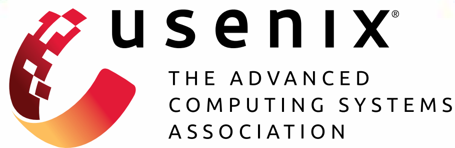


Scalable Private Set Union, with Stronger Security

Yanxue Jia, *Purdue University;* Shi-Feng Sun, *Shanghai Jiao Tong University;* Hong-Sheng Zhou, *Virginia Commonwealth University;* Dawu Gu, *Shanghai Jiao Tong University*

https://www.usenix.org/conference/usenixsecurity24/presentation/jia-yanxue

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Scalable Private Set Union, with Stronger Security

Yanxue Jia[1*], Shi-Feng Sun[2†], Hong-Sheng Zhou[3], and Dawu Gu[2]

[1]*Purdue University*
[2]*Shanghai Jiao Tong University*
[3]*Virginia Commonwealth University*

Abstract

Private Set Union (PSU) protocol allows parties, each holding an input set, to jointly compute the union of the sets without revealing anything else. In the literature, scalable PSU protocols follow the "split-execute-assemble" paradigm (Kolesnikov et al., ASIACRYPT 2019); in addition, those fast protocols often use Oblivious Transfer as building blocks. Kolesnikov et al. (ASIACRYPT 2019) and Jia et al. (USENIX Security 2022), pointed out that certain security issues can be introduced in the "split-execute-assemble" paradigm. In this work, surprisingly, we observe that the typical way of invoking Oblivious Transfer also causes unnecessary leakage, and only the PSU protocols based on additively homomorphic encryption (AHE) can avoid the leakage. However, the AHE-based PSU protocols are far from being practical.

To bridge the gap, we also design a new PSU protocol that can avoid the unnecessary leakage. Unlike the AHE-based PSU protocols, our new construction only relies on symmetric-key operations other than base OTs, thereby being much more scalable. The experimental results demonstrate that our protocol can obtain at least 873.74× speedup over the best-performing AHE-based scheme. Moreover, our performance is comparable to that of the state-of-the-art PSU protocol (Chen et al., USENIX Security 2023), which also suffers from the unnecessary leakage.

1 Introduction

In a Private Set Union (PSU) protocol, two players, a sender and a receiver, holding input sets X and Y, respectively, can jointly compute the union $X \cup Y$ as output. To ensure the joint computation is *private*, any additional information except the union $X \cup Y$, is not allowed to be learned by the players. Especially, information about the items in the intersection set $X \cap Y$ should not be learned by the players. Often we consider a simplified version of PSU: Instead of having both players to obtain the same output $X \cup Y$, in the simplified version of PSU, only the receiver obtains the output $X \cup Y$.

Symmetric Key-based PSU protocols. Kolesnikov et al. [21] is the first to only leverage symmetric key techniques to design a PSU protocol, such that their protocol is truly practical and scalable. Multiple followup results then are developed [1,8,13,19,29,31]. In this work, we observe that all of these symmetric key-based protocols suffer from unnecessary leakage.

Kolesnikov et al.'s "split-execute-assemble" based PSU. Kolesnikov et al. [21] introduce for the first time the "split-execute-assemble" paradigm into the design of scalable PSU protocols: First, the pair of input sets X and Y are **split** into multiple much smaller pairs of subsets, i.e., $\{(X_1,Y_1),(X_2,Y_2),\cdots,(X_\beta,Y_\beta)\}$, where $\beta \in \mathbb{N}$; Here, $|X| = N_1$ and $|Y| = N_2$; for all $i \in [\beta]$, $|X_i| \ll N_1$ and $|Y_i| \ll N_2$. Then, the two parties **execute** β number of PSU protocol instances, as subroutine, and each instance is on pair of subsets (X_i,Y_i). The receiver obtains $Z_i = X_i \cup Y_i$. Finally, the receiver **assembles** the outputs of all subroutine protocol instances, and obtains the output $Z = Z_1 \cup Z_2 \cup \cdots \cup Z_\beta$.

Security concerns in the "split-execute-assemble" based PSU. Applying the above paradigm to the PSU design is a natural and interesting idea. However, as discussed by Kolesnikov et al. [21], in the "split-execute-assemble" paradigm, the receiver can learn if a subset Y_i includes items that are in the intersection $X \cap Y$, which is not allowed in PSU. In order to eliminate the information leakage, Kolesnikov et al. developed a careful *padding strategy* in [21]. Unfortunately, Jia et al. [19] pointed out that this padding strategy is *insufficient* to eliminate the leakage: Roughly, whenever the output $Z_i = X_i \cup Y_i$ is equal to Y_i for the i-th PSU sub-protocol instance, the receiver will be aware that the i-th subset Y_i includes the items in $X \cap Y$ with an overwhelming probability (see Section 3.1 and Appendix A for more details).

In our work, we observe that the leakage pointed out by Jia et al. [19] can actually be deduced from the output $X \cup Y$. In other words, as long as the receiver obtains the output $X \cup Y$, he can learn the leakage. At first glance, the protocol

*Part of the work was done at Shanghai Jiao Tong University.
†Corresponding author.

by Kolesnikov et al. [21] seems to be secure enough. However, we find that the receiver in Kolesnikov et al. [21] can learn the leakage *during the execution*, rather than *after receiving the output*, and we call this leakage as "during-execution leakage". Unlike obtaining leakage only after protocol execution is completed, during-execution leakage may lead the receiver to terminate the protocol upon learning sufficient leakage[1].

OT-based PSU, and its security concerns. The subsequent works [1, 8, 13, 19, 29, 31] do not leverage the "split-execute-assemble" paradigm, but we find that these solutions still suffer from the during-execution leakage. More specifically, for each item in the input set X, the receiver first learns whether it is a member of the set Y, and then obtains the output $Z = X \cup Y$ by invoking the underlying Oblivious Transfer (OT) instances with the membership information. In other words, the OT-based PSU protocols leaks the membership information *before* the execution of the protocol is completed.

AHE-based PSU. The during-execution leakage is not necessary for a PSU protocol, as we observe that the AHE-based protocols can avoid it. Using additively homomorphic encryption (AHE), Frikken et al. [12] construct the first efficient PSU protocols. Later Davidson et al. [10] improve the performance along this line. More concretely, the receiver generates a representation $P(\cdot)$ of his set Y such that if $x \in Y$, then $P(x) = 0$, and sends $\mathsf{Enc}(P(\cdot))$ to the sender. Then, for each $x \in X$, the sender calculates a ciphertext based on $\mathsf{Enc}(P(\cdot))$. Finally, the receiver can obtain the items in $X \setminus Y$ by decrypting the ciphertexts *without* needing extra information in advance. However, the AHE-based PSU protocols are still far from being practical and scalable.

Main question. Based on the discussions so far, we can see that existing provably secure PSU protocols:

(1) either avoid the during-execution leakage but are not scalable, as in AHE-based PSU protocols;
(2) or are scalable but suffer from the during-execution leakage, as in symmetric key-based PSU protocols.

