ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

Fast Enhanced Private Set Union in the Balanced and Unbalanced Scenarios

Binbin Tu[1,2,3], Yujie Bai[1,2,3], Cong Zhang[4], Yang Cao[1,2,3], and Yu Chen[1,2,3](✉)
[1]*School of Cyber Science and Technology, Shandong University, Qingdao 266237, China*
[2]*Quan Cheng Laboratory, Jinan 250103, China*
[3]*Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao 266237, China*
[4]*Institute for Advanced Study, BNRist, Tsinghua University, Beijing, China*
{tubinbin,yuchen}@sdu.edu.cn, {baiyujie,caoyang24}@mail.sdu.edu.cn, zhangcong@mail.tsinghua.edu.cn

Abstract

Private set union (PSU) allows two parties to compute the union of their sets without revealing anything else. It can be categorized into balanced and unbalanced scenarios depending on the size of the set on both sides. Recently, Jia et al. (USENIX Security 2024) highlight that existing scalable PSU solutions suffer from during-execution leakage and propose a PSU with enhanced security for the balanced setting. However, their protocol's complexity is superlinear with the size of the set. Thus, the problem of constructing a linear enhanced PSU remains open, and no unbalanced enhanced PSU exists. In this work, we address these two open problems:

- **Balanced case:** We propose the first linear enhanced PSU. Compared to the state-of-the-art enhanced PSU (Jia et al., USENIX Security 2024), our protocol achieves a 2.2 - 8.8× reduction in communication cost and a 1.2 - 8.6× speedup in running time, depending on set sizes and network environments.

- **Unbalanced case:** We present the first unbalanced enhanced PSU, which achieves sublinear communication complexity in the size of the large set. Experimental results demonstrate that the larger the difference between the two set sizes, the better our protocol performs. For unbalanced set sizes $(2^{10}, 2^{20})$ with single thread in 1Mbps bandwidth, our protocol requires only 2.322 MB of communication. Compared with the state-of-the-art enhanced PSU, there is 38.1× shrink in communication and roughly 17.6× speedup in the running time.

1 Introduction

Private set union (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 numerous applications, such as cyber risk assessment [28, 32, 33], privacy-preserving data aggregation [11], and private ID [23] etc.

Early PSU constructions are mainly based on additively homomorphic encryption (AHE), resulting in excessive computational overhead, due to the heavy public key operations. Kolesnikov et al. [32] introduce a new cryptographic primitive named the reversed private membership test (RPMT) and provide the first scalable PSU construction based on symmetric-key primitives. After that, a series of works [16, 23, 30, 45, 48] have further explored the efficiency of PSU. Recently, Zhang et al. [48] formalize an ideal functionality named multi-query RPMT (mqRPMT) and construct PSU protocols whose complexity is linear in the size of both sets. Considering the unbalanced case, Tu et al. [45, 46] propose a new functionality called permuted multi-point private equality test (pmpPEQT) and combine it with fully homomorphic encryption (FHE) to construct an unbalanced mqRPMT and obtain an efficient PSU whose communication is sublinear in the size of the large set. Currently, efficient PSU protocols are mainly based on mqRPMT and oblivious transfer (OT).

However, Jia et al. [29] point out that the PSU framework based on mqRPMT and OT suffers from *during-execution leakage*. More specifically, both parties input their sets X and Y and run mqRPMT to let the receiver learn whether each item in X belongs to the set Y. Then, the receiver obtains the items in $X \backslash Y$ by invoking the underlying OT instances with the membership information. Thus, the mqRPMT-based PSU framework above leaks the membership information (mqRPMT outputs the indication vector to indicate the membership of X) before the execution of the protocol is completed. To remedy the leakage issue, they define a new PSU functionality with enhanced security and construct an *enhanced PSU* protocol based on symmetric-key operations. However, the complexity of their protocol scaled superlinearly with the size of the set.

Influence of during-execution leakage. As pointed out by Jia et al. [29]: *Guo et al. [27] launched attacks on protocols that aim to hide intersections, but allow leakage of intersection sizes. They implemented the attack on practical datasets to obtain tokens of COVID-19 patients and the interest of the* *person associated with a specific personal_id.* Due to the leakage of indication bits (i.e., the intersection size) during the execution of the PSU using mqRPMT, an attacker can exploit the attack from [27] on the mqRPMT-based PSU, repeatedly interact with the sender, and interrupt the execution to obtain more intersection sizes and infer the intersection.

Motivation. Based on the discussions so far, existing scalable PSU protocols suffer from the following problems:

- In the balanced setting, the existing PSU protocols either have superlinear complexity [29] or suffer from during-execution leakage [16, 48].
- In the unbalanced setting, PSU [45, 47] suffer from during-execution leakage.

Motivated by the above discussions, we ask the following questions and try to achieve the "best of both worlds".

Is it possible to design an enhanced PSU protocol in the balanced setting, whose complexity is linear in the size of the set? Is it possible to design an enhanced PSU protocol in the unbalanced setting, whose communication complexity is sublinear in the size of the large set?

1.1 Our contribution

In this paper, we give an affirmative answer to the above questions through the following results.

A new framework of enhanced PSU. All mqRPMT-based PSU protocols leak indication bits (membership information of X) to the receiver and then the receiver aggregates the non-intersection items $X \backslash Y$ by OT. In this work, we formalize a new ideal functionality named permuted non-membership conditional randomness generation (pnMCRG). Combined with the standard hash-to-bin technique (cuckoo/simple hashing), we propose a generic construction of enhanced PSU (ePSU) from pnMCRG. Compared with mqRPMT, pnMCRG hides the indication bits (avoiding during-execution leakage) and outputs the corresponding indication values: For membership $x \in Y$, it outputs unequal random values. For non-membership $x \notin Y$, it outputs equal random values. Indication values can be used as one-time pads to encrypt and decrypt non-intersection items $X \backslash Y$ for constructing the PSU.

Two constructions in balanced/unbalanced settings. We first present a generic construction of permuted non-membership conditional randomness generation (pnMCRG) from newly introduced protocols called permuted membership conditional randomness generation (pMCRG) and non-equality conditional randomness generation (nECRG). nECRG could be constructed in turn from the secret-shared private equality test (ssPEQT) [20, 34] and random oblivious transfer (ROT). In the balanced case, permuted membership conditional randomness generation (pMCRG) could be constructed based on batched OPRF (bOPRF) [31], oblivious key-value store (OKVS) [8, 42], and a new protocol called permuted equality conditional randomness generation (pECRG), achieving linear complexity. In the unbalanced case, permuted membership conditional randomness generation (pMCRG) could be constructed based on bOPRF [31], FHE [13, 15, 17], and permuted equality conditional randomness generation (pECRG), achieving sublinear communication in the size of the larger set. Finally, we realize pECRG from the DDH assumption and achieve linear complexity. Combined with linear non-equality conditional randomness generation (nECRG), pnMCRG-based PSU inherits the complexity of permuted membership conditional randomness generation (pMCRG). Therefore, we obtain two constructions of enhanced PSU:

- In the balanced setting, our enhanced PSU achieves linear complexity in the size of both sets.
- In the unbalanced setting, our enhanced PSU achieves sublinear communication in the size of the larger set.

Table 1 provides a comparison of our protocols with existing scalable PSU protocols. In the balanced setting, n represents the size of both sets. In the unbalanced setting, m and n denote the sizes of the small set and large set, respectively, where $m \ll n$. We ignore the pub-key cost of κ base OTs.

Protocols	Communication	Computation	Enhanced Security
PSU [32]	$O(n \log n)$	$O(n \log n \log\log n)$	×
PSU [23]	$O(n \log n)$	$O(n \log n)$	×
PSU [30]	$O(n \log n)$	$O(n \log n)$	×
PSU [48]	$O(n)$	$O(n)$	×
PSU [16]	$O(n)$	$O(n)$	×
PSU [29]	$O(n \log n)$	$O(n \log n)$	✓
Our PSU	$O(n)$	$O(n)$	✓
PSU [45]	$O(m \log n)$	$O(n + m \log n)$	×
Our PSU	$O(m \log n)$	$O(n + m \log n)$	✓

Table 1: Comparisons of PSU in the semi-honest model

Evaluations. We implement and compare our balanced enhanced PSU with the state-of-the-art enhanced PSU [29]. Experiments show that our protocol achieves a 2.2 - 8.8× reduction in communication cost and a 1.2 - 8.6× speedup in running time, depending on set sizes and network environments. Since the complexity of our PSU is linear, the larger the set size, the more significant our advantage becomes.

Since the implementation of PSU [29] does not support running in unbalanced settings, we compare our unbalanced ePSU (eUPSU) with our balanced ePSU in the unbalanced setting. Experimental results demonstrate that our unbalanced ePSU reduces communication costs by a factor of 1.2 to 38.1, with the small set size fixed at 2^{10} and the

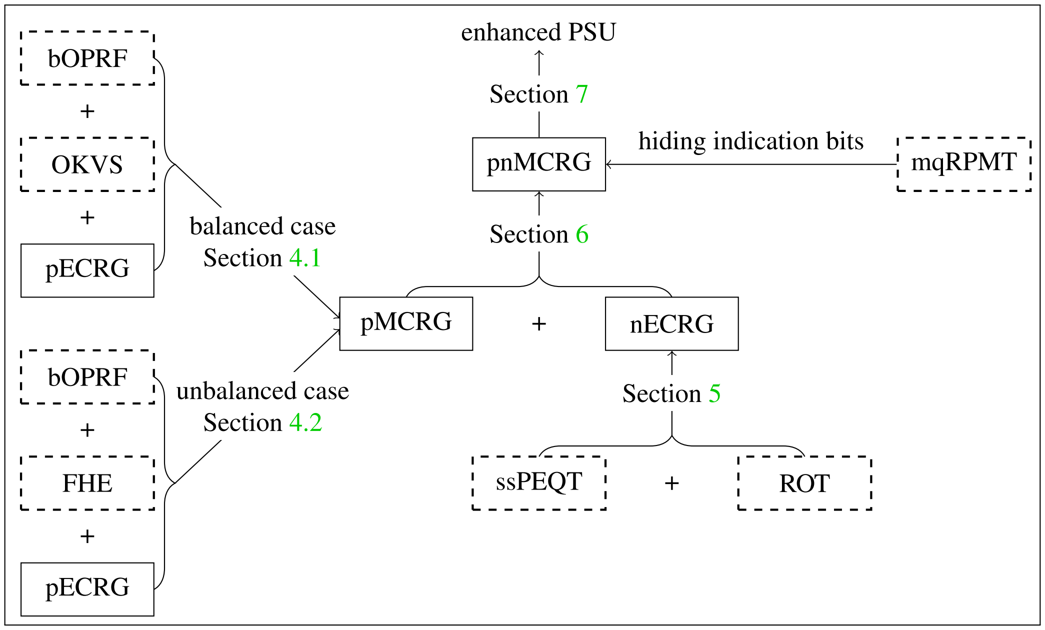

Figure 1: Technical overview of our frameworks. The rectangle with dotted lines denotes the previous notions. The rectangles with solid lines denote new notions, including permuted non-membership conditional randomness generation (pnMCRG), permuted membership conditional randomness generation (pMCRG), permuted equality conditional randomness generation (pECRG), and non-equality conditional randomness generation (nECRG).

large set size ranging from 2^{14} to 2^{20}. Especially, for set sizes $(|X| = 2^{10}, |Y| = 2^{20})$ with $T = 1$ thread in 1Mbps bandwidth, our protocol requires only 2.322 MB and 42.734 seconds, there are $38.1\times$ shrink in communication and roughly $17.6\times$ speedup in the running time.

We compare our balanced/unbalanced ePSU with efficient PSU [45,47,48], which suffers from during-execution leakage. Our protocols resolve their issue of during-execution leakage with only a minor efficiency trade-off. Specifically, in the unbalanced setting, our ePSU also demonstrates an advantage in terms of communication overhead.

1.2 Technical Overview

We provide a high-level technical overview of our enhanced PSU protocol depicted in Figure 1. First, we formalize an ideal functionality named permuted non-membership conditional randomness generation (pnMCRG) and present a framework of enhanced PSU (ePSU). Then, we propose a generic construction of pnMCRG from permuted membership conditional randomness generation (pMCRG) and non-equality conditional randomness generation (nECRG), where nECRG could be constructed from secret-shared private equality test (ssPEQT) and random oblivious transfer (ROT). In the balanced case, pMCRG could be constructed from bOPRF, OKVS, and permuted equality conditional randomness generation (pECRG). In the unbalanced case, pMCRG could be constructed from bOPRF, FHE, and pECRG. For convenience, 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 $|X| = m$, $|Y| = n$, where $m = n$ in the balanced setting, $m \ll n$ in the unbalanced setting.

1.2.1 Enhanced PSU from pnMCRG

We start with a special case that the sender $\mathcal{S}$ has only one item x, and the receiver $\mathcal{R}$ has a set $Y = \{y_1, \cdots, y_n\}$. First, we formalize a new functionality named single-point nMCRG: $\mathcal{S}$ inputs an item x and $\mathcal{R}$ inputs a set Y, the result is that $\mathcal{S}$ and $\mathcal{R}$ obtain their indication values u and v, respectively, such that if $x \notin Y$, $u = v$, otherwise $u \neq v$. $(1,n)$-PSU[1] can be constructed from single-point nMCRG as follows: The sender uses u as one-time pads to encrypt the item $c = u \oplus x$[2], and the receiver can decrypt c by computing $c \oplus v$. For non-membership $x \notin Y$, we have $u = v$. Thus, the receiver obtains a non-intersection item $c \oplus v = u \oplus x \oplus v = x$. For membership $x \in Y$, we have $u \neq v$, and the receiver can only learn a random value.

[1] $(1,n)$-PSU denotes the sender holds one item and the receiver holds n items.

[2] To constructing PSU, the sender could encode its item x concatenating a hash value $x||h(x)$ by a pre-agreed hash function h, or padding x with 64-bit 0 strings so that the receiver could check whether the item belongs to the union or is a random value [20,34]. For convenience, we omit the encoded operations.

