communication (in MB) and runtime (in seconds) of PSUFS and PSUDDH for unbalanced sets size $(|X| \in \{2^{10}, 2^{11}\}$ and $|Y| \in \{2^{18}, 2^{20}, 2^{22}\}$) with threads $T \in \{1, 4\}$, and 10Gbps network bandwidth, 0.2ms RTT.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Protocols</th>
<th>Comm. (MB)</th>
<th>Total running time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>X</td>
<td></td>
<td>Y</td>
</tr>
<tr>
<td>$2^{10}$</td>
<td>$2^{18}$</td>
<td>PSU [30]</td>
<td>326.313</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSU* [30]</td>
<td>117.99</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSU+ [33]</td>
<td>600.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUFS</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUDDH</td>
<td>2.212</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>$2^{18}$</td>
<td>PSU [30]</td>
<td>683.001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSU* [30]</td>
<td>117.991</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSU+ [33]</td>
<td>2470.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUFS</td>
<td>3.07</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUDDH</td>
<td>2.212</td>
</tr>
<tr>
<td>$2^{20}$</td>
<td>$2^{18}$</td>
<td>PSU [30]</td>
<td>326.386</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSU+ [33]</td>
<td>600.62</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUFS</td>
<td>5.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUDDH</td>
<td>3.499</td>
</tr>
<tr>
<td>$2^{19}$</td>
<td>$2^{18}$</td>
<td>PSU [30]</td>
<td>683.001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSU* [30]</td>
<td>237.864</td>
</tr>
<tr>
<td></td>
<td></td>
<td>PSU+ [33]</td>
<td>2470.1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUFS</td>
<td>5.41</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Our PSUDDH</td>
<td>3.499</td>
</tr>
</tbody>
</table>

Table 3: Comparisons of communication (in MB) and runtime (in seconds) between PSU [30], PSU* [30], PSU+ [33], PSUFS and PSUDDH for unbalanced sets size $(|X| \in \{2^{10}, 2^{11}\}$, $|Y| \in \{2^{18}, 2^{19}\}$) with threads $T \in \{1, 4\}$, and 10Gbps bandwidth, 0.2ms RTT; 100Mbps and 10Mbps bandwidth, 80ms RTT. The best results are marked in blue.

We first give our experimental environment, then compare our protocols with the state-of-the-art works in terms of communication and runtime on different network environment. Our source code is available upon request.
The size of large set $Y$

Communication (in MB), $|X| = 2^{10}$

The network bandwidth (Mbps), $|X| = 2^{10}$

Runtime (s), $|X| = 2^{10}$

Communication (in MB), $|X| = 2^{10}$

Runtime (s), $|X| = 2^{10}$

The size of large set $Y$

The network bandwidth (Mbps)

Fig. 14: Comparisons of communication (in MB) and runtime (in seconds) between PSU [30], PSU* [30], PSU* [33], PSU_{DDH} and PSU_{PS}

7.1 Experimental Setup

We run our experiments on a single Intel Core i7-9700 CPU @ 3.00GHz with 8 physical cores and 8GB 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 [20], Permute + Share in [35], mp-OPRF in [9] and OT extension in [29]. For concrete analysis, we set the computational security parameter $\kappa = 128$ and the statistical security parameter $\lambda = 40$. Our implementation is written in C++, and we use the following libraries in our implementation.

- FHE: SEAL [https://github.com/microsoft/SEAL] and APSI [https://github.com/microsoft/APSI]
- Permute + 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 list the experimental results of our protocols in Table 2. Then, we compare our PSU protocols with PSU [30], PSU* [30] and PSU* [33] in terms of runtime and communication, and the results are reported in Table 3 and Figure 14.
We stress that all reported costs are computed in the same environment. For comparisons of other works [30,33], we use the parameters recommended in their open-source code and fix the item length to 128-bit.

- PSU and PSU*: [30]: https://github.com/dujiajun/PSU
- PSU*: [33]: https://github.com/osu-crypto/PSU

**Communication comparison.** Our PSU$_{DDH}$ achieves the lowest communication among all protocols [30,33] in unbalanced case. For set sizes ($|X| = 2^{10}$, $|Y| = 2^{19}$), the communication of our PSU$_{DDH}$ requires 2.242 MB, which is about 300× lower than PSU [30] requiring 683 MB, about 50× lower than PSU* [30] requiring 117.9 MB and 1100× lower than PSU* [33] requiring 2470 MB. As shown in Figure [14] the larger difference between two set sizes, the better our protocols perform.

**Runtime comparison.** Our PSU$_{PS}$ and PSU$_{DDH}$ are faster than PSU [30,33] in unbalanced case. As shown in Figure [14] the larger difference between two set sizes, the better our protocols perform. For set sizes ($|X| = 2^{10}$, $|Y| = 2^{19}$) with $T = 1$ thread in LAN setting, the runtime of our PSU$_{DDH}$ requires 12 seconds, while PSU [30] requires 384.55 seconds, about 30× improvement, PSU* [30] requires 34.72 seconds, about 2.8× improvement, PSU* [33] requires 112.41 seconds, about 9× improvement. The performance of our protocols improves significantly in the case of low bandwidth. For set sizes ($|X| = 2^{10}$, $|Y| = 2^{19}$) with $T = 4$ thread in 10Mbps, our PSU$_{DDH}$ requires 8.52 seconds, while PSU [30] requires 959.75 seconds, about 110× improvement, PSU* [30] requires 100.05 seconds, about 11× improvement, PSU* [33] requires 2077.88 seconds, about 240× improvement.

**References**

the protocol is trivially preserved as long as the smaller modulus $q$, a ciphertext encrypting the same plaintext, but with a smaller parameter encryption schemes. It is a public operation, which transforms a ciphertext with encryption parameter to effectively reduce the size of the modulus switching.

We employ modulus switching as $[12,10,15,5]$ to significantly reduce the size 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 $[24,47,11,25]$. 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 seeing each ciphertext $c_i$, the receiver samples a vector $r = (r_{i1}, \cdots, r_{i\gamma}) \in (\mathbb{Z}_q \setminus \{0\})^\gamma$ at random, homomorphically computes $r_i + \Pi_{y \in Y} (c_i - y)$, 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 per each vector coefficient.

If the sender sends encryptions of extra powers of $x$, the receiver uses 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 $c_{ij} = \text{FHE.Enc}(x^{i \cdot 2^j})$ to the receiver for all $1 \leq i \leq 2^l - 1, 0 \leq j \leq \lfloor \log_2 (n) / l \rfloor$. 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 $[12,10,15]$. 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 it can reuse for each of the $\alpha$ partitions.

**Modulus switching.** We employ modulus switching as $[12,10,15,5]$ to effectively reduce the size of the response ciphertexts. 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.

A Optimization techniques

Our PSU use the dual structure of unbalanced PSI $[12,10,15]$. Thus, we can take advantage of same optimization techniques of FHE following $[12,10,15]$, such as batching, windowing and modulus switching, to significantly reduce the depth of the homomorphic circuit. We review the optimization techniques as follows.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Page Numbers</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>43.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>44.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>45.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>46.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>47.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>48.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>49.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>51.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>52.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>53.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>54.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>57.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>63.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>64.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>65.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>66.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>67.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>68.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>69.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>70.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>71.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>72.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

21