It will be desirable to achieve the "best of the two worlds". Therefore, we have the following research question: *Is it possible to design a provably secure* <u>*scalable*</u> *PSU that does* <u>*NOT*</u> *suffer from the leakage?*

1.1 Our results

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

Revisiting the existing PSU protocols. In Section 3, we investigate existing PSU protocols, and find that scalable symmetric key-based PSU protocols suffer from the during-execution leakage, but non-scalable AHE-based PSU protocols do not. To provide a formal analysis, we define a new enhanced ideal functionality for PSU, denoted as $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$ in Section 4, that does not allow the leakage during the execution. The formal analysis is provided in Section 6.

[1] In practice, the receiver may be required to pay a fee to the sender after completing the execution. If the execution is terminated, the receiver may not be obligated to pay the fee.

A new PSU protocol. In Section 5, we provide the first PSU protocol which achieves both scalability and the enhanced PSU functionality $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$ *simultaneously*.

<u>*Our design:*</u> The main difference between the OT-based and the AHE-based PSU lies in the method the receiver uses to obtain items in $X \setminus Y$. In the AHE-based PSU, the receiver obtains the items in $X \setminus Y$ via decryption, without knowing any membership information in advance. Whereas, the receiver in the OT-based PSU needs to use the membership information to obtain the items in $X \setminus Y$ through OTs. To achieve the enhanced PSU functionality $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$ as in the AHE-based PSU, our core idea is to "mimic" the decryption process in the AHE-based PSU: For each item $x \in X$, the sender randomly chooses a secret key r and sends the ciphertext $c = x \oplus r$ to the receiver. Then, if $x \notin Y$, the receiver obtains the identical secret key r; otherwise, the receiver obtains a distinct secret key $r' \neq r$. This way enables the receiver to obtain x by calculating $c \oplus r'$ only when $x \notin Y$.

To this end, we employ the following two steps. First, we transform the problem of "determining whether an item x belongs to a set Y" into the problem of "evaluating the equality of two strings, using Oblivious Programmable PRF (OPPRF)." More concretely, if $x \in Y$, the PRF value t obtained by the sender will be equal to the receiver's PRF value t'. Then, we propose a novel building block, called "Equality-Conditional Random Generation (ECRG)", where the sender inputs (t, r) and the receiver inputs t', then the receiver obtains $r' = r$ if $t' \neq t$, or another random string $r' \neq r$ otherwise. Obviously, through ECRG, the receiver can obtain x only when $x \notin Y$.

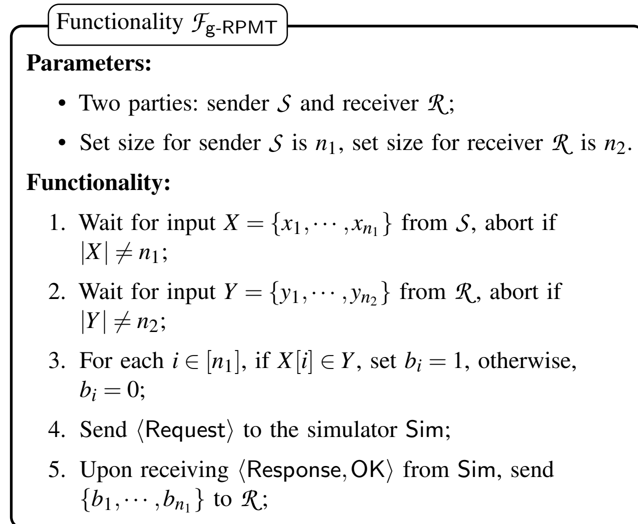
A straightforward way to support the general case, where the sender holds multiple items, is to repeat the above process while using a batched ECRG (bECRG). However, this will incur a quadratic cost. To reduce the cost, we use Cuckoo hashing, and the leakage incurred by Cuckoo hashing can be avoided by leveraging Permute+Share. As in the OT-based PSU, our new PSU construction only relies on symmetric-key operations other than base OTs, and is thus significantly more scalable than AHE-based PSU.

<u>*Performance comparision:*</u> We implement our protocol in C++. The experimental results in Table 2 (see Section 5.5) show that our protocol can obtain *at least* 873.74× *speedup* over the latest AHE-based scheme [10] in a LAN setting, and our communication cost is 5× less than theirs. Moreover, the performance of our protocol is *comparable* to that of the state-of-the-art PSU protocol [31] for large balanced sets, in a LAN setting; Note that the PSU protocol in [31] cannot achieve $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$ due to the during-execution leakage.

2 Preliminaries

In this section, we briefly recall "generalized Reversed Private Membership Test (g-RPMT)", "Oblivious Transfer (OT)", "Oblivious Programmable PRF (OPPRF)", "Private Equality Test (PET)", "Permute+Share (PS)", simple hashing and Cuckoo hashing.

Generalized Reversed Private Membership Test. Reversed Private Membership Test (RPMT) was first proposed and formalized in [21]. More concretely, the sender P_0 holding an item x and the receiver P_1 holding a set Y. Then, the receiver P_1 can learn a bit b without obtaining any information else about item x; if $x \in Y$, $b = 1$, otherwise, $b = 0$. Meanwhile, the sender P_0 knows nothing about P_1's set Y. Based on the RPMT, a generalized RPMT was proposed in [19] where the sender P_0 inputs a set X, rather than an item x. We give the functionality $\mathcal{F}_{\text{g-RPMT}}$ in Figure 1.


Functionality $\mathcal{F}_{\text{g-RPMT}}$

Parameters:

- Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$;
- Set size for sender $\mathcal{S}$ is n_1, set size for receiver $\mathcal{R}$ is n_2.

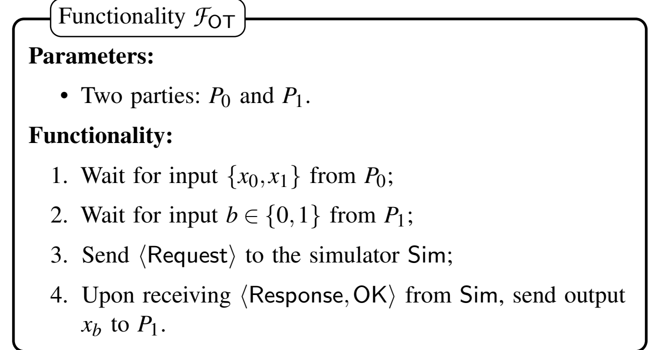
Functionality:

1. Wait for input $X = \{x_1, \cdots, x_{n_1}\}$ from $\mathcal{S}$, abort if $|X| \neq n_1$;
2. Wait for input $Y = \{y_1, \cdots, y_{n_2}\}$ from $\mathcal{R}$, abort if $|Y| \neq n_2$;
3. For each $i \in [n_1]$, if $X[i] \in Y$, set $b_i = 1$, otherwise, $b_i = 0$;
4. Send $\langle \mathsf{Request} \rangle$ to the simulator Sim;
5. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, send $\{b_1, \cdots, b_{n_1}\}$ to $\mathcal{R}$;

Figure 1: Generalized Reversed Private Membership Test Functionality.

1-out-of-2 Oblivious Transfer. 1-out-of-2 oblivious transfer (OT) is a two-party protocol, where party P_0 takes as input two strings $\{x_0, x_1\}$, and the other party P_1 chooses a random bit b and obtains nothing other than x_b while P_0 learns nothing about b. The first OT protocol was proposed by Rabin in [27]. And due to the lower bound in [16], all the OT protocols require expensive public-key operations. To improve the performance, Ishai et al. [17] introduced the concept of OT extension that enables us to carry out many OTs based on a small number of basic OTs. The functionality $\mathcal{F}_{\text{OT}}$ is shown in Figure 2.