Now, we show how to extend the special case to a general case of $|X| = m > 1$. Intuitively, one can simply execute the above process for each $x_i \in X$. However, in this simple method, each item x_i requires to be compared to the entire set Y, causing a significant overhead. We utilize the standard hash-to-bin technique to reduce the costs. Specifically, $\mathcal{S}$ assigns each of its items $x||\gamma$ (concatenated with one hash function index $\gamma \in [3]$) to one of the bins $h_1(x), h_2(x), h_3(x)$ by cuckoo hashing [35], where the cuckoo hash table has m_c bins and each bin contains at most one item. $\mathcal{R}$ assigns each of its items $y||\gamma$ (concatenated with all hash function indices $\gamma \in [3]$ to ensure that there are no identical items in the hash table.) to all of the bins $h_1(y), h_2(y), h_3(y)$. Let Y_i, $i \in [m_c]$ denote the i-th bin. Then, the parties perform the above single-point nMCRG on each bin. Although this method greatly reduces the input size of $\mathcal{R}$ from the entire set $|Y|$ to a small hash bin, it cannot be used directly to construct the PSU, because it leaks additional information on the sender's items to $\mathcal{R}$. That is, whether the sender's i-th item belongs to the i-th bin (subset of Y). Note that PSU can only allow $\mathcal{R}$ to know if x_i belongs to the entire set Y, rather than some small subsets (i-th bin), which narrows the range of x_i.

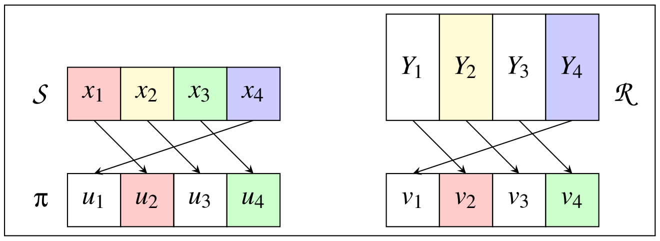

Figure 2: Illustration of pnMCRG. $\pi(1) = 2, \pi(2) = 3, \pi(3) = 4, \pi(4) = 1$. $x_{\pi^{-1}(1)} \in Y_{\pi^{-1}(1)} \Rightarrow u_1 \neq v_1$; $x_{\pi^{-1}(2)} \notin Y_{\pi^{-1}(2)} \Rightarrow u_2 = v_2$; $x_{\pi^{-1}(3)} \in Y_{\pi^{-1}(3)} \Rightarrow u_3 \neq v_3$; $x_{\pi^{-1}(4)} \notin Y_{\pi^{-1}(4)} \Rightarrow u_4 = v_4$.

To remedy the above leakage, we employ the shuffling technique and introduce a new cryptographic protocol named permuted non-membership conditional randomness generation (pnMCRG), depicted in Figure 2[3]. Roughly speaking, pnMCRG is a two-party protocol between a sender holding a set $X = \{x_1, \cdots, x_m\}$ and a permutation π over $[m]$, and a receiver holding m sets $\{Y_1, \cdots, Y_m\}$. After execution, the sender and the receiver obtain random vectors $\mathbf{u} = (u_1, \cdots, u_m)$ and $\mathbf{v} = (v_1, \cdots, v_m)$, respectively, such that if $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, $u_i = v_i$, otherwise $u_i \neq v_i$. To construct the PSU, for all $i \in [m]$, the sender uses each u_i as one-time pads to encrypt the corresponding items $c_i = u_i \oplus x_{\pi^{-1}(i)}$[4], and the receiver can decrypt c_i by computing $c_i \oplus v_i$. Therefore, for non-membership $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, we have $u_i = v_i$, and the receiver obtains non-intersection items $c_i \oplus v_i = u_i \oplus x_{\pi^{-1}(i)} \oplus v_i = x_{\pi^{-1}(i)}$. For membership $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, we have $u_i \neq v_i$, and it obtains random values.

[3]The white color represents non-membership and non-equality, and the same other color represents membership and equality.

[4]For convenience, we omit the concatenation of hash value.

pnMCRG vs. mqRPMT. Multi-query reversed private membership test (mqRPMT) [48] is a two-party protocol between a sender holding a vector $X = (x_1, \cdots, x_m)$ and a receiver holding a set Y. After execution, the receiver obtains an indication bit vector $(e_1, \cdots, e_m)$ such that $e_i = 1$ if and only if $x_i \in Y$ but without knowing x_i, while the sender obtains nothing. In the construction of PSU, the receiver can retrieve the non-intersection items $X \backslash Y$ by OT. As discussed above, due to the leakage of indication bits (membership information of X) in mqRPMT, mqRPMT-based PSU protocols suffer from during-execution leakage [29].

Permuted non-membership conditional randomness generation (pnMCRG) can be seen as a weak notion of mqRPMT, where both parties obtain two indication values (hiding membership information of X) instead of indication bits in mqRPMT. Thus, pnMCRG-based PSU avoids the during-execution leakage.

1.2.2 Generic Constructions of pnMCRG

Here, we present a generic construction of permuted non-membership conditional randomness generation (pnMCRG) from pMCRG and nECRG.

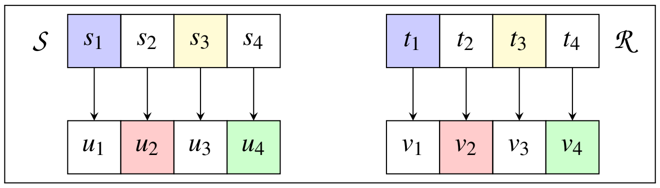

Figure 3: Illustration of nECRG. $s_1 = t_1 \Rightarrow u_1 \neq v_1$; $s_2 \neq t_2 \Rightarrow u_2 = v_2$; $s_3 = t_3 \Rightarrow u_3 \neq v_3$; $s_4 \neq t_4 \Rightarrow u_4 = v_4$.

pnMCRG from pMCRG and nECRG. Starting with the single-point case, where $\mathcal{S}$ has only one item and the receiver $\mathcal{R}$ has a set, we formalize the functionality of single-point membership conditional randomness generation (MCRG): $\mathcal{S}$ inputs an item x and $\mathcal{R}$ inputs a set Y, the result is that $\mathcal{S}$ and $\mathcal{R}$ obtains indication values s and t, respectively. For membership $x \in Y$, we have $s = t$, otherwise, $s \neq t$. Then, following the ideas of [29, 34], we introduce a functionality named non-equality conditional randomness generation (nECRG), depicted in Figure 3, in which both parties input s and t, the result is outputting new random values u and v, such that if $s = t$, $u \neq v$, otherwise $u = v$. That is, nECRG could exchange the equality and non-equality conditions. So far, we can obtain a single-point non-membership conditional randomness generation (nMCRG), from single-point MCRG and nECRG.

Now, we formalize a new functionality of permuted membership conditional randomness generation (pMCRG), depicted in Figure 4, to extend the special case to a general case of $|X| = m > 1$. In the pMCRG, the sender inputs a set $X = \{x_1, x_2, \cdots, x_m\}$ and a permutation π over $[m]$ and the receiver inputs a series of sets $(Y_1, Y_2, \cdots, Y_m)$. As a result, the sender and the receiver obtain random vectors $\mathbf{s} = (s_1, s_2, \cdots, s_m)$ and $\mathbf{t} = (t_1, t_2, \cdots, t_m)$, respectively. If $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, $s_i = t_i$, otherwise $s_i \neq t_i$.

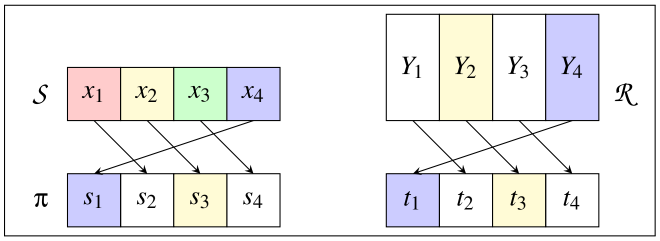

Figure 4: Illustration of pMCRG. $\pi(1) = 2, \pi(2) = 3, \pi(3) = 4, \pi(4) = 1$. $x_{\pi^{-1}(1)} \in Y_{\pi^{-1}(1)} \Rightarrow s_1 = t_1$; $x_{\pi^{-1}(2)} \notin Y_{\pi^{-1}(2)} \Rightarrow s_2 \neq t_2$; $x_{\pi^{-1}(3)} \in Y_{\pi^{-1}(3)} \Rightarrow s_3 = t_3$; $x_{\pi^{-1}(4)} \notin Y_{\pi^{-1}(4)} \Rightarrow s_4 \neq t_4$.

So far, we obtain a generic construction of permuted non-membership conditional randomness generation (pnMCRG) from permuted membership conditional randomness generation (pMCRG) and non-equality conditional randomness generation (nECRG): pMCRG generates shuffled equal indication values $s_i = t_i$ of memberships $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$ and shuffled unequal indication values $s_i \neq t_i$ of non-memberships $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, and then nECRG exchanges the equality conditions ($s_i = t_i \Rightarrow u_i \neq v_i$) and non-equality conditions ($s_i \neq t_i \Rightarrow u_i = v_i$). Thus, for non-membership $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, both parties obtains equal random values $u_i = v_i$. For membership $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, both parties obtain unequal random values $u_i \neq v_i$.

1.2.3 Constructions of pMCRG

Here, we give two constructions of permuted membership conditional randomness generation (pMCRG) in balanced and unbalanced settings, respectively.

Balanced pMCRG. The first construction based on bOPRF, OKVS, and permuted equality conditional randomness generation (pECRG) in the balanced case is as follows.

First, both parties input $(x_1, x_2, \cdots, x_m)$ and $(Y_1, Y_2, \cdots, Y_m)$, and then invoke $\mathcal{F}_{\text{bOPRF}}$. The result is that the sender obtains all PRF values $F(k_i, x_i)$ and the receiver obtains all PRF keys $k_1, k_2, \cdots, k_m$. The receiver then computes all PRF values $F(k_i, Y_i[j])$ by the PRF key k_i, where $Y_i[j]$ denotes the j-th item in the i-th set Y_i. Subsequently, the receiver chooses m random indication values d_i, $i \in [m]$ and encodes all key-value pairs $\{(Y_i[j], d_i \oplus F(k_i, Y_i[j]))\}_{i \in [m], j \in [|Y_i|]}$ into a OKVS data structure D. The receiver sends D to the sender. For each $i \in [m]$, the sender inputs $F(k_i, x_i)$ and runs Decode to output $e_i = F(k_i, x_i) \oplus \mathsf{Decode}(x_i)$. According to the correctness of bOPRF and OKVS, we obtain a batched membership conditional randomness generation (MCRG): For each $i \in [m]$, if $x_i \in Y_i$, we have $e_i = F(k_i, x_i) \oplus \mathsf{Decode}(x_i) = F(k_i, x_i) \oplus d_i \oplus F(k_i, x_i) = d_i$, otherwise, $e_i \neq d_i$.

Given the above, it remains to investigate how to realize the shuffled operation. Here, we formalize a new functionality called permuted equality conditional randomness generation (pECRG), depicted in Figure 5. In the pECRG, the sender inputs a vector $(e_1, e_2, \cdots, e_m)$ and the receiver inputs a vector $(d_1, d_2, \cdots, d_m)$. As a result, both parties output permuted equality and non-equality conditions, where the sender outputs a random vector $(s_1, s_2, \cdots, s_m)$ and the receiver outputs a random vector $(t_1, t_2, \cdots, t_m)$, so that if $e_{\pi^{-1}(i)} = d_{\pi^{-1}(i)}$, $s_i = t_i$, otherwise, $s_i \neq t_i$. In Section 3, we show that pECRG can be realized from the DDH assumption and achieves linear complexity.

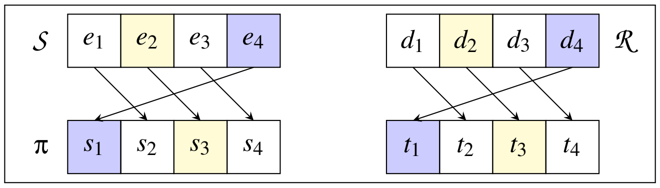

Figure 5: Illustration of pECRG. $\pi(1) = 2, \pi(2) = 3, \pi(3) = 4, \pi(4) = 1$. $e_{\pi^{-1}(1)} = d_{\pi^{-1}(1)} \Rightarrow s_1 = t_1$; $e_{\pi^{-1}(2)} \neq d_{\pi^{-1}(2)} \Rightarrow s_2 \neq t_2$; $e_{\pi^{-1}(3)} = d_{\pi^{-1}(3)} \Rightarrow s_3 = t_3$; $e_{\pi^{-1}(4)} \neq d_{\pi^{-1}(4)} \Rightarrow s_4 \neq t_4$.

Therefore, we obtain a construction of pMCRG from bOPRF, OKVS, and pECRG: For membership $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, we have $e_{\pi^{-1}(i)} = d_{\pi^{-1}(i)}$ and $s_i = t_i$. For non-membership $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, we have $e_{\pi^{-1}(i)} \neq d_{\pi^{-1}(i)}$ and $s_i \neq t_i$. Note that pMCRG inherits the linear complexity of bOPRF, OKVS, and pECRG.

Unbalanced pMCRG. The second construction based on bOPRF, FHE, and permuted equality conditional randomness generation (pECRG) in the unbalanced case is as follows.

Here, we use the polynomial randomization method following [13, 45] to encode each set Y_i, $i \in [m]$ of the receiver, so that $f_i(y) = \Pi_{j=1}^{B_i}(y - Y_i[j]) + r_i$, where $B_i = |Y_i|$ and r_i is a random value. The sender sends a FHE ciphertext of encrypting x_i, denoted as $[\![x_i]\!]$ to the receiver. Then, the receiver homomorphically computes and returns $[\![f_i(x_i)]\!]$. Finally, the sender decrypts $[\![f_i(x_i)]\!]$ and outputs $e_i = f_i(x_i)$. The receiver outputs $d_i = r_i$.[5] As discussed above, we obtain a batched membership conditional randomness generation (MCRG) in the unbalanced case: For each $i \in [m]$, if $x_i \in Y_i$, we have

[5]Following [13, 17, 45], we can use a bOPRF to compute the items on both sides before engaging in the pMCRG, which prevents the sender from learning anything about the original items and allows efficient FHE parameters.

$f_i(x_i) = \Pi_{j=1}^{B_i}(x_i - Y_i[j]) + r_i = r_i$, and $e_i = r_i = d_i$, otherwise, $e_i \neq d_i$.

Therefore, we obtain an unbalanced construction of pMCRG from bOPRF, FHE, and pECRG: For membership $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, we have $e_{\pi^{-1}(i)} = d_{\pi^{-1}(i)}$ and $s_i = t_i$. For non-membership $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, we have $e_{\pi^{-1}(i)} \neq d_{\pi^{-1}(i)}$ and $s_i \neq t_i$. Note that similar to the frameworks of [13, 17, 45], pMCRG achieves sublinear communication in the size of the larger set.