Oblivious Programmable PRF. Oblivious Programmable PRF was first proposed and formalized in [20]. Compared to oblivious PRF (OPRF), OPPRF allows the sender P_0 to additionally "program" the PRF values of some inputs, and outputs pseudorandom PRF values everywhere else. More specifically, the sender P_0 inputs $T = \{(x_i, y_i)\}_{i\in[n]}$, which means that the PRF value of x_i is set to be y_i. The receiver P_1 inputs $(q_1, \cdots, q_t)$. After the execution, the sender obtains (k, hint), and the receiver obtains $\{F(k, \mathsf{hint}, q_i)\}_{i\in[t]}$ and hint. Note that if $q_j = x_i$, $F(k, \mathsf{hint}, q_j) = y_i$, otherwise, $F(k, \mathsf{hint}, q_j)$ is pseudorandom. The functionality $\mathcal{F}_{\text{OPPRF}}$ is given in Figure 3.


Functionality $\mathcal{F}_{\text{OT}}$

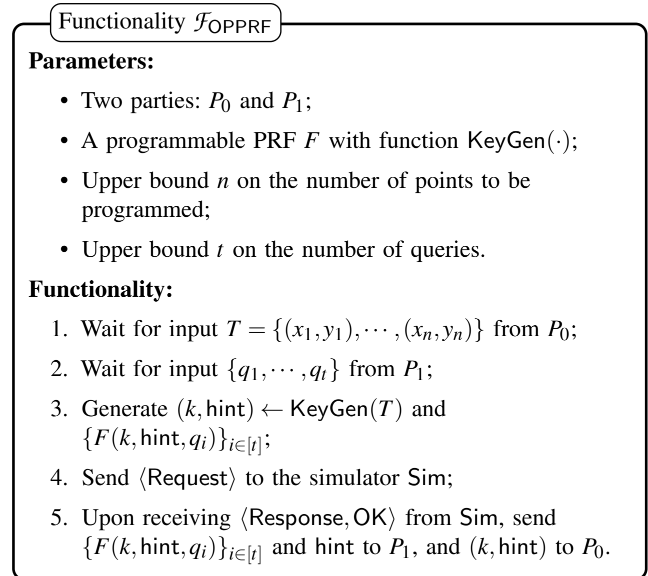
Parameters:

- Two parties: P_0 and P_1.

Functionality:

1. Wait for input $\{x_0, x_1\}$ from P_0;
2. Wait for input $b \in \{0, 1\}$ from P_1;
3. Send $\langle \mathsf{Request} \rangle$ to the simulator Sim;
4. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, send output x_b to P_1.

Figure 2: 1-out-of-2 Oblivious Transfer functionality.


Functionality $\mathcal{F}_{\text{OPPRF}}$

Parameters:

- Two parties: P_0 and P_1;
- A programmable PRF F with function $\mathsf{KeyGen}(\cdot)$;
- Upper bound n on the number of points to be programmed;
- Upper bound t on the number of queries.

Functionality:

1. Wait for input $T = \{(x_1, y_1), \cdots, (x_n, y_n)\}$ from P_0;
2. Wait for input $\{q_1, \cdots, q_t\}$ from P_1;
3. Generate $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(T)$ and $\{F(k, \mathsf{hint}, q_i)\}_{i\in[t]}$;
4. Send $\langle \mathsf{Request} \rangle$ to the simulator Sim;
5. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, send $\{F(k, \mathsf{hint}, q_i)\}_{i\in[t]}$ and hint to P_1, and (k, hint) to P_0.

Figure 3: Oblivious Programmable PRF Functionality.

Private Equality Test. Private Equality Test (PET) is used to test whether two strings are equal. More concretely, the two parties P_0 and P_1 hold strings x_0 and x_1 respectively. PET outputs a bit b_0 to P_0 and b_1 to P_1 such that if $x_0 = x_1$, $b_0 \oplus b_1 = 0$, otherwise, $b_0 \oplus b_1 = 1$. The existing works [6, 9, 11] designed efficient PET protocols. We give the functionality $\mathcal{F}_{\text{PET}}$ in Figure 4.

Permute + Share. The Permute+Share functionality $\mathcal{F}_{\text{PS}}$ is defined by Chase et al. in [7]. There are two parties P_0 and P_1 in this functionality, where P_0 possesses a set $X = \{x_1, \cdots, x_n\}$ of size n and P_1 picks a permutation π on n elements. The goal of $\mathcal{F}_{\text{PS}}$ is to let P_0 learn the shares

Functionality $\mathcal{F}_{\mathsf{PET}}$

Parameters:

- Two parties: P_0 and P_1;

Functionality:

1. Wait for input x_0 from P_0;
2. Wait for input x_1 from P_1;
3. Generate b_0 and b_1 such that if $x_0 = x_1$, $b_0 \oplus b_1 = 0$, otherwise $b_0 \oplus b_1 = 1$;
4. Send $\langle \mathsf{Request} \rangle$ to the simulator Sim;
5. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, send b_1 to P_1, and b_0 to P_0.

Figure 4: Private Equality Test Functionality.

$\{s_{\pi(1)}, s_{\pi(2)}, \cdots, s_{\pi(n)}\}$ and P_1 learn nothing but the other shares $\{x_{\pi(1)} \oplus s_{\pi(1)}, x_{\pi(2)} \oplus s_{\pi(2)}, \cdots, x_{\pi(n)} \oplus s_{\pi(n)}\}$. As mentioned in [7], some earlier works [15, 23] can also be used for securely realizing $\mathcal{F}_{\mathsf{PS}}$. These solutions all have computation/communication complexity $O(n \log n)$. The functionality $\mathcal{F}_{\mathsf{PS}}$ is shown in Figure 5.

Functionality $\mathcal{F}_{\mathsf{PS}}$

Parameters:

- Two parties: P_0 and P_1;
- Set size n for P_0;
- Length of element ℓ.

Functionality:

1. Wait for input $X = \{x_1, \cdots, x_n\}$ from P_0, abort if $|X| \neq n$, or $\exists\ x_i \in X$ such that $|x_i| > \ell$;
2. Wait for input a permutation π from P_1, abort if π is not a permutation on n items;
3. Send $\langle \mathsf{Request} \rangle$ to the simulator Sim;
4. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, give output shuffled shares $\{s_{\pi(1)}, s_{\pi(2)}, \cdots, s_{\pi(n)}\}$ to P_0, and another shuffled shares $\{x_{\pi(1)} \oplus s_{\pi(1)}, x_{\pi(2)} \oplus s_{\pi(2)}, \cdots, x_{\pi(n)} \oplus s_{\pi(n)}\}$ to P_1.

Figure 5: Permute + Share functionality.

Simple Hashing. In the simple hashing scheme, there are γ hash functions $h_i : \{0,1\}^* \rightarrow [b]$, where $i \in [\gamma]$, used to map n items into b bins $B_1, \cdots, B_b$. An item x will be added into $B_{h_1(x)}, B_{h_2(x)}, \cdots, B_{h_\gamma(x)}$, regardless of whether these bins are empty. The maximum bin size ρ can be set to ensure that no bin will contain more than ρ items except with probability $2^{-\lambda}$ when hashing n items into b bins.

Cuckoo Hashing. Cuckoo hashing was introduced by Pagh and Rodler in [24]. In this hashing scheme, there are γ hash functions $h_1, \cdots, h_\gamma$ used to map n items into $b = \varepsilon n$ bins and a stash, and we denote the i-th bin as B_i. Unlike the simple hashing, the Cuckoo hashing can guarantee that there is only one item in each bin, and the approach to avoid collisions is as follows: For an item x, it can be inserted into any empty bin of $B_{h_1(x)}, B_{h_2(x)}, \cdots, B_{h_\gamma(x)}$. If there are no empty bins in the k bins, randomly select a bin $B_{h_r(x)}$ in these γ bins, and evict the prior item y in $B_{h_r(x)}$ where $h_r(x) = h_r(y)$ to a new bin $B_{h_i}(y)$ where $i \neq r$. The above procedure is repeated until no more evictions are necessary, or until the number of evictions has reached a threshold. In the latter case, the last item will be put in the stash. According to the empirical analysis in [26], we can adjust the values of γ and ε to reduce the stash size to 0 while achieving a hashing failure probability of 2^{-40}. Moreover, the works [22, 30] have shown that Cuckoo hashing can theoretically achieve negligible failure probability.

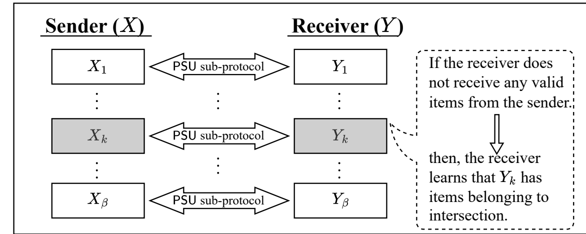


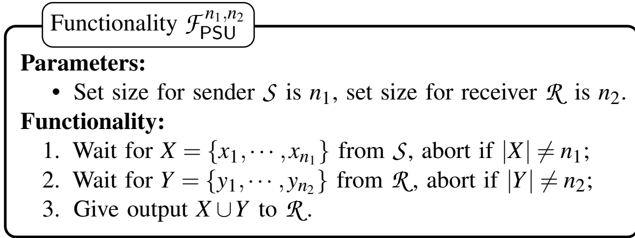
Figure 6: The leakage of protocol in [21].

3 Leakage Analysis

In this section, we thoroughly analyze the leakage in existing PSU constructions. Specifically, we observe that the symmetric key-based PSU constructions [1, 8, 13, 19, 21, 29, 31] suffer from the during-execution leakage, whereas the AHE-based PSU protocols [10, 12] can avoid it.

3.1 Revisiting the Leakage of PSU in [21]

The PSU protocol in [21] is performed in a "split-execute-assemble" paradigm. As shown in Figure 6, the sender and receiver map their input sets X and Y into two simple hash tables respectively, such that the set X (resp. Y) is divided into subsets $X_1, \cdots, X_\beta$ (resp., $Y_1, \cdots, Y_\beta$). Note that each bin of the sender is filled with a special item, and each bin of the receiver is filled with one special item and some dummy items. Then, for each pair of bins, the two parties execute a PSU sub-protocol. We briefly recall the leakage pointed out by Jia et al. [19] in Figure 6; if the receiver does not receive any valid items from the sender in the k-th bin, the receiver can know that there are items belonging to $X \cap Y$ in Y_k with an overwhelming probability once the sub-protocol in this bin is finished. Roughly speaking, there are two cases in which the receiver does not receive any valid items; Case_1 is that $X_k \neq \emptyset \wedge X_k \subseteq Y_k$, and Case_2 is that $X_k = \emptyset$. According to the analysis in [19], the receiver can learn that Case_1 happens



Functionality $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$

Parameters:

- Set size for sender $\mathcal{S}$ is n_1, set size for receiver $\mathcal{R}$ is n_2.

Functionality:

1. Wait for $X = \{x_1, \cdots, x_{n_1}\}$ from $\mathcal{S}$, abort if $|X| \neq n_1$;
2. Wait for $Y = \{y_1, \cdots, y_{n_2}\}$ from $\mathcal{R}$, abort if $|Y| \neq n_2$;
3. Give output $X \cup Y$ to $\mathcal{R}$.

Figure 7: The Original Ideal Functionality for PSU.

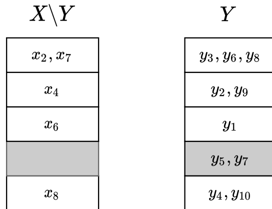


Figure 8: The receiver maps $X \setminus Y$ and Y into two simple hash tables respectively.

with an overwhelming probability, which means that she can know that there are items belonging to $X \cap Y$ in Y_k with the same probability. Please see Appendix A for more details about the leakage.

After reviewing the security proof in [21], we find that the protocol in [21] can indeed securely realize $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$ (recalled in Figure 7), which means that the functionality $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$ actually allows the leakage. Next, we will analyze the leakage from the perspective of ideal functionality.

3.2 Leakage Allowed in $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$

According to $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$, the simulator for the corrupted receiver only obtains the output, i.e., $X \cup Y$, which means that the leakage recalled above can be deduced from $X \cup Y$. Therefore, in any protocols that securely realize $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$, the receiver can obtain the leaked information once receiving *all the items* in $X \cup Y$, which will be explained by the following example:

Assuming the receiver's set $Y = \{y_1, y_2, \cdots, y_{10}\}$ and $X \setminus Y = \{x_7, x_2, x_4, x_6, x_8\}$, the receiver can map $X \setminus Y$ and Y into two simple hash tables respectively as shown in Figure 8. Note that even if the PSU protocol does not use the bucketing technique as in [21], the receiver can choose some hash functions to perform the mapping. We can see that in the left table, no items in $X \setminus Y$ are mapped to the 4th bin marked in gray. Then, according to the analysis in [19], the receiver can learn that items belonging to $X \cap Y$ are in $\{y_5, y_7\}$ with a high probability.

3.3 Leakage Occurring in Symmetric Key-based PSU Protocols

In this section, we further analyze the leakage of PSU protocol in [21] and the subsequent symmetric key-based PSU protocols [1, 8, 13, 19, 29, 31].

PSU Protocol in [21]. As analyzed before, the leakage of PSU protocol in [21] is actually allowed by the functionality $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$. However, we notice that the receiver in [21] can learn the leakage *during the execution*, rather than after receiving *all the items* in $X \cup Y$. We call the leakage occurring in [21] as "during-execution leakage". Recall the example shown in Figure 6, once completing the execution in the k-th bin, the receiver learns with an overwhelming probability that there are intersection items in Y_k.

Intuitively, the during-execution leakage in protocol [21] is incurred by the "split-execute-assemble" paradigm. Therefore, a natural question is *"Do the subsequent OT-based PSU protocols (including [1, 8, 13, 19, 29, 31] and the basic scheme in [21]) avoid the during-execution leakage?"* While these OT-based PSU protocols do not leverage the split-execute-assemble paradigm, we observe the answer is still negative.