1.2.4 Constructions of nECRG

Following [34], we construct non-equality conditional randomness generation (nECRG), based on secret-shared private equality test (ssPEQT) and random oblivious transfer (ROT), achieving linear complexity.

nECRG from ssPEQT and ROT. First, both parties input their vectors $(s_1, s_2, \cdots, s_m)$ and $(t_1, t_2, \cdots, t_m)$, and then invoke the functionality of ssPEQT $\mathcal{F}_{\text{ssPEQT}}$. The result is that both parties obtain bit vectors $(a_1, a_2, \cdots, a_m)$ and $(b_1, b_2, \cdots, b_m)$, so that if $s_i = t_i$, $a_i \oplus b_i = 0$, otherwise, $a_i \oplus b_i = 1$. Next, for each $i \in [m]$, the receiver inputs a bit b_i and invokes the functionality of ROT $\mathcal{F}_{\text{ROT}}$ with the sender, so that the receiver obtains r_{i,b_i} and the sender obtains $(r_{i,0}, r_{i,1})$. Finally, the sender outputs $u_i = r_{i,(a_i \oplus 1)}$, and the receiver outputs $v_i = r_{i,b_i}$.

As a result, we obtain the construction of nECRG: For $s_i \neq t_i$, we have $a_i \oplus b_i = 1$, and $u_i = r_{i,(a_i \oplus 1)} = r_{i,b_i} = v_i$. For $s_i = t_i$, we have $a_i \oplus b_i = 0$, and $u_i = r_{i,(a_i \oplus 1)} \neq r_{i,b_i} = v_i$. Note that nECRG inherits the linear complexity of ssPEQT and ROT.

1.3 Related Work

Here, we review previous scalable PSU protocols that are relevant to our work. Kolesnikov et al. [32] propose the notion of reverse private membership test (RPMT), then use it to build a PSU protocol whose performance is much better than AHE-based PSU [19]. Garimella et al. [23] present a framework for all private set operations from permuted characteristics, which could be viewed as a variant of RPMT. After that, Jia et al. [30] also employ the shuffling technique to develop a generalized reversed private membership test (gRPMT) and give a PSU construction. Recently, Zhang et al. [48] extend the notion of RPMT [32] to multi-query RPMT (mqRPMT), and propose a generic construction of mqRPMT from the oblivious key-value store (OKVS) [24], set-membership encryption and the vector oblivious decryption-then-matching protocol. By instantiating their generic construction from symmetric-key and public-key encryption, respectively, they obtain two concrete mqRPMT protocols with linear complexity, yielding two linear PSU protocols. However, their two mqRPMT protocols have a large multiplicative constant (the statistical security parameter) in computation complexity, and so do the resulting PSU protocols. Then, Chen et al. [16] present two generic constructions of mqRPMT from commutative weak PRF (cwPRF) and permuted oblivious PRF (pOPRF), respectively. Both can be realized from DDH-like assumptions, yielding mqRPMT constructions with linear complexity. Thus, their PSU protocols inherit the linear complexity. Considering the unbalanced case, Tu et al. [45, 46] introduce a permuted multi-point private equality test (pmpPEQT) and combine it with fully homomorphic encryption (FHE) to construct an unbalanced mqRPMT, yielding the PSU protocol whose communication is linear in the small set, and logarithmic in the large set. Jia et al. [29] point out that all mqRPMT-based PSU protocols are limited by the underlying mqRPMT leaking member information and suffer from during-execution leakage. Then, they define an enhanced PSU functionality and give a construction based on oblivious programmable PRF, batched equality conditional random generation, and permuted + share. However, the complexity of their PSU scales superlinearly with the size of the set.

2 Preliminaries

2.1 Notation

For $n \in \mathbb{N}$, let $[n]$ denote the set $\{1, 2, \cdots, n\}$. 1^λ denotes the string of λ ones. We use κ and λ 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 lowercase bold letters, e.g. $\mathbf{s}$.

2.2 Enhanced Private Set Union

An enhanced PSU (ePSU) [29] can be seen as a PSU with enhanced security. We review the ideal functionality of ePSU in Figure 6.

2.3 Building Blocks

Random oblivious transfer. Oblivious transfer (OT) [41] is a central cryptographic primitive in the area of MPC. In random oblivious transfer (ROT), the sender outputs random messages, rather than selecting them as in standard OT. We recall the 1-out-of-2 random oblivious transfer functionality $\mathcal{F}_{\text{ROT}}$ in Figure 7.

Batched oblivious pseudorandom function. Oblivious pseudorandom function (OPRF) [22] is a central primitive in the area of PSO. A batched OPRF (bOPRF) allows a receiver to input $\{x_i\}_{i \in [m]}$ and obtains all PRF values $\{F(k_i, x_i)\}_{i \in [m]}$, and the keys $\{k_i\}_{i \in [m]}$ is known to a sender. Kolesnikov et al. [31] propose an efficient bOPRF and give a construction of private set intersection. We recall the bOPRF functionality $\mathcal{F}_{\text{bOPRF}}$ in Figure 8.

Parameters: The functionality interacts with two parties, the sender $\mathcal{S}$ with input $X = \{x_1, \cdots, x_m\} \subseteq \{0,1\}^*$ and the receiver $\mathcal{R}$ with input $Y = \{y_1, \cdots, y_n\} \subseteq \{0,1\}^*$, and the simulator Sim.
Functionality $\mathcal{F}_{\text{ePSU}}^{m,n}$:

1. Initialize an ideal state $\text{state}_{\mathcal{U}} = \emptyset$ for party $\mathcal{U}$ where $\mathcal{U} \in \{\mathcal{S}, \mathcal{R}\}$; if $\mathcal{U}$ is corrupted, the simulator Sim is allowed to access $\text{state}_{\mathcal{U}}$.
2. Upon receiving input $X = \{x_1, \cdots, x_m\}$ from the sender $\mathcal{S}$, abort if $|X| \neq m$; otherwise, update state $\text{state}_{\mathcal{S}} = \langle X \rangle$, and send $\langle \text{Request}, \mathcal{S} \rangle$ to Sim.
3. Upon receiving input $Y = \{y_1, \cdots, y_n\}$ from the receiver $\mathcal{R}$, abort if $|Y| \neq n$; otherwise, update state $\text{state}_{\mathcal{R}} = \langle Y \rangle$, and send $\langle \text{Request}, \mathcal{R} \rangle$ to Sim.
4. Upon receiving $\langle \text{Response}, \text{OK} \rangle$ from Sim, compute $Z = X \cup Y$, and add $\langle \text{Finished} \rangle$ to the sender's state $\text{state}_{\mathcal{S}}$ and $\langle Z \rangle$ to the receiver's state $\text{state}_{\mathcal{R}}$.
5. Output Z to $\mathcal{R}$, and $\langle \text{Finished} \rangle$ to $\mathcal{S}$.

Figure 6: Ideal functionality for enhanced private set union

Parameters: Two parties: $\mathcal{S}$ and $\mathcal{R}$. The message length l.
Functionality $\mathcal{F}_{\text{ROT}}$:

1. Wait for input b from $\mathcal{R}$.
2. Send $\langle \text{Request} \rangle$ to the simulator Sim.
3. Upon receiving $\langle \text{Response}, \text{OK} \rangle$ from Sim, sample $r_0, r_1 \leftarrow \{0,1\}^l$. Give (r_0, r_1) to $\mathcal{S}$ and give r_b to $\mathcal{R}$.

Figure 7: 1-out-of-2 random oblivious transfer functionality

Secret-shared private equality test. Secret-Shared Private Equality Test (ssPEQT) can be seen as a secret share of private equality test (PEQT). More concretely, the two parties $\mathcal{S}$ and $\mathcal{R}$ hold strings x_0 and x_1, respectively. ssPEQT outputs random bits a to $\mathcal{S}$ and b to $\mathcal{R}$ such that if $x_0 = x_1$, $a \oplus b = 0$, otherwise $a \oplus b = 1$. Existing works [12, 18, 20, 34] design linear ssPEQT protocols. We give the functionality $\mathcal{F}_{\text{ssPEQT}}$ in Figure 9.

Oblivious key-value stores. The oblivious key-value store [8, 24, 37, 42] is a data structure that compactly represents a desired mapping from a set of keys to corresponding values. The definition is as follows:

Definition 1. *An OKVS is parameterized by a set $\mathcal{K}$ of keys, a set $\mathcal{V}$ of values, and consists of two algorithms:*

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

1. Wait for input $\{x_1, \cdots, x_m\}$ from $\mathcal{R}$.
2. Sample random PRF keys $\{k_1, \cdots, k_m\}$ and compute $\{F(k_1, x_1), \cdots, F(k_m, x_m)\}$.
3. Send $\langle \text{Request} \rangle$ to the simulator Sim.
4. Upon receiving $\langle \text{Response}, \text{OK} \rangle$ from Sim, send the keys $\{k_1, \cdots, k_m\}$ to $\mathcal{S}$ and send $\{F(k_1, x_1), \cdots, F(k_m, x_m)\}$ to $\mathcal{R}$.

Figure 8: Batched oblivious pseudorandom function functionality

Parameters: Two parties: $\mathcal{S}$ and $\mathcal{R}$.
Functionality $\mathcal{F}_{\text{ssPEQT}}$:

1. Wait for the input x_0 from $\mathcal{S}$.
2. Wait for the input x_1 from $\mathcal{R}$.
3. Generate two random bits a and b such that if $x_0 = x_1$, $a \oplus b = 0$, otherwise, $a \oplus b = 1$.
4. Send $\langle \text{Request} \rangle$ to the simulator Sim.
5. Upon receiving $\langle \text{Response}, \text{OK} \rangle$ from Sim, send a to $\mathcal{S}$, and b to $\mathcal{R}$.

Figure 9: Secret-shared private equality test functionality

- $\text{Encode}(\{(k_1, v_1), \cdots, (k_n, v_n)\})$*: On input key-value pairs* $\{(k_i, v_i)\}_{i \in [n]} \subseteq \mathcal{K} \times \mathcal{V}$*, outputs an object* D *(or, with statistically small probability, an error* $\perp$*).*
- $\text{Decode}(D, k)$*: On input* D *and a key* k*, outputs* $v \in \mathcal{V}$*.*

Correctness. For all $A \subseteq \mathcal{K} \times \mathcal{V}$ with distinct keys: $(k, v) \in A$ and $\perp \neq D \leftarrow \text{Encode}(A) \Rightarrow \text{Decode}(D, k) = v$.

Obliviousness. For all distinct $\{k_1^0, \cdots, k_n^0\}$ and all distinct $\{k_1^1, \cdots, k_n^1\}$, if Encode does not output $\perp$ for $\{k_1^0, \cdots, k_n^0\}$ and $\{k_1^1, \cdots, k_n^1\}$, then the distribution of $\{D | v_i \leftarrow \mathcal{V}, i \in [n], \text{Encode}((k_1^0, v_1), \cdots, (k_n^0, v_n))\}$ is computationally indistinguishable to $\{D | v_i \leftarrow \mathcal{V}, i \in [n], \text{Encode}((k_1^1, v_1), \cdots, (k_n^1, v_n))\}$.

Randomness. We also require an additional randomness property [48] from the OKVS. For any $A = \{(k_1, v_1), \cdots, (k_n, v_n)\}$ and $k^* \notin \{k_1, \cdots, k_n\}$, the output of $\text{Decode}(D, k^*)$ is indistinguishable to that of uniform distribution over V, where $D \leftarrow \text{Encode}(A)$.

Hash-to-bin from cuckoo/simple hash. The hash-to-bin from cuckoo/simple hash technique was introduced by Pinkas et al. [38, 40], which is originally applied to construct PSI [13, 15, 17, 31, 36, 37, 39, 42] and PSU [23, 29, 30, 45]. At the high level, the sender uses hash functions $h_1, h_2, h_3 : \{0,1\}^* \to [m_c]$ to assign its items $X = \{x_i\}_{i\in[m]}$ to m_c bins $\{X_c[1], \cdots, X_c[m_c]\}$ via cuckoo hashing [35], such that each bin has at most one item, where for each x_i there is some $\gamma \in \{1,2,3\}$ such that $X_c[h_\gamma(x_i)] = x_i||\gamma$. The receiver uses the same hash functions $h_1, h_2, h_3 : \{0,1\}^* \to [m_c]$ to assign its items $Y = \{y_j\}_{j\in[n]}$ to m_c bins $\{Y_1, \cdots, Y_{m_c}\}$ via simple hashing, where for $j \in [n]$, all items are concatenated with hash function indices $(y_j||1, y_j||2, y_j||3)$ and are inserted to the bins $(Y_{h_1(y_j)}, Y_{h_2(y_j)}, Y_{h_3(y_j)})$, respectively. Therefore, if $x_i \in Y, \exists\, j \in [n], x_i = y_j$, and $\exists\, \gamma \in \{1,2,3\}$, such that $h_\gamma(x_i) \in \{h_1(y_j), h_2(y_j), h_3(y_j)\}$ and $x_i||\gamma \in \{y_j||1, y_j||2, y_j||3\}$.[6] Following [13, 15, 45], 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 $2^{-\lambda}$.

Fully homomorphic encryption. Fully homomorphic encryption (FHE) is a family of encryption schemes that allow arbitrary operations to be performed on encrypted data without decryption. The leveled fully homomorphic encryption supports circuits of a certain bounded depth. In this work, we use an array of optimization techniques of FHE as [13, 15, 17, 45], such as batching (SIMD), windowing, and partitioning to significantly reduce the depth of the homomorphic circuit.

3 Permuted Equality Conditional Randomness Generation

In this section, we introduce a new cryptographic protocol named permuted equality conditional randomness generation (pECRG). In the pECRG, $\mathcal{S}$ inputs a vector $\mathbf{s} = [s_i]_{i\in[m]}$ and a permutation π over $[m]$, and $\mathcal{R}$ inputs a vector $\mathbf{t} = [t_i]_{i\in[m]}$. The result is that $\mathcal{S}$ and $\mathcal{R}$ obtain two random vectors $\mathbf{u} = [u_i]_{i\in[m]}$ and $\mathbf{v} = [v_i]_{i\in[m]}$, respectively, where for $i \in [m]$, if $s_{\pi^{-1}(i)} = t_{\pi^{-1}(i)}$, we have $u_i = v_i$, otherwise, $u_i \neq v_i$. In Figure 10, we define the functionality of pECRG, denoted as $\mathcal{F}_{\text{pECRG}}$.

We give a construction of pECRG from the DDH assumption, as described in Figure 11.

Theorem 1. *The construction of Figure 11 UC realizes the functionality $\mathcal{F}_{\text{pECRG}}$ based on DDH in the random oracle model, in the presence of semi-honest adversaries.*

Proof. We exhibit simulators Sim to simulate corrupt $\mathcal{S}$ and corrupt $\mathcal{R}$ respectively and argue the indistinguishability of the produced transcript from the real execution.