OT-based PSU Protocols. We observe that all the OT-based PSU protocols in [1, 8, 13, 19, 29, 31] and the basic scheme in [21] follow the design framework shown in Figure 9. Concretely, the sender first randomly permute his set X. Then, through generalized Reversed Private Membership Test (g-RPMT), the receiver can learn a bit b_i for each item in X; $b_i = 0$ means $x_i \notin Y$, otherwise $x_i \in Y$. The receiver knows no more information about x_i beyond whether it belongs to Y, and the sender learns nothing about Y. At last, the receiver obtains x_i if $b_i = 0$, otherwise, obtains $\perp$. Intuitively, in the design framework shown in Figure 9, the set Y is processed as a whole and the receiver obtains the items in $X \setminus Y$ in random order. Therefore, the receiver in the design framework cannot obtain the leakage shown in Figure 6 during the execution. However, it can be observed that the receiver obtains $\{b_1, \cdots, b_{n_1}\}$ before obtaining the output $X \cup Y$. In other words, even if the receiver does not perform OTs with the sender after executing g-RPMT, the receiver can still learn $\{b_1, \cdots, b_{n_1}\}$. Therefore, $\{b_1, \cdots, b_{n_1}\}$ is also during-execution leakage.

Practical Influence. If any PSU protocol can ensure that the receiver obtains the entire output, it does not matter whether the leakage happens *after* the execution or *during* the execution[2]. However, in practice, many realistic factors (e.g., network interruptions and server failures) may interrupt the execution. Moreover, the receiver may refuse to finish the execution once he obtains sufficient leakage, and leverage the during-execution leakage to launch attacks.

More specifically, in some attacks, the attacker needs to collect information across multiple executions. However, in practice, victims may limit the number of executions for a

[2]Here, we ignore the difference in the time of obtaining the leakage.

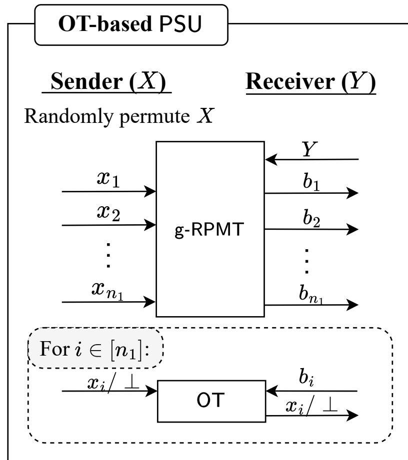


Figure 9: OT-based solutions [1, 8, 13, 19, 21, 29, 31].

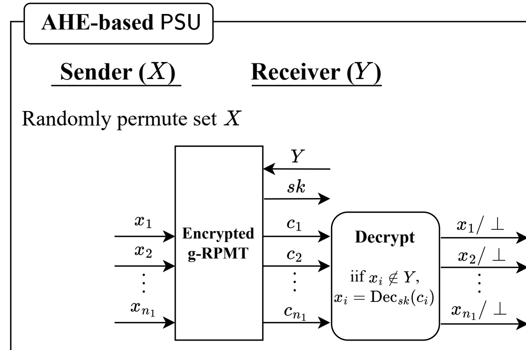


Figure 10: AHE-based solutions [10, 12].

period of time, but they may allow executions to be restarted after interruptions. Therefore, when the protocol suffers from during-execution leakage, the attacker could utilize the during-execution leakage: the attacker interrupts the execution once obtaining the leakage and then initiates a new execution. In this way, the attacker obtains more information within a certain period of time. As an example, we next explain how an attacker can use the during-execution leakage $\{b_1, \cdots, b_{n_1}\}$ in OT-based PSU to launch the attack shown in [14].

Guo et al. [14] launched attacks on protocols that aim to hide intersections but allow leaking intersection sizes, including Private Set Intersection Cardinality (PSI-CA), Private Intersection-Sum with Cardinality (PSI-SUM) and PSU. Specifically, the attacker can leverage the intersection sizes obtained across multiple executions to infer whether some elements are in a set (that is, intersection). They implemented the attack on practical datasets to obtain the tokens of COVID-19 patients, and the interest of the person associated with specific personal_id. The during-execution leakage $\{b_1, \cdots, b_{n_1}\}$ in OT-based PSU actually leaks intersection size. By launching the attack in [14] on the OT-based PSU, the attacker can interrupt the execution upon obtaining $\{b_1, \cdots, b_{n_1}\}$; the attacker can interact with the sender more times, thereby obtaining more intersection sizes to infer intersections.

Guo et al. [14] also mentioned that limiting the number of protocol invocations may be a potential defense. However, if the targeted protocol suffers from the during-execution leakage, only limiting the number of protocol invocations is not enough. Being attentive to interruption events and limiting the number of restarts after interruptions are also necessary.

Given the above example, we know that the during-execution leakage could be exploited by attackers. Fortunately, we observe that the during-execution leakage is *unnecessary*, as the AHE-based solutions [10, 12] can avoid it.

3.4 AHE-based PSU Protocols avoiding the During-Execution Leakage

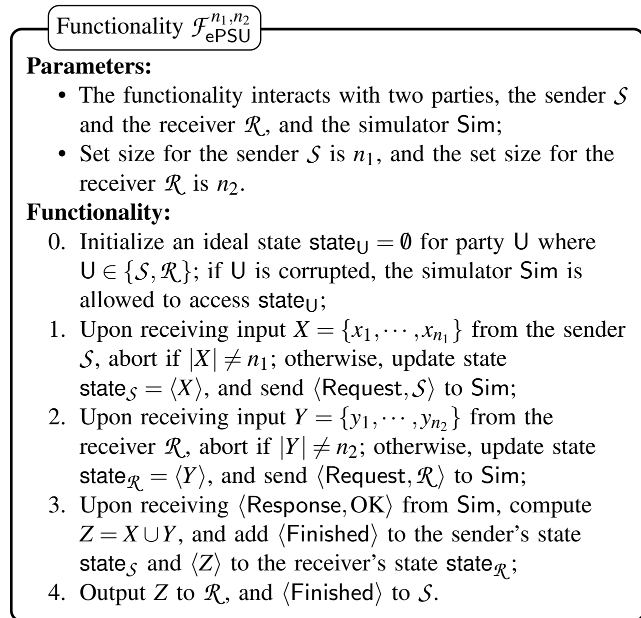
Different from the symmetric key-based solutions [1, 8, 13, 29, 31] following the design framework shown in Figure 9, the receiver in the AHE-based solutions [10, 12] do not need to first obtain the membership test result (i.e., $\{b_1, \cdots, b_{n_1}\}$) and then interact with the sender to obtain the items in $X \setminus Y$. We give the design framework used by the AHE-based solutions [10, 12] in Figure 10.

More specifically, the two parties in the schemes [10, 12] actually perform a *encrypted* g-RPMT, rather than a g-RPMT. The "encrypted" means that from the output of encrypted g-RPMT, the receiver can obtain a ciphertext c_i for each item x_i in the set X and a decryption key sk. Then, if the receiver can decrypt the ciphertext c_i to obtain a valid item x_i, this means that the corresponding bit $b_i = 0$, otherwise, $b_i = 1$. Note that in the AHE-based solutions [10, 12], the decryption key sk is the secret key of a key pair (pk, sk) chosen by the receiver. It can be observed that in this design framework shown in Figure 10, for an item in the set X, the receiver directly obtains the item itself or $\perp$ without needing to obtain extra information in advance. Therefore, the AHE-based protocols [10, 12] do not suffer from the during-execution leakage.