[6] Appending the index of the hash function is helpful for dealing with edge cases like $h_1(y) = h_2(y)$, which happen with non-negligible probability [23], ensuring that there are no identical items in the hash table.

Parameters: Two parties: $\mathcal{S}$ and $\mathcal{R}$.
Functionality $\mathcal{F}_{\text{pECRG}}$:

1. Initialize an ideal state $\mathsf{state}_{\mathcal{U}} = \emptyset$ for party $\mathcal{U}$ where $\mathcal{U} \in \{\mathcal{S}, \mathcal{R}\}$; if $\mathcal{U}$ is corrupted, the simulator Sim is allowed to access $\mathsf{state}_{\mathcal{U}}$.
2. Upon receiving input $\mathbf{s} = [s_i]_{i\in[m]}$ and a permutation π over $[m]$ from $\mathcal{S}$ and update state $\mathsf{state}_{\mathcal{S}} = \langle \mathbf{s}, \pi \rangle$, and send $\langle \mathsf{Request}, \mathcal{S} \rangle$ to Sim.
3. Upon receiving input $\mathbf{t} = [t_i]_{i\in[m]}$ from $\mathcal{R}$ and update state $\mathsf{state}_{\mathcal{R}} = \langle \mathbf{t} \rangle$, and send $\langle \mathsf{Request}, \mathcal{R} \rangle$ to Sim.
4. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, generate two random vectors $\mathbf{u} = [u_i]_{i\in[m]}$ and $\mathbf{v} = [v_i]_{i\in[m]}$, where for $i \in [m]$, if $s_{\pi^{-1}(i)} = t_{\pi^{-1}(i)}$, $u_i = v_i$, otherwise $u_i \neq v_i$.
5. Add $\langle \mathbf{u} \rangle$ to the state $\mathsf{state}_{\mathcal{S}}$ and $\langle \mathbf{v} \rangle$ to the state $\mathsf{state}_{\mathcal{R}}$.
6. Output $\mathbf{u} = [u_i]_{i\in[m]}$ to $\mathcal{S}$. Output $\mathbf{v} = [v_i]_{i\in[m]}$ to $\mathcal{R}$.

Figure 10: Permuted equality conditional randomness generation functionality

Input: $\mathcal{S}$ inputs a vector $\mathbf{s}$ and a permutation π over $[m]$. $\mathcal{R}$ inputs a vector $\mathbf{t}$. $\mathbb{G}$ is a cyclic group with order q.
Output: $\mathcal{S}$ outputs a vector $\mathbf{u}$. $\mathcal{R}$ output a vector $\mathbf{v}$.

1. $\mathcal{R}$ chooses a random value $b \leftarrow \mathbb{Z}_q$ and computes $\hat{t}_i = H(t_i)^b$ for all $i \in [m]$, where $H(\cdot)$ is modeled as a random oracle and the output is a group element in $\mathbb{G}$, and sends $\hat{\mathbf{t}} = [\hat{t}_i]_{i\in[m]}$ to $\mathcal{S}$.
2. $\mathcal{S}$ chooses a random value $a \leftarrow \mathbb{Z}_q$ and computes $t'_i = (\hat{t}_i)^a$ for all $i \in [m]$ and $s'_i = H(s_i)^a$ Then, $\mathcal{S}$ shuffles $\mathbf{t}' = [t'_i]_{i\in[m]}$ and $\mathbf{s}' = [s'_i]_{i\in[m]}$ by same permutation π and sends $\pi([s'_i]_{i\in[m]})$ to $\mathcal{R}$.
3. $\mathcal{S}$ sets and outputs $\mathbf{u} = [u_i]_{i\in[m]} = \pi([t'_i]_{i\in[m]})$. $\mathcal{R}$ sets and outputs $\mathbf{v} = [v_i]_{i\in[m]} = \pi([s'_i]_{i\in[m]})^b$.

Figure 11: DDH-based pECRG

Corrupted $\mathcal{S}$. Sim simulates the view of corrupt $\mathcal{S}$ as follows: It chooses random group elements d_i, $i \in [m]$, to simulate the view. We argue that the outputs of Sim 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 $H(t_i)^b$, $i \in [m]$, where $b \leftarrow \mathbb{Z}_q$.

Hyb_1: Same as Hyb_0 except that Sim chooses random group elements d_i, $i \in [m]$ instead of $H(t_i)^b$, $i \in [m]$, where $b \leftarrow \mathbb{Z}_q$. The hybrid is the view output by Sim.

We argue that the views in Hyb_0 and 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_i}, g^{z_i}, where $x, y_i \leftarrow \mathbb{Z}_q$, $\mathcal{A}$ is asked to distinguish if $z_i = x \cdot y_i$ or random values. $\mathcal{A}$ implicitly sets $b = x$, and simulates (with the knowledge of $\mathbf{t}$) the view as follows:

- RO queries: Sim honestly emulates random oracle (RO) H. For queries t_i, if $t_i \notin \mathbf{t}$, it picks a random group element to assign $H(t_i)$, otherwise, it assigns $H(t_i) = g^{y_i}$.
- Outputs g^{z_i}, $i \in [m]$.

Clearly, if $z_i = x \cdot y_i$, $\mathcal{A}$ simulates Hyb_0. Otherwise, it simulates Hyb_1 (without the knowledge of $\mathbf{t}$), because it responds to all RO queries with random group elements without knowing whether the inputs belong to $\mathbf{t}$ or not. Therefore, the outputs of Sim are computationally indistinguishable from the real view based on the DDH assumption.

Corrupted $\mathcal{R}$. Sim simulates the view of corrupt $\mathcal{R}$ as follows: It chooses random group elements e_i, $i \in [m]$, to simulate the view. We argue that the outputs of Sim 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 $H(s_{\pi(i)})^a$, $i \in [m]$, where $a \leftarrow \mathbb{Z}_q$.

Hyb_1: Same as Hyb_0 except that Sim chooses random group elements e_i, $i \in [m]$ instead of $H(s^a_{\pi(i)}$, $i \in [m]$, where $a \leftarrow \mathbb{Z}_q$. The hybrid is the view output by Sim.

We argue that the views in Hyb_0 and 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_i}, g^{z_i}, where $x, y_i \leftarrow \mathbb{Z}_q$, $\mathcal{A}$ is asked to distinguish if $z_i = x \cdot y_i$ or random values. $\mathcal{A}$ implicitly sets $a = x$, and simulates (with the knowledge of $\mathbf{s}$ and π) the view as follows:

- RO queries: Sim honestly emulates random oracle (RO) H. For queries s_i, if $s_i \notin \mathbf{s}$, it picks a random group element to assign $H(s_i)$, otherwise, it assigns $H(s_i) = g^{y_i}$.
- Outputs g^{z_i}, $i \in [m]$.

Clearly, if $z_i = x \cdot y_i$, $\mathcal{A}$ simulates Hyb_0. Otherwise, it simulates Hyb_1 (without the knowledge of $\mathbf{s}$ and π), because it responds to all RO queries with random group elements without knowing whether the inputs belong to $\mathbf{s}$ or not. Therefore, the outputs of Sim are computationally indistinguishable from the real view based on the DDH assumption.

4 Permuted Membership Conditional Randomness Generation

In this section, we introduce a new cryptographic protocol named permuted membership conditional randomness generation (pMCRG), in which $\mathcal{S}$ inputs a set $X = \{x_i\}_{i \in [m]}$ and a permutation π over $[m]$, and $\mathcal{R}$ inputs m sets $\{Y_i\}_{i \in [m]}$. As a result, $\mathcal{S}$ and $\mathcal{R}$ obtain two random vectors $\mathbf{s} = [s_i]_{i \in [m]}$ and $\mathbf{t} = [t_i]_{i \in [m]}$, respectively, such that for $i \in [m]$, if $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, $s_i = t_i$, otherwise, $s_i \neq t_i$. In Figure 12, we define the functionality of pMCRG, denoted as $\mathcal{F}_{\text{pMCRG}}$.

Parameters: Two parties: $\mathcal{S}$ and $\mathcal{R}$.
Functionality $\mathcal{F}_{\text{pMCRG}}$:

1. Initialize an ideal state $\mathsf{state}_{\mathcal{U}} = \emptyset$ for party $\mathcal{U}$ where $\mathcal{U} \in \{\mathcal{S}, \mathcal{R}\}$; if $\mathcal{U}$ is corrupted, the simulator Sim is allowed to access $\mathsf{state}_{\mathcal{U}}$.
2. Upon receiving input $X = \{x_i\}_{i \in [m]}$ and a permutation π over $[m]$ from $\mathcal{S}$, then update state $\mathsf{state}_{\mathcal{S}} = \langle X, \pi \rangle$, and send $\langle \mathsf{Request}, \mathcal{S} \rangle$ to Sim.
3. Upon receiving input $\{Y_i\}_{i \in [m]}$ from $\mathcal{R}$ and update state $\mathsf{state}_{\mathcal{R}} = \langle \{Y_i\}_{i \in [m]} \rangle$, and send $\langle \mathsf{Request}, \mathcal{R} \rangle$ to Sim.
4. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, generate two random vectors $\mathbf{s} = [s_i]_{i \in [m]}$ and $\mathbf{t} = [t_i]_{i \in [m]}$, where for $i \in [m]$, if $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, $s_i = t_i$, otherwise $s_i \neq t_i$.
5. Add $\langle \mathbf{t} \rangle$ to the state $\mathsf{state}_{\mathcal{S}}$ and $\langle \mathbf{s} \rangle$ to the state $\mathsf{state}_{\mathcal{R}}$.
6. Output $\mathbf{t} = [t_i]_{i \in [m]}$ to $\mathcal{S}$. Output $\mathbf{s} = [s_i]_{i \in [m]}$ to $\mathcal{R}$.

Figure 12: Permuted membership conditional randomness generation functionality

Here, we present two pMCRG constructions in the balanced and unbalanced setting. Note that since OKVS and polynomial interpolation require all keys to be distinct, our constructions require the m sets $\{Y_i\}_{i \in [m]}$ input by $\mathcal{R}$ are mutually exclusive[7], meaning they do not contain any same items.

4.1 pMCRG Construction in the Balanced Setting

We give a construction of pMCRG based on bOPRF, OKVS, and pECRG in the balanced case, as described in Figure 13.

Theorem 2. *The protocol Π_{pMCRG} shown in Figure 13 UC-realizes the functionality $\mathcal{F}_{\text{pMCRG}}$ (as in Figure 12) in the*

[7]The property of mutual exclusivity is easy to achieve: All items of the set with equal length are linked to the indices of the set.

Input: $\mathcal{S}$ inputs a set $X = \{x_i\}_{i\in[m]}$ and a permutation π over $[m]$. $\mathcal{R}$ inputs m mutually exclusive sets $\{Y_i\}_{i\in[m]}$.
Output: $\mathcal{S}$ outputs a vector $\mathbf{s} = [s_i]_{i\in[m]}$. $\mathcal{R}$ outputs a vector $\mathbf{t} = [t_i]_{i\in[m]}$.

1. $\mathcal{S}$ and $\mathcal{R}$ invoke the bOPRF functionality $\mathcal{F}_{\text{bOPRF}}$.
    (a) $\mathcal{S}$ input a set $X = \{x_i\}_{i\in[m]}$.
    (b) $\mathcal{S}$ obtains all PRF values $F(k_i, x_i)$, $i \in [m]$. $\mathcal{R}$ obtains PRF keys $\{k_1, \cdots, k_m\}$.
2. For all $i \in [m]$, $\mathcal{R}$ computes PRF values $F(k_i, Y_i[j])$, where $Y_i[j]$ denotes j-th item in Y_i.
3. $\mathcal{R}$ encodes an OKVS.
    (a) $\mathcal{R}$ chooses m random values $[d_i]_{i\in[m]}$, and defines $\mathcal{P} = \{(Y_i[j], F(k_i, Y_i[j]) \oplus d_i)\}_{i\in[m], j\in[|Y_i|]}$.
    (b) $\mathcal{R}$ computes an OKVS: $D = \mathsf{Encode}(\mathcal{P})$, and sends D to $\mathcal{S}$.
4. $\mathcal{S}$ decodes $e_i = \mathsf{Decode}(D, x_i)) \oplus F(k_i, x_i)$, $i \in [m]$.
5. $\mathcal{S}$ and $\mathcal{R}$ invoke the pECRG functionality $\mathcal{F}_{\text{pECRG}}$.
    (a) $\mathcal{S}$ inputs a permutation π on $[m]$ and a vector $[e_i]_{i\in[m]}$. $\mathcal{R}$ input a vector $[d_i]_{i\in[m]}$.
    (b) $\mathcal{S}$ outputs $\mathbf{s} = [s_i]_{i\in[m]}$. $\mathcal{R}$ outputs $\mathbf{t} = [t_i]_{i\in[m]}$.

Figure 13: pMCRG from bOPRF, OKVS, and pECRG

($\mathcal{F}_{\text{bOPRF}}$, $\mathcal{F}_{\text{pECRG}}$)-hybrid model, against static, semi-honest adversaries, provided a secure OKVS scheme.

Proof. We will show that for any adversary $\mathcal{A}$, we can construct a simulator Sim that can simulate the view of the corrupted $\mathcal{S}$ and the corrupted $\mathcal{R}$, such that any PPT environment cannot distinguish the execution in the ideal world from that in the real world.

Corrupted $\mathcal{S}$. Sim simulates a real execution in which the sender $\mathcal{S}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input X and π of $\mathcal{S}$ directly, and externally send them to $\mathcal{F}_{\text{pMCRG}}$ and then receives $\langle \mathsf{Request}, \mathcal{S} \rangle$. Once receiving the set $X = \{x_i\}_{i\in[m]}$, the input of Π_{bOPRF}, from $\mathcal{A}$, Sim randomly selects $\mathbf{x}' = [x'_i]_{i\in[m]}$ as the PRF values to simulate the execution of Π_{bOPRF}. Sim computes a random OKVS D by selecting random key-value pairs. Upon receiving a permutation π on $[m]$ and a set $[e_i]_{i\in[m]}$, the input of Π_{pECRG}, from $\mathcal{A}$, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{pMCRG}}$, and obtains $\mathbf{s} = [s_i]_{i\in[m]}$. Finally, Sim simulates the execution of Π_{pECRG} with $\mathbf{s} = [s_i]_{i\in[m]}$ as the output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{S}$ by the following hybrids:

Hyb_0: $\mathcal{S}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{bOPRF} is replaced by $\mathbf{x}'$ chosen by Sim, and Sim runs the Π_{bOPRF} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{bOPRF} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_2: Same as Hyb_1 except that computing a random OKVS D by Sim. Briefly, this simulation is indistinguishable for the following reasons: The pseudorandomness of PRF value is indistinguishable from random, and then by the obliviousness and randomness of OKVS, D is distributed uniformly.

Hyb_3: Same as Hyb_2 except that the output of Π_{pECRG} is replaced by $\mathbf{s} = [s_i]_{i\in[m]}$ output by $\mathcal{F}_{\text{pMCRG}}$, and Sim runs the Π_{pECRG} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{pECRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Corrupted $\mathcal{R}$. Sim simulates a real execution in which the receiver $\mathcal{R}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input (m mutually exclusive sets $\{Y_i\}_{i\in[m]}$) of $\mathcal{R}$, and externally send them to $\mathcal{F}_{\text{pMCRG}}$ and then receives $\langle \mathsf{Request}, \mathcal{R} \rangle$. Sim randomly selects the PRF key $\{k_1, \cdots, k_m\}$ to simulate the execution of Π_{bOPRF}. Once receiving $\mathbf{d} = [d_i]_{i\in[m]}$, the input of Π_{pECRG}, from $\mathcal{A}$, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{pMCRG}}$, and obtains $\mathbf{t} = [t_i]_{i\in[m]}$. Finally, Sim simulates the execution of Π_{pECRG} with $\mathbf{t} = [t_i]_{i\in[m]}$ as the output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:

Hyb_0: $\mathcal{R}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{bOPRF} is replaced by $\{k_1, \cdots, k_m\}$ chosen by Sim, and Sim runs the Π_{bOPRF} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{bOPRF} 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 of Π_{pECRG} is replaced by $\mathbf{t} = [t_i]_{i\in[m]}$ output by $\mathcal{F}_{\text{pMCRG}}$, and Sim runs the Π_{pECRG} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{pECRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

4.2 pMCRG Construction in the Unbalanced Setting

We give the construction of pMCRG from bOPRF, FHE and pECRG in the unbalanced case, as described in Figure 14. We use an array of optimization techniques of FHE as [13, 15, 17, 45], such as batching (SIMD), windowing, and partitioning to significantly reduce the depth of the homomorphic circuit. We review the optimizations in appendix A.

Input: $\mathcal{S}$ inputs a set $X = \{x_i\}_{i\in[m]}$ and a permutation π over $[m]$. $\mathcal{R}$ inputs m mutually exclusive sets $\{Y_i\}_{i\in[m]}$.
Output: $\mathcal{S}$ outputs a vector $\mathbf{s} = [s_i]_{i\in[m]}$. $\mathcal{R}$ outputs a vector $\mathbf{t} = [t_i]_{i\in[m]}$.

1. Both parties agree on the parameters of bOPRF, FHE, and pECRG.
2. $\mathcal{S}$ and $\mathcal{R}$ invoke the bOPRF functionality $\mathcal{F}_{\text{bOPRF}}$.
   (a) $\mathcal{S}$ input a set $X = \{x_i\}_{i\in[m]}$.
   (b) $\mathcal{S}$ obtains all PRF values $\bar{x}_i = F(k_i, x_i)$, $i \in [m]$. $\mathcal{R}$ obtains PRF keys $\{k_1, \cdots, k_m\}$.
3. For $i \in [m]$, $\mathcal{R}$ computes PRF values $Y'_i[j] = F(k_i, Y_i[j])$, where $Y_i[j]$ denotes j-th item, $B_i = |Y_i|$.
4. $\mathcal{R}$ chooses a random vector $\mathbf{d} = [d_i]_{i\in[m]}$. For all $i \in [m]$, $\mathcal{R}$ computes polynomials $F_i(x) = f_i(x) + d_i$, where for all $j \in [B_i]$, $f_i(Y'_i[j]) = 0$. Thus, $\mathcal{R}$ obtains the coefficient matrix $\mathbf{A}$, where i-th column of $\mathbf{A}$ are the coefficients of F_i.
5. $\mathcal{S}$ uses its FHE public key to encrypt $\bar{\mathbf{x}} = [\bar{x}_i]_{i\in[m]}$ and sends ciphertexts $[\![\bar{x}_i]\!]$, $i \in [m]$ to $\mathcal{R}$.
6. For each $[\![\bar{x}_i]\!]$, $\mathcal{R}$ homomorphically computes encryptions of all powers $\mathbf{C}_i = [[\![0]\!], [\![\bar{x}_i^1]\!], \cdots, [\![\bar{x}_i^{B_i}]\!]]$. Then, $\mathcal{R}$ homomorphically evaluates $\bar{\mathbf{C}}_i = \mathbf{C}_i \bigotimes \mathbf{A}_i$, and sends all ciphertexts to $\mathcal{S}$.
7. $\mathcal{S}$ decrypts the ciphertexts into $\mathbf{e} = [e_i]_{i\in[m]}$.
8. $\mathcal{S}$ and $\mathcal{R}$ invoke the pECRG functionality $\mathcal{F}_{\text{pECRG}}$.
   (a) $\mathcal{S}$ inputs a permutation π on $[m]$ and a vector $\mathbf{e} = [e_i]_{i\in[m]}$. $\mathcal{R}$ input a vector $\mathbf{d} = [d_i]_{i\in[m]}$.
   (b) $\mathcal{S}$ outputs $\mathbf{s} = [s_i]_{i\in[m]}$. $\mathcal{R}$ outputs $\mathbf{t} = [t_i]_{i\in[m]}$.

Figure 14: pMCRG from bOPRF, FHE and pECRG

Theorem 3. *The protocol Π_{pMCRG} shown in Figure 14 UC-realizes the functionality $\mathcal{F}_{\text{pMCRG}}$ (as in Figure 12) in the $(\mathcal{F}_{\text{bOPRF}}, \mathcal{F}_{\text{pECRG}})$-hybrid model, against static, semi-honest adversaries, provided that the fully homomorphic encryption scheme is IND-CPA secure.*

Proof. We will show that for any adversary $\mathcal{A}$, we can construct a simulator Sim that can simulate the view of the corrupted $\mathcal{S}$ and the corrupted $\mathcal{R}$, such that any PPT environment cannot distinguish the execution in the ideal world from that in the real world.

Corrupted $\mathcal{S}$. Sim simulates a real execution in which the sender $\mathcal{S}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input X and π of $\mathcal{S}$ directly, and externally send them to $\mathcal{F}_{\text{pMCRG}}$ and then receives $\langle \text{Request}, \mathcal{S} \rangle$. Once receiving $X = \{x_i\}_{i\in[m]}$, the input of Π_{bOPRF}, from $\mathcal{A}$, Sim randomly selects $\mathbf{x}' = [x'_i]_{i\in[m]}$ to simulate the execution of Π_{bOPRF}. Upon receiving all ciphertexts $[\![x'_i]\!]$, $i \in [m]$, from $\mathcal{A}$, Sim randomly selects $\mathbf{r} = [r_i]_{i\in[m]}$ and encrypts them to simulate the ciphertexts. Once receiving $\mathbf{r} = [r_i]_{i\in[m]}$ and π, the input of Π_{pECRG}, from $\mathcal{A}$, Sim sends $\langle \text{Response}, \text{OK} \rangle$ to $\mathcal{F}_{\text{pMCRG}}$, and obtains $\mathbf{s} = [s_i]_{i\in[m]}$. Finally, Sim simulates the execution of Π_{pECRG} with $\mathbf{s} = [s_i]_{i\in[m]}$ as the output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{S}$ by the following hybrids:

Hyb_0: $\mathcal{S}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{bOPRF} is replaced by $\mathbf{x}'$ chosen by Sim, and Sim runs the Π_{bOPRF} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{bOPRF} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_2: Same as Hyb_1 except that encrypting random values in place of the ciphertexts in step 5. The plaintexts are randomized in the real view which is indistinguishable from the random values in the simulated view.

Hyb_3: Same as Hyb_2 except that the output of Π_{pECRG} is replaced by $\mathbf{s} = [s_i]_{i\in[m]}$ output by $\mathcal{F}_{\text{pMCRG}}$, and Sim runs the Π_{pECRG} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{pECRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Corrupted $\mathcal{R}$. Sim simulates a real execution in which the receiver $\mathcal{R}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input (m mutually exclusive sets $\{Y_i\}_{i\in[m]}$) of $\mathcal{R}$ directly, and externally send them to $\mathcal{F}_{\text{pMCRG}}$ and then receives $\langle \text{Request}, \mathcal{R} \rangle$. Sim randomly selects the PRF key $\{k_1, \cdots, k_m\}$ to simulate the execution of Π_{bOPRF}, and then encrypts random value in place of the ciphertexts in step 6. Once receiving $\mathbf{d} = [d_i]_{i\in[m]}$, the input of Π_{pECRG}, from $\mathcal{A}$, Sim sends $\langle \text{Response}, \text{OK} \rangle$ to $\mathcal{F}_{\text{pMCRG}}$, and obtains $\mathbf{t} = [t_i]_{i\in[m]}$. Finally, Sim simulates the execution of Π_{pECRG} with $\mathbf{t} = [t_i]_{i\in[m]}$ as output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:

Hyb_0: $\mathcal{R}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{bOPRF} is replaced by $\{k_1, \cdots, k_m\}$ chosen by Sim, and Sim runs the Π_{bOPRF} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{bOPRF} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_2: Same as Hyb_1 except that encrypting random values in place of the ciphertexts in step 6. The IND-CPA security of the fully homomorphic encryption scheme guarantees that the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_3: Same as Hyb_2 except that the output of Π_{pECRG} is replaced by $\mathbf{t} = [t_i]_{i\in[m]}$ output by $\mathcal{F}_{\text{pMCRG}}$, and Sim runs the Π_{pECRG} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{pECRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Parameters: Two parties: $\mathcal{S}$ and $\mathcal{R}$.
Functionality $\mathcal{F}_{\text{nECRG}}$:

1. Initialize an ideal state $\mathsf{state}_{\mathcal{U}} = \emptyset$ for party $\mathcal{U}$ where $\mathcal{U} \in \{\mathcal{S}, \mathcal{R}\}$; if $\mathcal{U}$ is corrupted, the simulator Sim is allowed to access $\mathsf{state}_{\mathcal{U}}$.
2. Upon receiving input $\mathbf{s} = [s_i]_{i\in[m]}$ from $\mathcal{S}$ and update state $\mathsf{state}_{\mathcal{S}} = \langle \mathbf{s} \rangle$, and send $\langle \mathsf{Request}, \mathcal{S} \rangle$ to Sim.
3. Upon receiving input $\mathbf{t} = [t_i]_{i\in[m]}$ from $\mathcal{R}$ and update state $\mathsf{state}_{\mathcal{R}} = \langle \mathbf{t} \rangle$, and send $\langle \mathsf{Request}, \mathcal{R} \rangle$ to Sim.
4. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, generate two random vectors $\mathbf{u} = [u_i]_{i\in[m]}$ and $\mathbf{v} = [v_i]_{i\in[m]}$, where for $i \in [m]$, if $s_i = t_i$, $u_i \neq v_i$, otherwise $u_i = v_i$.
5. Add $\langle \mathbf{u} \rangle$ to the state $\mathsf{state}_{\mathcal{S}}$ and $\langle \mathbf{v} \rangle$ to the state $\mathsf{state}_{\mathcal{R}}$.
6. Output $\mathbf{u} = [u_i]_{i\in[m]}$ to $\mathcal{S}$. Output $\mathbf{v} = [v_i]_{i\in[m]}$ to $\mathcal{R}$.

Figure 15: Non-equality conditional randomness generation functionality

5 Non-Equality Conditional Randomness Generation

Here, we introduce a new cryptographic protocol named non-equality conditional randomness generation (nECRG). In the nECRG, $\mathcal{S}$ and $\mathcal{R}$ input two vectors $\mathbf{s} = [s_i]_{i\in[m]}$ and $\mathbf{t} = [t_i]_{i\in[m]}$, respectively. The result is that $\mathcal{S}$ and $\mathcal{R}$ obtain two random vectors $\mathbf{u} = [u_i]_{i\in[m]}$ and $\mathbf{v} = [v_i]_{i\in[m]}$, such that for $i \in [m]$, if $s_i = t_i$, $u_i \neq v_i$, otherwise, $u_i = v_i$. In Figure 15, we define the functionality of nECRG, denoted as $\mathcal{F}_{\text{nECRG}}$.

We give a construction of nECRG based on ssPEQT and ROT as described in Figure 16.

Theorem 4. *The protocol Π_{nECRG} shown in Figure 16 UC-realizes the functionality $\mathcal{F}_{\text{nECRG}}$ (as in Figure 15) in the ($\mathcal{F}_{\text{ssPEQT}}$, $\mathcal{F}_{\text{ROT}}$)-hybrid model, against static, semi-honest adversaries.*

Proof. We will show that for any adversary $\mathcal{A}$, we can construct a simulator Sim that can simulate the view of the corrupted $\mathcal{S}$ and the corrupted $\mathcal{R}$, such that any PPT environment cannot distinguish the execution in the ideal world from that in the real world.

Corrupted $\mathcal{S}$. Sim simulates a real execution in which the sender $\mathcal{S}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input $\mathbf{s}$ of $\mathcal{S}$ directly, and externally send $\mathbf{s}$ to $\mathcal{F}_{\text{nECRG}}$ and then receives $\langle \mathsf{Request}, \mathcal{S} \rangle$. Once receiving $\mathbf{s} = [s_i]_{i\in[m]}$, the input of Π_{ssPEQT}, from $\mathcal{A}$, Sim randomly chooses a bit vector $\mathbf{a} = [a_i]_{i\in[m]}$ to simulate the execution of Π_{ssPEQT}. Sim

Input: $\mathcal{S}$ inputs a vector $\mathbf{s} = [s_i]_{i\in[m]}$. $\mathcal{R}$ inputs a vector $\mathbf{t} = [t_i]_{i\in[m]}$.
Output: $\mathcal{S}$ outputs a vector $\mathbf{u} = [u_i]_{i\in[m]}$. $\mathcal{R}$ outputs a vector $\mathbf{v} = [v_i]_{i\in[m]}$.