In summary, we discover that during-execution leakage is prevalent in symmetric key-based PSU protocols, but can be avoided by AHE-based PSU protocols. Given that attackers could exploit during-execution leakage to launch attacks, it is necessary to analyze whether a PSU protocol is susceptible to during-execution leakage when designing it. However, as previously analyzed, in the existing PSU functionality $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$ (see Figure 7), the during-execution leakage is allowed. Therefore, we next in Section 4 define a new PSU functionality in which the during-execution leakage is *not allowed*. In addition, since that the AHE-based PSU protocols are not efficient enough, in Section 5, we design a new symmetric key-based PSU while avoiding the during-execution leakage.

4 Defining a New PSU Functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$

As discussed above, the previous scalable protocols [1, 8, 13, 19, 21, 29, 31] all suffer from the during-execution leakage. However, these protocols have been proven to securely realize the functionality $\mathcal{F}_{\text{PSU}}^{n_1,n_2}$ shown in Figure 7. This means that the functionality $\mathcal{F}_{\text{PSU}}^{n_1,n_2}$ cannot capture the security without during-execution leakage. Therefore, in Figure 11, we define a new enhanced PSU functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$ that can subtly capture the security where the receiver cannot obtain any during-execution leakage before obtaining the output $X \cup Y$.


Functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$

Parameters:

- The functionality interacts with two parties, the sender $\mathcal{S}$ and the receiver $\mathcal{R}$, and the simulator Sim;
- Set size for the sender $\mathcal{S}$ is n_1, and the set size for the receiver $\mathcal{R}$ is n_2.

Functionality:

0. Initialize an ideal state $\mathsf{state}_{\mathsf{U}} = \emptyset$ for party U where $\mathsf{U} \in \{\mathcal{S}, \mathcal{R}\}$; if U is corrupted, the simulator Sim is allowed to access $\mathsf{state}_{\mathsf{U}}$;
1. Upon receiving input $X = \{x_1, \cdots, x_{n_1}\}$ from the sender $\mathcal{S}$, abort if $|X| \neq n_1$; otherwise, update state $\mathsf{state}_{\mathcal{S}} = \langle X \rangle$, and send $\langle \mathsf{Request}, \mathcal{S} \rangle$ to Sim;
2. Upon receiving input $Y = \{y_1, \cdots, y_{n_2}\}$ from the receiver $\mathcal{R}$, abort if $|Y| \neq n_2$; otherwise, update state $\mathsf{state}_{\mathcal{R}} = \langle Y \rangle$, and send $\langle \mathsf{Request}, \mathcal{R} \rangle$ to Sim;
3. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, compute $Z = X \cup Y$, and add $\langle \mathsf{Finished} \rangle$ to the sender's state $\mathsf{state}_{\mathcal{S}}$ and $\langle Z \rangle$ to the receiver's state $\mathsf{state}_{\mathcal{R}}$;
4. Output Z to $\mathcal{R}$, and $\langle \mathsf{Finished} \rangle$ to $\mathcal{S}$.

Figure 11: An enhanced ideal functionality for PSU.

Differences from $\mathcal{F}_{\text{PSU}}^{n_1,n_2}$. The main difference from the original PSU functionality $\mathcal{F}_{\text{PSU}}^{n_1,n_2}$ in Figure 7 is that the enhanced functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$ additionally sends an output $\langle \mathsf{Finished} \rangle$ to the sender once the joint computation is completed. We remark that returning $\langle \mathsf{Finished} \rangle$ to the sender is reasonable: in a natural real-world PSU protocol execution, the sender should be aware if the protocol execution has been completed or not; therefore, in the ideal world, the sender should also be informed of the completion. In addition, in the original functionality $\mathcal{F}_{\text{PSU}}^{n_1,n_2}$, the interactions between the functionality and the simulator (i.e., ideal world adversary), are not explicitly described. This presentation is consistent with that in [5], in which the simulator is in charge of the message delivery in the ideal world execution. In this work, to address the subtle issues in existing PSU protocols, we follow Canetti's original formulation [3, 4]; thus, we explicitly present the interactions between the functionality and the simulator. For example, when the PSU functionality receives an input, the simulator must be notified; when a player is corrupted, the simulator must be allowed to "see" the corresponding "ideal state".

During-Execution Leakage Not Allowed in $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$. Next, we take the OT-based design framework shown in Figure 9 as an example to intuitively explain how $\langle \mathsf{Finished} \rangle$ can help us to recognize that a protocol suffers from the during-execution leakage (i.e., a protocol with the during-execution leakage cannot securely realize $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$).

We assume that there is an adversary who corrupts the receiver and that at a certain moment t[3], g-RPMT has finished while OTs have not yet started. Then, the simulator needs to simulate $B = \{b_1, \cdots, b_{n_1}\}$ for the adversary. The simulator only has the two strategies: (1) Do send $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$ immediately; (2) Do not send $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$ immediately.

Following the first strategy, the simulator can simulate $B = \{b_1, \cdots, b_{n_1}\}$, but the sender will forward $\langle \mathsf{Finished} \rangle$ to the environment in the ideal world, while the environment in the real world will not obtain $\langle \mathsf{Finished} \rangle$ since the execution has not been completed in the real world. Therefore, the environment can distinguish the two worlds. On the other hand, if following the second strategy, regardless of whether in the real or ideal world, the environment will not obtain $\langle \mathsf{Finished} \rangle$. However, in the real world, the number of 1 in B is equal to $|X \cap Y|$. Note that the environment knows $|X \cap Y|$, as X and Y are chosen by the environment. Whereas, in the real world, the probability that the simulator does not guess $|X \cap Y|$ successfully is overwhelming. Therefore, the environment can still distinguish the two worlds. The idea of proof can be naturally extended to the protocol of [21] following the split-execute-assemble paradigm. We postpone the rigorous proof to Section 6.1.

As for the AHE-based protocols following the design framework shown in Figure 10, the simulator for corrupted receiver does not need extra information before simulating the ciphertexts of the encrypted g-RPMT. In order to simulate the ciphertexts, the simulator needs to obtain the output $X \cup Y$ from $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$, which means that the environment will receive $\langle \mathsf{Finished} \rangle$. Note that in the real world, once sending the ciphertexts, the sender completes the execution, so the environment can also obtain $\langle \mathsf{Finished} \rangle$. Therefore, the environment cannot distinguish the two worlds. Please refer to Section 6.2 for more details.

5 A New PSU $\Pi_{\text{PSU}}^{\text{bECRG}}$ Realizing $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$

In the previous section, we find that only the protocols following the AHE-based framework shown in Figure 10 can securely realize our new enhanced PSU functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$. However, the AHE-based protocols are not efficient enough in practice, especially for large datasets. In this section, we design a scalable PSU protocol $\Pi_{\text{PSU}}^{\text{bECRG}}$ that can securely realize $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$ by using symmetric-key operations. Next, we first give

[3] Note that the environment can know the internal state of the adversary, and thus know when the execution starts and when g-RPMT has finished while OTs have not yet started.

an overview of $\Pi^{\mathsf{bECRG}}_{\mathsf{PSU}}$, and then describe two new building blocks Π_{eqOTe} and Π_{bECRG}. Finally, we explain $\Pi^{\mathsf{bECRG}}_{\mathsf{PSU}}$ in detail and give a performance evaluation and comparison.