1. $\mathcal{S}$ and $\mathcal{R}$ invoke the ssPEQT functionality $\mathcal{F}_{\text{ssPEQT}}$.
   (a) $\mathcal{S}$ and $\mathcal{R}$ input vectors $\mathbf{s} = [s_i]_{i\in[m]}$ and $\mathbf{t} = [t_i]_{i\in[m]}$.
   (b) $\mathcal{S}$ and $\mathcal{R}$ output random bit vectors $\mathbf{a} = [a_i]_{i\in[m]}$ and $\mathbf{b} = [b_i]_{i\in[m]}$, where for all $i \in [m]$, if $s_i = t_i$, $a_i \oplus b_i = 0$, otherwise $a_i \oplus b_i = 1$.
2. $\mathcal{S}$ and $\mathcal{R}$ invoke the ROT functionality $\mathcal{F}_{\text{ROT}}$.
   (a) For each $i \in [m]$, $\mathcal{R}$ inputs b_i.
   (b) $\mathcal{S}$ obtains $r_{i,0}$ and $r_{i,1}$, $i \in [m]$. $\mathcal{R}$ obtains r_{i,b_i}, $i \in [m]$.
3. For all $i \in [m]$, $\mathcal{S}$ sets $u_i = r_{i,(a_i \oplus 1)}$. $\mathcal{R}$ sets $v_i = r_{i,b_i}$.
4. $\mathcal{S}$ outputs a vector $\mathbf{u} = [u_i]_{i\in[m]}$. $\mathcal{R}$ outputs a vector $\mathbf{v} = [v_i]_{i\in[m]}$.

Figure 16: nECRG from ssPEQT and ROT

sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{nECRG}}$, and obtains $\mathbf{u} = [u_i]_{i\in[m]}$. For all $i \in [m]$, Sim sets $r_{i,(a_i \oplus 1)} = u_i$ and randomly chooses r_{i,a_i}. Finally, Sim simulates the execution of Π_{ROT} with

$(r_{i,(a_i \oplus 1)}, r_{i,a_i})$ for each $i \in [m]$ as output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{S}$ by the following hybrids:

Hyb_0: $\mathcal{S}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{ssPEQT} is replaced by the random bit vector $\mathbf{a} = [a_i]_{i \in [m]}$ chosen by Sim, and Sim runs the Π_{ssPEQT} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{ssPEQT} 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 of Π_{ROT} is replaced by $u_i, i \in [m]$ output by $\mathcal{F}_{\text{nECRG}}$ and random $r_{i,a_i}, i \in [m]$ chosen by Sim, and Sim runs the Π_{ROT} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{ROT} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Corrupted $\mathcal{R}$. Sim simulates a real execution in which the receiver $\mathcal{R}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain input $\mathbf{t}$ of $\mathcal{R}$ directly, and externally send $\mathbf{t}$ to $\mathcal{F}_{\text{nECRG}}$ and then receives $\langle \mathsf{Request}, \mathcal{R} \rangle$. Once receiving $\mathbf{t} = [t_i]_{i \in [m]}$, the input of Π_{ssPEQT}, from $\mathcal{A}$, Sim randomly chooses a bit vector $\mathbf{b} = [b_i]_{i \in [m]}$ to simulate the execution of Π_{ssPEQT}. Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{nECRG}}$, and obtains $\mathbf{v} = [v_i]_{i \in [m]}$. For all $i \in [m]$, Sim sets $r_{i,b_i} = v_i$. Finally, Sim simulates the execution of Π_{ROT} with (b_i, v_i) for each $i \in [m]$ as output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:

Hyb_0: $\mathcal{R}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{ssPEQT} is replaced by the random bit vector $\mathbf{b} = [b_i]_{i \in [m]}$ chosen by Sim, and Sim runs the Π_{ssPEQT} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{ssPEQT} 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 of Π_{ROT} is replaced by v_i, $i \in [m]$ output by $\mathcal{F}_{\text{nECRG}}$, and Sim runs the Π_{ROT} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{ROT} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Comparison with bECRG. To construct enhanced PSU, Jia et al. [29] introduce a new functionality called batched equality conditional randomness generation (bECRG), in which both parties can also achieve the exchange of the equality and non-equality conditions. However, bECRG limits the output of $\mathcal{S}$ and only generates random values of $\mathcal{R}$, resulting in its construction only based on the complicated equality oblivious transfer extension protocol. Our nECRG generates corresponding random values for $\mathcal{S}$ and $\mathcal{R}$, according to the equality and non-equality conditions between the inputs of both parties and allows the construction based on the simple random oblivious transfer protocol following [34].

6 Permuted Non-Membership Conditional Randomness Generation

In this section, we introduce a new cryptographic protocol named permuted non-membership conditional randomness generation (pnMCRG), in which $\mathcal{S}$ inputs a set $X = \{x_i\}_{i \in [m]}$ and a permutation π over $[m]$, and $\mathcal{R}$ inputs m sets $\{Y_i\}_{i \in [m]}$, respectively. As a result, $\mathcal{S}$ and $\mathcal{R}$ obtain two random vectors $\mathbf{u} = [u_i]_{i \in [m]}$ and $\mathbf{v} = [v_i]_{i \in [m]}$, such that for $i \in [m]$, if $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, $u_i = v_i$, otherwise, $u_i \neq v_i$. In Figure 17, we define the functionality of pnMCRG, denoted as $\mathcal{F}_{\text{pnMCRG}}$.

Parameters: Two parties: $\mathcal{S}$ and $\mathcal{R}$.
Functionality $\mathcal{F}_{\text{pnMCRG}}$:

1. Initialize an ideal state $\mathsf{state}_{\mathcal{U}} = \emptyset$ for party $\mathcal{U}$ where $\mathcal{U} \in \{\mathcal{S}, \mathcal{R}\}$; if $\mathcal{U}$ is corrupted, the simulator Sim is allowed to access $\mathsf{state}_{\mathcal{U}}$.
2. Upon receiving input $X = \{x_i\}_{i \in [m]}$ and a permutation π over $[m]$ from $\mathcal{S}$, then update state $\mathsf{state}_{\mathcal{S}} = \langle X, \pi \rangle$, and send $\langle \mathsf{Request}, \mathcal{S} \rangle$ to Sim.
3. Upon receiving input $\{Y_i\}_{i \in [m]}$ from $\mathcal{R}$, and update state $\mathsf{state}_{\mathcal{R}} = \langle \{Y_i\}_{i \in [m]} \rangle$, and send $\langle \mathsf{Request}, \mathcal{R} \rangle$ to Sim.
4. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, generate two random vectors $\mathbf{u} = [u_i]_{i \in [m]}$ and $\mathbf{v} = [v_i]_{i \in [m]}$, where for $i \in [m]$, if $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$, $u_i = v_i$, otherwise $u_i \neq v_i$.
5. Add $\langle \mathbf{u} \rangle$ to the state $\mathsf{state}_{\mathcal{S}}$ and $\langle \mathbf{v} \rangle$ to the state $\mathsf{state}_{\mathcal{R}}$.
6. Output $\mathbf{u} = [u_i]_{i \in [m]}$ to $\mathcal{S}$. Output $\mathbf{v} = [v_i]_{i \in [m]}$ to $\mathcal{R}$.

Figure 17: Permuted non-membership conditional randomness generation functionality

We give a construction of pnMCRG based on pMCRG and nECRG as described in Figure 18.

Theorem 5. *The protocol Π_{pnMCRG} shown in Figure 18 UC-realizes the functionality $\mathcal{F}_{\text{pnMCRG}}$ (as in Figure 17) in the $(\mathcal{F}_{\text{pMCRG}}, \mathcal{F}_{\text{nECRG}})$-hybrid model, against static, semi-honest adversaries.*

Proof. We will show that for any adversary $\mathcal{A}$, we can construct a simulator Sim that can simulate the view of the cor-

Input: $\mathcal{S}$ inputs a set $X = \{x_i\}_{i\in[m]}$ and a permutation π over $[m]$. $\mathcal{R}$ inputs m mutually exclusive sets $\{Y_i\}_{i\in[m]}$..
Output: $\mathcal{S}$ outputs a vector $\mathbf{u} = [u_i]_{i\in[m]}$. $\mathcal{R}$ outputs a vector $\mathbf{v} = [v_i]_{i\in[m]}$.

1. $\mathcal{S}$ and $\mathcal{R}$ invoke the pMCRG functionality $\mathcal{F}_{\text{pMCRG}}$.
   (a) $\mathcal{S}$ inputs the set $X = \{x_i\}_{i\in[m]}$ and a permutation π over $[m]$, and $\mathcal{R}$ inputs $\{Y_i\}_{i\in[m]}$.
   (b) $\mathcal{S}$ and $\mathcal{R}$ outputs two vectors $\mathbf{s} = [s_i]_{i\in[m]}$ and $\mathbf{t} = [t_i]_{i\in[m]}$, where for all $i \in [m]$, if $x_{\pi^{-1}(i)} \in Y_{\pi^{-1}(i)}$, $s_i = t_i$, otherwise $s_i \neq t_i$.
2. $\mathcal{S}$ and $\mathcal{R}$ invoke the nECRG functionality $\mathcal{F}_{\text{nECRG}}$.
   (a) $\mathcal{S}$ inputs a vector $\mathbf{s} = [s_i]_{i\in[m]}$. $\mathcal{R}$ inputs a vector $\mathbf{t} = [t_i]_{i\in[m]}$.
   (b) $\mathcal{S}$ outputs a vector $\mathbf{u} = [u_i]_{i\in[m]}$. $\mathcal{R}$ outputs a vector $\mathbf{v} = [v_i]_{i\in[m]}$, where for $i \in [m]$, if $s_i = t_i$, we have $u_i \neq v_i$, otherwise $u_i = v_i$.

Figure 18: pnMCRG from pMCRG and nECRG

rupted $\mathcal{S}$ and the corrupted $\mathcal{R}$, such that any PPT environment cannot distinguish the execution in the ideal world from that in the real world.

__Corrupted $\mathcal{S}$.__ Sim simulates a real execution in which the sender $\mathcal{S}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input X and a permutation π of $\mathcal{S}$ directly, and externally send them to $\mathcal{F}_{\text{pnMCRG}}$ and then receives $\langle \mathsf{Request}, \mathcal{S} \rangle$. Once receiving X and π, the input of Π_{pMCRG}, from $\mathcal{A}$, Sim randomly selects $\mathbf{s} = [s_i]_{i\in[m]}$ to simulate the execution of Π_{pMCRG}. Once receiving $\mathbf{s} = [s_i]_{i\in[m]}$, the input of Π_{nECRG}, from $\mathcal{A}$, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{pnMCRG}}$, and obtains $\mathbf{u} = [u_i]_{i\in[m]}$. Finally, Sim simulates the execution of Π_{nECRG} with $\mathbf{u} = [u_i]_{i\in[m]}$ as output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{S}$ by the following hybrids:

Hyb_0: $\mathcal{S}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{pMCRG} is replaced by $\mathbf{s} = [s_i]_{i\in[m]}$ chosen by Sim, and Sim runs the Π_{pMCRG} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{pMCRG} 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 of Π_{nECRG} is replaced by $\mathbf{u} = [u_i]_{i\in[m]}$ output by $\mathcal{F}_{\text{pnMCRG}}$, and Sim runs the Π_{nECRG} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{nECRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

__Corrupted $\mathcal{R}$.__ Sim simulates a real execution in which the receiver $\mathcal{R}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input (m mutually exclusive sets $\{Y_i\}_{i\in[m]}$) of $\mathcal{R}$ directly, and externally send the set $\{Y_i\}_{i\in[m]}$ to $\mathcal{F}_{\text{pnMCRG}}$ and then receives $\langle \mathsf{Request}, \mathcal{R} \rangle$. Sim randomly selects $\mathbf{t} = [t_i]_{i\in[m]}$ to simulate the execution of Π_{pMCRG}. Once receiving $\mathbf{t} = [t_i]_{i\in[m]}$, the input of Π_{nECRG}, from $\mathcal{A}$, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{pnMCRG}}$, and obtains $\mathbf{v} = [v_i]_{i\in[m]}$. Finally, Sim simulates the execution of Π_{nECRG} with $\mathbf{v} = [v_i]_{i\in[m]}$ as output.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:

Hyb_0: $\mathcal{R}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{pMCRG} is replaced by $\mathbf{t} = [t_i]_{i\in[m]}$ chosen by Sim, and Sim runs the Π_{pMCRG} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{pMCRG} 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 of Π_{nECRG} is replaced by $\mathbf{v} = [v_i]_{i\in[m]}$ output by $\mathcal{F}_{\text{pnMCRG}}$, and Sim runs the Π_{nECRG} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{nECRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

7 A New Framework of Enhanced PSU

In this section, we present a new framework of enhanced PSU based on pnMCRG, as described in Figure 19. In this construction, each bin of the simple hash table is treated as a separate set. Since each item y_j of equal length is linked to different hash function indices $(y_j||1, y_j||2, y_j||3)$, the m_c sets satisfy mutual exclusivity.

Theorem 6. *The protocol Π_{ePSU} shown in Figure 19 UC-realizes the functionality $\mathcal{F}_{\text{ePSU}}$ (as in Figure 6) in the $\mathcal{F}_{\text{pnMCRG}}$-hybrid model, against static, semi-honest adversaries.*

Proof. We will show that for any adversary $\mathcal{A}$, we can construct a simulator Sim that can simulate the view of the corrupted $\mathcal{S}$ and the corrupted $\mathcal{R}$, such that any PPT environment cannot distinguish the execution in the ideal world from that in the real world.

[8]Here, $\mathcal{S}$ needs to remove the hash function index.

Input: The sender inputs a set $X = \{x_i\}_{i \in [m]}$. The receiver inputs a set $Y = \{y_j\}_{j \in [n]}$.
Output: The receiver outputs the union $X \cup Y$. The sender outputs Finished.

1. $\mathcal{S}$ inserts set X into the cuckoo hash table and fills empty bins with the dummy item d, where the cuckoo hash table $X_c = \{X_c[i]\}_{i \in [m_c]}$ consists of m_c bins and each bin $X_c[i]$ has only one item, where for each x_i there is some $\gamma \in \{1,2,3\}$ such that $X_c[h_\gamma(x_i)] = x_i || \gamma$.
2. $\mathcal{R}$ uses the same hash functions to insert Y into the simple hash table, where all item y_j are concatenated with hash function indices ($y_j||1, y_j||2, y_j||3$) and are inserted to the bins ($Y_{h_1(y_j)}, Y_{h_2(y_j)}, Y_{h_3(y_j)}$), respectively, the table consists of m_c bins $\{Y_1, Y_2, \cdots, Y_{m_c}\}$ and each bin consists of $B_i = |Y_i|$ items.
3. $\mathcal{S}$ and $\mathcal{R}$ invoke the pnMCRG functionality $\mathcal{F}_{\text{pnMCRG}}$.
   (a) $\mathcal{S}$ chooses a random permutation π over $[m_c]$ and input $X_c = \{X_c[i]\}_{i \in [m_c]}$ and $\mathcal{R}$ input $\{Y_1, Y_2, \cdots, Y_{m_c}\}$.
   (b) $\mathcal{S}$ obtains $\mathbf{u} = [u_i]_{i \in [m_c]}$. $\mathcal{R}$ obtains $\mathbf{v} = [v_i]_{i \in [m_c]}$.
4. $\mathcal{S}$ computes $c_i = u_i \oplus (X_c[\pi^{-1}(i)] || h(X_c[\pi^{-1}(i)]))$[8] $, i \in [m_c]$ and sends them to $\mathcal{R}$, where h is a pre-agreed hash function used to distinguish between real items and random values.
5. $\mathcal{R}$ computes $m_i || m'_i = v_i \oplus c_i$, $i \in [m_c]$. For $i \in [m_c]$, if $m'_i = h(m_i)$ and $m_i \neq d$, let $Z = Y \cup \{m_i\}$. $\mathcal{R}$ outputs the union Z. $\mathcal{S}$ outputs Finished.

Figure 19: A framework of enhanced PSU from pnMCRG

Corrupted $\mathcal{S}$. Sim simulates a real execution in which the sender $\mathcal{S}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input X of $\mathcal{S}$ directly, and externally send the set X to $\mathcal{F}_{\text{ePSU}}$ and then receives $\langle \text{Request}, \mathcal{S} \rangle$. Once receiving X_c and π, the input of Π_{pnMCRG}, from $\mathcal{A}$, Sim randomly selects $\mathbf{u} = [u_i]_{i \in [m_c]}$ to simulate the execution of Π_{pnMCRG}. After receiving $c_1, \cdots, c_{m_c}$, Sim sends $\langle \text{Response}, \text{OK} \rangle$ to $\mathcal{F}_{\text{ePSU}}$.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{S}$ by the following hybrids:

Hyb_0: $\mathcal{S}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{pnMCRG} is replaced by $\mathbf{u} = [u_i]_{i \in [m]}$ chosen by Sim, and Sim runs the Π_{pnMCRG} simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{pnMCRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Corrupted $\mathcal{R}$. Sim simulates a real execution in which the receiver $\mathcal{R}$ is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input Y of $\mathcal{R}$ directly, and externally send the set Y to $\mathcal{F}_{\text{ePSU}}$ and then receives $\langle \text{Request}, \mathcal{R} \rangle$. Once receiving m_c mutually exclusive sets $\{Y_1, Y_2, \cdots, Y_{m_c}\}$, the input of Π_{pnMCRG}, from $\mathcal{A}$, Sim randomly selects $\mathbf{v} = [v_i]_{i \in [m_c]}$ to simulate the execution of Π_{pnMCRG}. Sim sends $\langle \text{Response}, \text{OK} \rangle$ to $\mathcal{F}_{\text{ePSU}}$. After receiving $Z = X \cup Y$ from $\mathcal{F}_{\text{ePSU}}$, Sim calculates $X' = Z \backslash Y$ and randomly selects a subset T from $\{v_1, \cdots, v_{m_c}\}$, where $|T| = |X'|$. For each items in T, Sim sets $c_i = T[i] \oplus (X'[i] || h(X'[i]))$. Finally, Sim chooses random values for $i \in \{|T|+1, |T|+2, \cdots, m_c\}$ and sends all shuffled ciphertexts to $\mathcal{A}$.

We argue that the outputs of Sim are indistinguishable from the real view of $\mathcal{R}$ by the following hybrids:

Hyb_0: $\mathcal{R}$'s view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{pnMCRG} is replaced by $\mathbf{u} = [u_i]_{i \in [m]}$ chosen by Sim, and Sim runs the Π_{pnMCRG} simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{pnMCRG} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

8 Implementation and Performance

Here, we evaluate our enhanced balanced/unbalanced PSU (ePSU/eUPSU), and then compare with the state-of-the-art protocols, such as enhanced PSU [29] and balanced/unbalanced PSU suffering from during-execution leakage [45, 47, 48], in terms of communication and runtime in different network environments.

8.1 Experimental Setup

We implement our protocols in C++ and the source code is available at https://doi.org/10.5281/zenodo.14725816. The experiments are conducted on a single Intel Core i7-13700 CPU @ 5.20GHz with 32 threads and 64GB of RAM, running Ubuntu. We evaluate our protocols in two network settings: LAN (10Gbps bandwidth, 0.2ms RTT) and WAN (100Mbps, 10Mbps, and 1Mbps bandwidth, 80ms RTT), emulated using the Linux tc command. We leverage the constructions in [21, 31, 42, 43] to implement bOPRF, OKVS, ssPEQT and FHE (the building block of balanced/unbalanced pnMCRG) and use the code from the Vole-PSI library [6], the SEAL library [2], and the APSI/APSU library [1, 5]. As for ROT (the building block of nECRG) and DDH-based pECRG, we follow the libOTe library [4] and the OpenSSL library [3].

Size $\lvert X\rvert = \lvert Y\rvert$	Protocols	Comm. (MB)	Total running time (s) with single thread			
			10Gbps	100Mbps	10Mbps	1Mbps
2^{10}	PSU [48]	0.188	2.047	3.087	3.249	4.655
	ePSU [29]	2.882	1.250	9.261	11.116	33.806
	Our ePSU	1.31	0.172	4.046	5.093	15.127
2^{12}	PSU [48]	0.700	2.240	3.116	3.482	9.246
	ePSU [29]	8.600	1.260	11.491	17.241	82.397
	Our ePSU	2.028	0.369	4.824	6.142	21.683
2^{14}	PSU [48]	2.726	3.410	4.387	5.963	27.447
	ePSU [29]	32.921	1.516	15.602	38.770	289.266
	Our ePSU	5.202	1.276	6.248	9.729	49.795
2^{16}	PSU [48]	11.143	8.124	9.579	16.096	99.819
	ePSU [29]	137.419	2.599	28.190	129.929	1176.530
	Our ePSU	17.955	4.899	11.058	23.445	160.392
2^{18}	PSU [48]	44.712	26.758	30.182	56.318	393.639
	ePSU [29]	577.731	7.235	80.436	514.592	4942.710
	Our ePSU	69.03	19.094	31.567	80.679	604.347
2^{20}	PSU [48]	179.061	98.406	110.124	214.808	1569.242
	ePSU [29]	2430.470	35.344	310.406	2143.990	20717.900
	Our ePSU	277.402	78.144	108.729	316.009	2413.015
2^{22}	PSU [48]	716.875	413.917	446.72	850.761	6294.396
	ePSU [29]	—	—	—	—	—
	Our ePSU	1121.588	319.182	427.657	1244.048	9759.93

Table 2: Comparisons of communication and runtime between PSU [48], enhanced PSU [29] and our enhanced PSU in the balanced setting. — indicates an execution error. 10Gbps bandwidth, 0.2ms RTT; 100Mbps, 10Mbps and 1Mbps bandwidth, 80ms RTT. The best results are marked in magenta and the second best results are marked in blue.

Size $(\lvert X\rvert, \lvert Y\rvert)$	Protocols	Comm. (MB)	Total running time (s) with single thread			
			10Gbps	100Mbps	10Mbps	1Mbps
$(2^{10}, 2^{12})$	UPSU [45]	1.614	0.967	2.825	2.825	14.132
	UPSU [47]	20.270	10.294	14.309	28.676	181.680
	Our ePSU	1.578	0.297	4.044	5.395	17.199
	Our eUPSU	2.237	1.430	6.527	7.689	25.406
$(2^{10}, 2^{14})$	UPSU [45]	2.681	1.22	3.48	3.58	18.136
	UPSU [47]	20.271	26.196	29.903	45.061	189.584
	Our ePSU	2.607	0.313	4.113	6.312	25.856
	Our eUPSU	2.237	1.545	6.536	7.948	26.704
$(2^{10}, 2^{16})$	UPSU [45]	2.681	1.475	3.481	3.634	18.957
	UPSU [47]	21.286	35.141	40.486	54.984	215.567
	Our ePSU	6.724	0.398	4.569	9.849	60.592
	Our eUPSU	2.237	1.996	7.129	8.237	27.282
$(2^{10}, 2^{18})$	UPSU [45]	2.367	3.723	5.833	6.137	19.384
	UPSU [47]	21.321	103.552	107.441	123.590	283.007
	Our ePSU	23.052	0.727	6.693	23.694	198.757
	Our eUPSU	2.237	3.649	8.523	10.399	29.425
$(2^{10}, 2^{20})$	UPSU [45]	2.767	18.465	20.995	21.297	32.253
	UPSU [47]	23.884	197.250	202.368	219.585	397.725
	Our ePSU	88.468	2.515	16.140	79.531	750.332
	Our eUPSU	2.322	15.957	20.939	23.583	42.734

Table 3: Comparisons of communication and runtime between our two enhanced PSU (ePSU/eUPSU) and ubalanced PSU with during-execution leakage [45, 47] in the unbalanced setting. 10Gbps bandwidth, 0.2ms RTT; 100Mbps, 10Mbps and 1Mbps bandwidth, 80ms RTT. The best results are marked in magenta and the second best results are marked in blue.

Our code supports multithreading parallelism following the Vole-PSI library and the OpenMP library [7]. Following most PSU [29, 30, 45, 47, 48], we set the computational security parameter $\kappa = 128$, the statistical security parameter $\lambda = 40$ and use $\gamma = 3$ hash functions to insert sets X and Y into the cuckoo hash table and simple hash table, respectively. The item length is 64 bits following [20, 34].

8.2 Comparisons in the Balanced Setting

Here, we compare our ePSU with the state-of-the-art enhanced PSU [29] in the balanced setting, and the results are reported in Table 2 and Figure 20. Since their implementations [29] do not support multi-threading, we only provide a single-threaded comparison in Table 2. We present the runtime of our multi-threaded implementation in Table 4.

Communication comparison. As shown in Figure 20, our ePSU achieves lower communication overhead than [29] across all set sizes. As indicated in Table 2, our protocol reduces communication costs by a factor of 2.2 to 8.8 for both set sizes ranging from 2^{10} to 2^{20}. Since the complexity of PSU [29] is superlinear, while the complexity of our ePSU is linear, the larger the set size, the more significant the advantage of our ePSU becomes.

Runtime comparison. As shown in Figure 20, our protocol is more efficient than PSU [29] in terms of runtime in low-bandwidth environments for large sets, while for small sets, it consistently outperforms PSU [29] across the board. As indicated in Table 2, for both set sizes 2^{10}, the running time of our protocol surpasses PSU [29] by 2.2 - 7.6$\times$. For both set sizes 2^{20}, the running time of our protocol surpasses PSU [29] by 2.6 - 8.6$\times$ under bandwidths ranging from 1Mbps to 100Mbps.

8.3 Comparisons in the Unbalanced Setting

Since the implementation of PSU [29] does not support running in unbalanced settings, we compare our enhanced unbalanced PSU (eUPSU) with our enhanced balanced PSU (ePSU) in the unbalanced setting, and the results are reported in Figure 21 and Table 3.

Communication comparison. As shown in Figure 21, our eUPSU achieves lower communication overhead than ePSU in the unbalanced setting, and the larger the difference between two set sizes, the better our eUPSU performs. As indicated in Table 3, our protocol reduces communication costs by 1.2 - 38.1$\times$ under the large set size ranging from 2^{14} to 2^{20}. Especially, for set sizes $(\lvert X\rvert = 2^{10}, \lvert Y\rvert = 2^{20})$, the communication of our eUPSU requires 2.322 MB, which is about 38.1$\times$ lower than our ePSU requiring 88.468 MB.

Runtime comparison. As shown in Figure 21, our protocol is more efficient than ePSU in terms of runtime in low-bandwidth environments, the lower the bandwidth, the better

Size $\lvert X\rvert = \lvert Y\rvert$	Total running time (s)															
	10Gbps				100Mbps				10Mbps				1Mbps			
	$T=1$	$T=2$	$T=4$	$T=8$	$T=1$	$T=2$	$T=4$	$T=8$	$T=1$	$T=2$	$T=4$	$T=8$	$T=1$	$T=2$	$T=4$	$T=8$
2^{10}	0.172	0.128	0.119	0.102	4.046	4.043	4.041	3.875	5.093	5.052	5.044	5.012	15.127	15.085	15.077	15.013
2^{12}	0.369	0.258	0.228	0.208	4.824	4.708	4.659	4.582	6.142	5.978	5.955	5.923	21.683	21.555	21.495	21.283
2^{14}	1.276	0.783	0.605	0.565	6.248	5.533	5.469	5.363	9.729	9.122	8.695	8.570	49.795	49.362	49.056	48.997
2^{16}	4.899	3.024	2.172	2.011	11.058	9.417	8.371	7.935	23.445	21.458	20.624	20.173	160.392	158.653	157.592	157.102
2^{18}	19.094	11.951	8.513	7.720	31.567	23.281	19.847	18.905	80.679	72.708	69.260	66.905	604.347	597.356	592.981	590.666
2^{20}	78.144	48.862	36.416	33.170	108.729	79.396	67.317	62.653	316.009	286.499	269.169	263.700	2413.015	2387.234	2371.529	2363.548
2^{22}	319.182	204.297	157.931	149.532	427.657	311.539	265.428	240.792	1244.048	1136.868	1080.177	1060.55	9759.93	9671.29	9596.11	9560.027

Table 4: Runtime of our enhanced balanced PSU. Threads $T \in \{1,2,4,8\}$. 10Gbps bandwidth, 0.2ms RTT; 100Mbps, 10Mbps and 1Mbps bandwidth, 80ms RTT.

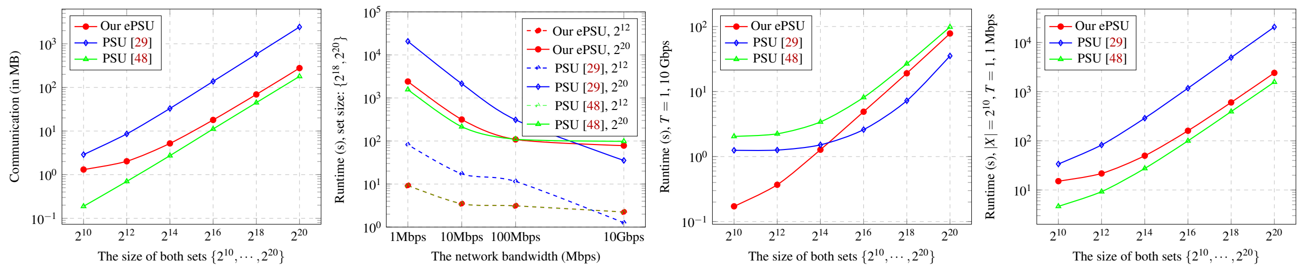

Figure 20: Comparisons of communication and runtime between our enhanced PSU and PSU [29] in the balanced setting. Both x and y-axis are in log scale. The first figure shows the communication cost increases as the set size increases. The second figure shows the runtime decreases as the bandwidth increases. The last two figures show the runtime increases as the set size increases.