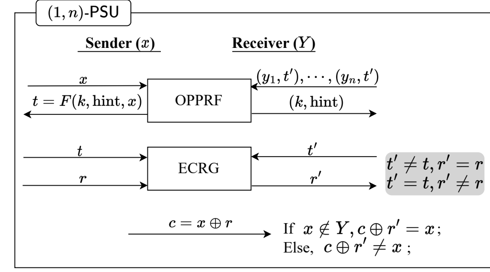
5.1 Overview

We observe that the difference in the way of "transmitting" the items in $X \setminus Y$ can lead to the difference in the security of PSU. More specifically, in the OT-based PSU (see Figure 9), the receiver "picks up" the items in $X \setminus Y$ from the sender's set X by using the information about $X \setminus Y$ obtained in advance. Whereas, the AHE-based PSU (see Figure 10) allows the receiver to directly obtain the items in $X \setminus Y$ by decryption without knowing any information about $X \setminus Y$ in advance, thus achieving $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$. However, the expensive public-key operations result in that the AHE-based PSU is not practical. Therefore, in order to design an efficient PSU protocol achieving $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$, a promising way is to design the "encrypted g-RPMT" only based on symmetric-key operations.

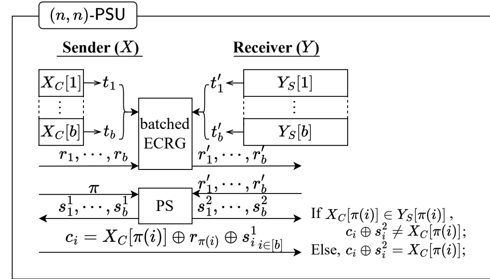
Our core idea is to use one-time pad (OTP) to encrypt the items in $X \setminus Y$, i.e., $c_i = x_i \oplus r_i$ where r_i is the secret key. Then, we need to guarantee that if $x_i \in X \setminus Y$, the receiver can obtain r_i and then learn x_i, otherwise, the receiver will obtain another randomness $r'_i \neq r_i$ such that the receiver cannot recover x_i by calculating $c_i \oplus r'_i$. Moreover, we need to guarantee that r_i and r'_i are pseudorandom, and thus not leaking extra information. Following the core idea, we propose a new design. For the sake of presentation, we first consider a simple case, denoted as $(1,n)-\mathsf{PSU}$, where the sender holds an item x and the receiver holds a set $Y = (y_1,\cdots,y_n)$. The design framework for $(1,n)-\mathsf{PSU}$ is shown in Figure 12a.

Simple case: $(\mathbf{1},\mathbf{n})-\mathsf{PSU}$. We first transfer the problem of testing whether $x \in Y$ to the problem of testing whether two strings t and t' are the same by leveraging "Oblivious Programmable PRF" (OPPRF). More specifically, in OPPRF, the receiver can set the PRF values of all the items in set Y to be a pseudorandom value t', such that the PRF value t of the sender's item would be equal to t' when $x \in Y$. Then, we propose a new protocol called "Equality-Conditional Randomness Generation" (ECRG) as a building block, such that if $t = t'$, ECRG outputs a pseudorandom string r' to the receiver, otherwise, outputs a string r' that is equal to r chosen by the sender. At last, following the above core idea, the receiver can obtain x only when $x \notin Y$.

General case: $(\mathbf{n},\mathbf{n})-\mathsf{PSU}$. It is natural to repeat $(1,n)-\mathsf{PSU}$ (shown in Figure 12a) n times to achieve $(n,n)-\mathsf{PSU}$, i.e., the sender's input set is $X = \{x_1,\cdots,x_n\}$. However, according to the existing OPPRF constructions [6, 25], the size of the hint in OPPRF is $O(n)$ and the corresponding computation cost is at least $O(n)$. Therefore, no matter how we design ECRG, the overall cost would be at least $O(n^2)$, which is not acceptable for large datasets. To reduce the cost, we insert set X and set Y into Cuckoo hashing and simple hashing with $b = \varepsilon n$ bins respectively, by using γ hash functions, such that



(a) $(1,n)-\mathsf{PSU}$. Here, OPPRF and ECRG constitute the "encrypted g-RPMT" in Figure 10 with one element x; r' corresponds to the key sk and $c = x \oplus r$ corresponds to the ciphertext of x.



(b) $(n,n)-\mathsf{PSU}$. Similarly, the OPPRF (to generate t_i and t'_i, but omitted here), the batched ECRG and PS constitute the "encrypted g-RPMT" in Figure 10 with input $\{X_C[1],\cdots,X_C[b]\}$; $\{s^2_1,\cdots,s^2_b\}$ corresponds to the key sk and c_i corresponds to the ciphertext of $X_C[\pi(i)]$.

Figure 12: Our new design framework achieving the "encrypted g-RPMT" based on symmetric-key techniques.

OPPRF is performed on small subsets of set Y. We give the design framework for $(n,n)-\mathsf{PSU}$ in Figure 12b.

More specifically, we denote the Cuckoo hash table and the simple hash table after insertion as X_C and Y_S, respectively. Each bin i of the Cuckoo hash table contains only one item $X_C[i]$, and each bin i of the simple hash table contains a subset $Y_S[i]$. Then, we perform an OPPRF to generate (t_i, t'_i) for all $i \in [b]$. Note that there are γn items in the simple hash table, and thus the overall cost for generating $\{(t_i,t'_i)\}_{i\in[b]}$ is $O(\gamma n)$. Also, we designed a batched ECRG (see Section 5.2) that can generate $\{r'_i\}_{i\in[b]}$ in a batched way and the cost is $O(n)$. At first glance, the sender can directly send $c_i = X_C[i] \oplus r_i$ for all $i \in [b]$, and the receiver can recover $X \setminus Y$ by using $\{r'_1,\cdots,r'_b\}$. However, the receiver will learn which bin each item in $X \setminus Y$ is mapped to, which will leak the information about set X (please refer to [21] for more discussions). Furthermore, like in [21], the receiver will know which subsets of the set Y contain items in $X \cap Y$. To achieve the enhanced functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$, we leverage Permute+Share (PS) to share $\{r'\}_{i\in[b]}$ while permuting it by using a random permutation π only known by the sender, i.e., $s^1_i \oplus s^2_i = r'_{\pi(i)}$. Finally, the sender calculates $c_i = X_C[\pi(i)] \oplus r_{\pi(i)} \oplus s^1_i$ for all $i \in [b]$, then the

Functionality $\mathcal{F}_{\mathsf{eqOTe}}$

1. Wait for input $\{(x_{1,0},x_{1,1}),(x_{2,0},x_{2,1}),\cdots,(x_{m,0},x_{m,1})\}$ and $\boldsymbol{b}^0=(b_1^0,b_2^0,\cdots,b_m^0)$, where $b_i^0\in\{0,1\}$ and $x_{i,j}\in\{0,1\}^\ell$ from the sender P_0;
2. Wait for input $\boldsymbol{b}^1=(b_1^1,b_2^1,\cdots,b_m^1)$ from the receiver P_1 where $b_i^1\in\{0,1\}$;
3. Send $\langle\mathsf{Request}\rangle$ to the simulator Sim;
4. Upon receiving $\langle\mathsf{Response},\mathsf{OK}\rangle$ from Sim, output $\{x_{i,b_i^0\oplus b_i^1}\}_{i\in[m]}$ to P_1, and $\mathsf{Finished}$ to P_0.