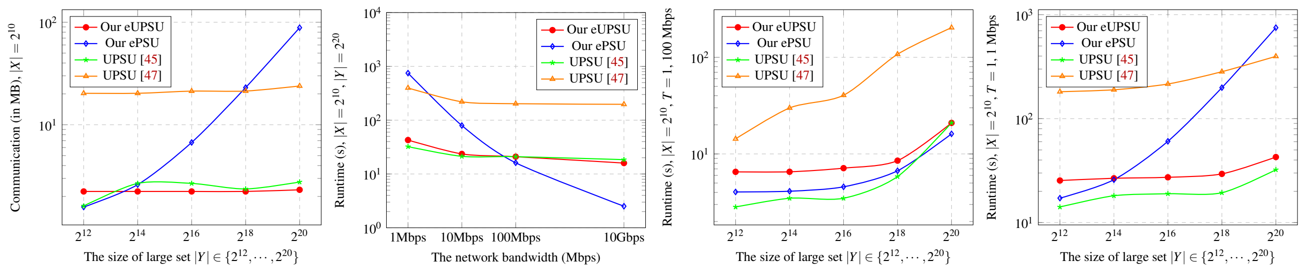

Figure 21: Comparisons of communication and runtime between our two enhanced PSU (ePSU/eUPSU) in the unbalanced setting. Both x and y-axis are in log scale. The first figure shows the communication cost increases as the large set size increases. The second figure shows the runtime decreases as the bandwidth increases. The last two figures show the runtime increases as the large set size increases.

our eUPSU performs. As indicated in Table 3, for set sizes ($\lvert X\rvert = 2^{10}$, $\lvert Y\rvert = 2^{20}$) with $T = 1$ thread in 1Mbps bandwidth, our eUPSU requires 42.734 seconds, while our ePSU requires 750.332 seconds, about 17.6× improvement.

8.4 Comparison with PSU Suffering from During-Execution Leakage

Here, we compare our ePSU/eUPSU with the efficient PSU/UPSU [45, 47, 48] in the balanced/unbalanced setting, which suffers from during-execution leakage.

In the balanced setting, experimental results show that our eUPSU has a slightly higher communication cost compared to PSU [48], but achieves shorter runtime under a 10Gbps bandwidth. As shown in Figure 20 and Table 2, the communication overhead becomes comparable as the set size increases. For set sizes ranging from 2^{12} to 2^{22}, PSU [48] has a 1.7 - 2.8× advantage in communication, whereas our ePSU is 1.3 - 6.1× faster in runtime. Hence, our ePSU addresses the issue of during-execution leakage in PSU [48] with only a minor efficiency trade-off.

In the unbalanced setting, experimental results demonstrate that while our eUPSU has a slightly higher computation cost, it offers an advantage in terms of communication complexity. As shown in Figure 21 and Table 3, our eUPSU and UPSU [45] have only minor differences in communication and computation. Our eUPSU performs slightly better in communication, while UPSU [45] has a slight advantage in computation. For large set sizes ranging from 2^{12} to 2^{20}, our eUPSU reduces communication overhead by a factor of 1.1 - 1.4$\times$ compared to UPSU [45], and by 9.1 - 10.3$\times$ compared to UPSU [47]. Therefore, our eUPSU effectively addresses the issue of during-execution leakage of UPSU [45, 47] with only a minor increase in computational overhead.

8.5 Microbenchmarks of Our Sub-Protocols

We present the hierarchy of all sub-protocols in ePSU/eUPSU, along with their communication and computation complexity, in Table 5. The protocol execution is divided into three steps. The first step is hashing (corresponding to steps 1 and 2 in Figure 19), which does not require communication and is completed locally by both parties. The second step is pnMCRG (corresponding to step 3 in Figure 19). The third step is one-time pad (corresponding to steps 4 and 5 in Figure 19). As demonstrated in Figure 1, our pnMCRG can be further divided into two steps, pMCRG and nECRG, which we represent as steps 2.1 and 2.2. Similarly, our pMCRG can be further divided into three steps, namely bOPRF+OKVS+pECRG in the balanced setting and bOPRF+FHE+pECRG in the unbalanced setting, denoted as steps 2.1.1, 2.1.2, and 2.1.3, respectively.

Step	Sub-Protocols		Communication		Computation	
	ePSU	eUPSU	ePSU	eUPSU	ePSU	eUPSU
1	cuckoo/simple hash		-	-	$O(n)$	$O(n+m)$
2	pnMCRG		$O(n)$	$O(m\log n)$	$O(n)$	$O(n+m\log n)$
2.1	pMCRG		$O(n)$	$O(m\log n)$	$O(n)$	$O(n+m\log n)$
2.1.1	bOPRF	bOPRF	$O(n)$	$O(m)$	$O(n)$	$O(m)$
2.1.2	OKVS	FHE	$O(n)$	$O(m\log n)$	$O(n)$	$O(n+m\log n)$
2.1.3	pECRG	pECRG	$O(n)$	$O(m)$	$O(n)$	$O(m)$
2.2	nECRG		$O(n)$	$O(m)$	$O(n)$	$O(m)$
3	one-time pad		$O(n)$	$O(m)$	$O(n)$	$O(m)$
	Total		$O(n)$	$O(m\log n)$	$O(n)$	$O(n+m\log n)$

Table 5: Communication and computation complexity of each sub-protocol in our ePSU/eUPSU.

Size	Sub-protocols			Comm.(MB)			Runtime (s)		
$(2^{16},2^{16})$	pnMCRG	pMCRG	bOPRF+OKVS	16.655	11.571	6.370	4.790	4.461	0.082
			pECRG			5.200			4.388
		nECRG			5.084			1.135	
	Total ePSU			17.955			4.899		
$(2^{20},2^{20})$	pnMCRG	pMCRG	bOPRF+OKVS	256.355	172.382	88.120	78.032	59.801	1.103
			pECRG			84.267			58.715
		nECRG			83.954			18.536	
	Total ePSU			277.402			78.144		
$(2^{10},2^{16})$	pnMCRG	pMCRG	bOPRF+FHE	2.212	1.952	1.850	1.319	1.263	1.181
			pECRG			0.102			0.082
		nECRG			0.253			0.069	
	Total eUPSU			2.237			1.996		
$(2^{10},2^{20})$	pnMCRG	pMCRG	bOPRF+FHE	2.296	2.016	1.910	15.656	15.568	15.472
			pECRG			0.106			0.093
		nECRG			0.28			0.095	
	Total eUPSU			2.322			15.957		

Table 6: Communication and runtime of our sub-protocols. 10Gbps bandwidth, 0.2ms RTT; Thread $T = 1$.

To demonstrate the performance of our sub-protocols, we perform microbenchmarks for our sub-protocols, as shown in Table 6. In the balanced setting, the communication overhead of the three components (bOPRF+OKVS, pECRG, nECRG) is similar, while the running time is primarily dominated by pECRG, which serves as the performance bottleneck. In the unbalanced setting, FHE costs the majority of both communication and computation, making it the primary performance bottleneck.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. We thank Weiran Liu for his help on conducting comparison experiment of [48]. This work is supported by the National Natural Science Foundation of China (No. 62272269 and No. 62402272), the National Key Research and Development Program of China (No. 2021YFA1000600), Taishan Scholar Program of Shandong Province, Shandong Laboratories Project of Bureau of Science & Technology of Shandong Province (SYS202201), and Key Project of Quan Cheng Laboratory (QCLZD202302).

Ethics Considerations and Compliance with the Open Science Policy

Ethics Considerations. We strictly follow the ethical guidelines set forth by USENIX Security. Our research does not involve any ethical issues. All experiments conducted in this paper are based on publicly available datasets and do not involve personal or sensitive data, ensuring full compliance with privacy and data protection standards.

Compliance with Open Science Policy. We fully support the principles of the Open Science Policy. We have incorporated our research artifacts into an open-source GitHub repository and are ready for commitment. The artifacts are prepared for submission, and we will openly share them in the final version of the paper, ensuring transparency and reproducibility. By adhering to the open science principles, we support the broad dissemination of scientific knowledge and facilitate further research in our field. Our approach aligns with the goal of enhancing the reproducibility and reliability of scientific findings, contributing to a more open and collaborative research environment.

References

[1] https://github.com/microsoft/APSI.

[2] https://github.com/microsoft/SEAL.

[3] https://github.com/openssl/openssl.

[4] https://github.com/osu-crypto/libOTe.

[5] https://github.com/real-world-cryprography/APSU.

[6] https://github.com/Visa-Research/volepsi.

[7] https://www.openmp.org/.

[8] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. Near-optimal oblivious key-value stores for efficient psi, PSU and volume-hiding multi-maps. In *USENIX Security*, 2023.

[9] Zvika Brakerski, Craig Gentry, and Shai Halevi. Packed ciphertexts in lwe-based homomorphic encryption. In *PKC*, 2013.

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

[11] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security*, 2010.

[12] Nishanth Chandran, Divya Gupta, and Akash Shah. Circuit-psi with linear complexity via relaxed batch OPPRF. *Proc. Priv. Enhancing Technol.*, 2022.

[13] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *ACM CCS*, 2018.

[14] Hao Chen, Kim Laine, and Rachel Player. Simple encrypted arithmetic library - SEAL v2.1. In *Financial Cryptography and Data Security*, 2017.

[15] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *ACM CCS*, 2017.

[16] Yu Chen, Min Zhang, Cong Zhang, Minglang Dong, and Weiran Liu. Private set operations from multi-query reverse private membership test. In *PKC*, 2024.

[17] 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 CCS*, 2021.

[18] Geoffroy Couteau. New protocols for secure equality test and comparison. In *ACNS*, 2018.

[19] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In *ACISP*, 2017.

[20] Minglang Dong, Yu Chen, Cong Zhang, and Yujie Bai. Breaking free: Efficient multi-party private set union without non-collusion assumptions. *IACR Cryptol. ePrint Arch.*, 2024.

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

[22] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[23] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In *PKC*, 2021.

[24] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In *CRYPTO*, 2021.

[25] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic evaluation of the AES circuit. In *CRYPTO*, 2012.

[26] 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 *ICML*, 2016.

[27] Xiaojie Guo, Ye Han, Zheli Liu, Ding Wang, Yan Jia, and Jin Li. Birds of a feather flock together: How set bias helps to deanonymize you via revealed intersection sizes. In *USENIX Security*, 2022.

[28] 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*, 2016.

[29] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, and Dawu Gu. Scalable private set union, with stronger security. In *USENIX Security*, 2024.

[30] Yanxue Jia, Shifeng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. In *USENIX Security*, 2022.

[31] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *ACM CCS*, 2016.

[32] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In *ASIACRYPT*, 2019.

[33] Arjen K. Lenstra and Tim Voss. Information security risk assessment, aggregation, and mitigation. In *ACISP*, 2004.

[34] Xiang Liu and Ying Gao. Scalable multi-party private set union from multi-query secret-shared private membership test. In *ASIACRYPT*, 2023.

[35] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms - ESA*, 2001.

[36] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *CRYPTO*, 2019.

[37] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from paxos: Fast, malicious private set intersection. In *EUROCRYPT*, 2020.

[38] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security*, 2015.

[39] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In *EUROCRYPT*, 2019.

[40] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *USENIX Security*, 2014.

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

[42] Srinivasan Raghuraman and Peter Rindal. Blazing fast PSI from improved OKVS and subfield VOLE. In *ACM CCS*, 2022.

[43] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: fast OPRF and circuit-psi from vector-ole. In *EUROCRYPT*, 2021.

[44] Nigel P. Smart and Frederik Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 2014.

[45] Binbin Tu, Yu Chen, Qi Liu, and Cong Zhang. Fast unbalanced private set union from fully homomorphic encryption. In *ACM CCS*, 2023.

[46] Binbin Tu, Xiangling Zhang, Yujie Bai, and Yu Chen. Fast unbalanced private computing on (labeled) set intersection with cardinality. *IACR Cryptol. ePrint Arch.*, 2023.

[47] Cong Zhang, Yu Chen, Weiran Liu, Liqiang Peng, Meng Hao, Anyu Wang, and Xiaoyun Wang. Unbalanced private set union with reduced computation and communication. In *ACM CCS*, 2024.

[48] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. Linear private set union from multi-query reverse private membership test. In *USENIX Security*, 2023.

A Optimization techniques

Following unbalanced PSI/PSU [13, 15, 17, 45], we can take advantage of the same optimization techniques of FHE [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 (FHE) that enables Single Instruction, Multiple Data (SIMD) operations on ciphertexts [9, 14, 25, 26, 44]. This technique allows the receiver to process γ items from the sender simultaneously, leading to a γ-fold improvement in both computation and communication efficiency. More precisely, the sender groups its items into vectors of length γ, encrypts them, and sends m/γ 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 $r_i + \Pi_{y \in Y}(c_i - y)$, and sends it back to the sender.

Windowing. We utilize the windowing technique [13, 15, 17, 45] to reduce the depth of the circuit. In our eUPSU, the receiver needs to evaluate the encrypted data. Given an encryption $c \leftarrow \mathsf{FHE.Enc}(x)$, the receiver samples a random r in $\mathbb{Z}_t \backslash \{0\}$ and homomorphically computes $r + \Pi_{y_i \in Y}(c - y_i)$. 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 $r + \Pi_{y_i \in Y}(x - y_i) = r + a_0 + a_1 x + \cdots + a_{n-1} x^{n-1} + x^n$. If the sender provides encryptions of additional powers of x, the receiver can use these to compute the same operation with a lower circuit depth. More precisely, for a window size of l bits, the sender computes and sends $c_{ij} = \mathsf{FHE.Enc}(x^{i \cdot 2^{lj}})$ 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 sender transmits encryptions of $x, x^2, \cdots, x^{2^{\lfloor \log_2 n \rfloor}}$. This approach significantly reduces the circuit depth.

Partitioning. Another approach to reducing circuit depth is to partition the receiver's set into α subsets [13, 15, 17, 45]. In our eUPSU, the receiver computes encryptions of all powers $x, \cdots, x^n$ for each sender's item x. With partitioning, the receiver only needs to compute encryptions of $x, \cdots, x^{n/\alpha}$, which can be reused across the α partitions.

Modulus switching. We employ modulus switching to effectively reduce the size of the response ciphertexts as [10,13,15]. Modulus switching is a well-known operation in FHE. It is a public operation that transforms a ciphertext encrypted with the parameter q into a ciphertext that encrypts the same plaintext but using a smaller parameter q', where $q' < q$. Note that the security of the protocol is trivially preserved as long as the smaller modulus q' is determined at setup.