Figure 13: Equality Oblivious Transfer Extension Functionality.

receiver recovers $X\setminus Y$ by using $\{s_1^2,\cdots,s_b^2\}$. Note that the receiver can not learn the corresponding bin of each item in $X\setminus Y$ without knowing the permutation π. Some earlier works [7,15,23] can be used for securely realizing $\mathcal{F}_{\mathsf{PS}}$. These solutions all have computation/communication complexity $O(n\log n)$. Therefore, after optimization, the overall cost of our protocol is $O(\gamma n+n+n\log n)=O(n\log n)$.

We can see that the above process is actually an encrypted g-RPMT as shown in Figure 10. Specifically, the sender uses $r_{\pi(i)}\oplus s_i^1$ as a one-time secret key to encrypt the item $X_C[\pi(i)]$ as a ciphertext c_i, and the receiver can obtain s_i^2 that is equal to $r_{\pi(i)}\oplus s_i^1$ only when $X_C[\pi(i)]\notin Y$, to decrypt c_i.

It is worth mentioning that our protocol does not incur the security issues as in [21], although we leverage bucketing technique. This is because that for each pair of subsets, our protocol just generates intermediate states not leaking extra information, rather than executing a whole PSU sub-protocol where a subset of union is generated.

5.2 New Building Blocks

As mentioned before, our protocol $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ is based on a new building block called "batched Equality-Conditional Randomness Generation (bECRG)". In this section, we define the functionality of bECRG, denoted as $\mathcal{F}_{\mathsf{bECRG}}$. To design protocol Π_{bECRG} that UC-realize $\mathcal{F}_{\mathsf{bECRG}}$, we also propose another new building block called "Equality Oblivious Transfer extension (eqOTe)". Next, we first introduce eqOTe, and then give the functionality and protocol of bECRG.

Equality Oblivious Transfer extension. "Equality Oblivious Transfer extension (eqOTe)" is a variant of OT extension (OTe) [17]. Roughly speaking, the sender holds m pairs $\{(x_{i,0},x_{i,1})\}_{i\in[m]}$ and the receiver holds m bits $\{b_i^1\}_{i\in[m]}$. Through OT extension, the receiver obtains $\{x_{i,b_i^1}\}_{i\in[m]}$ by using only κ base OTs, where $m\gg\kappa$. In eqOTe, the sender additionally holds m bits $\{b_i^0\}_{i\in[m]}$, and the receiver obtains $\{x_{i,b_i^0\oplus b_i^1}\}_{i\in[m]}$. An eqOTe can be easily obtained from an OTe. Specifically, the sender sets $a_{i,b_i^0}=x_{i,0}$ and $a_{i,1\oplus b_i^0}=x_{i,1}$.

Protocol Π_{eqOTe}

Parameters:

- An OT extension protocol Π_{OTe};

Inputs:

- The sender P_0: $\{(x_{1,0},x_{1,1}),(x_{2,0},x_{2,1}),\cdots,(x_{m,0},x_{m,1})\}$ and $\boldsymbol{b}^0=(b_1^0,b_2^0,\cdots,b_m^0)$ where $b_i^0\in\{0,1\}$ and $x_{i,j}\in\{0,1\}^\ell$;
- The receiver P_1: $\boldsymbol{b}^1=(b_1^1,b_2^1,\cdots,b_m^1)$ where $b_i^1\in\{0,1\}$;

Protocol:

1. Sender P_0 sets $a_{i,b_i^0}=x_{i,0}$ and $a_{i,1\oplus b_i^0}=x_{i,1}$;
2. The parties invoke Π_{OTe}, where P_0 acts as the sender with input $\{(a_{i,0},a_{i,1})\}_{i\in[m]}$ and P_1 acts as the receiver with input $\boldsymbol{b}^1$, and P_1 can obtain $\{a_{i,b_i^1}\}_{i\in[m]}$. (If $b_i^0=b_i^1$, $a_{i,b_i^1}=a_{i,b_i^0}=x_{i,0}$, else $a_{i,b_i^1}=a_{i,b_i^0\oplus 1}=x_{i,1}$.)

Figure 14: Equality Oblivious Transfer Extension Protocol.

Then, the two parties perform OTe with $\{(a_{i,0},a_{i,1})\}_{i\in[m]}$ and $\{b_i^0\}_{i\in[m]}$ as inputs, respectively. Obviously, through OTe, if $b_i^1=b_i^0$ (i.e., $b_i^0\oplus b_i^1=0$), the receiver can obtain $a_{i,b_i^1}=a_{i,b_i^0}=x_{i,0}$, otherwise ($b_i^1=b_i^0\oplus 1$), the receiver can obtain $a_{i,b_i^1}=a_{i,b_i^0\oplus 1}=x_{i,1}$. We give the functionality $\mathcal{F}_{\mathsf{eqOTe}}$ and protocol Π_{eqOTe} in Figure 13 and Figure 14, respectively.

Batched Equality-Conditional Randomness Generation. Two parties P_0 and P_1 input strings t and t', respectively, and P_0 additionally inputs r. "Equality-Conditional Randomness Generation (ECRG)" generates another string r' to P_1 such that if $t\neq t'$, $r=r'$, otherwise, r' is a random string. In Figure 15, we give the functionality $\mathcal{F}_{\mathsf{bECRG}}$ of a batched version, where m pairs $\{t_i,t_i'\}_{i\in[m]}$ are as input, and $\{r_i'\}_{\in[m]}$ are output to P_1 based on whether $t_i=t_i'$.

We design protocol Π_{bECRG} (shown in Figure 16) by using $\mathcal{F}_{\mathsf{eqOTe}}$ and $\mathcal{F}_{\mathsf{PET}}$ (see Figure 4) as building blocks. More specifically, for each pair (t_i,t_i'), the two parties P_0 and P_1 invoke $\mathcal{F}_{\mathsf{PET}}$ and obtain b_i^0 and b_i^1 respectively, such that if $t_i=t_i'$, $b_i^0\oplus b_i^1=0$, otherwise, $b_i^0\oplus b_i^1=1$. To realize the functionality $\mathcal{F}_{\mathsf{bECRG}}$ (i.e., if $t_i\neq t_i'$, $r_i'=r_i$, otherwise, r_i' is a random string), P_0 sets $x_{i,0}$ as a random string and $x_{i,1}=r_i$, and the two parties invoke $\mathcal{F}_{\mathsf{eqOTe}}$ such that if $b_i^0\oplus b_i^1=0$, P_0 obtains $x_{i,0}$ (namely, a random string), otherwise, P_0 obtains $x_{i,1}=r_i$. We show the security of Π_{bECRG} in Theorem 5.1.

Theorem 5.1. *The protocol Π_{bECRG} shown in Figure 16) UC-realizes the functionality $\mathcal{F}_{\mathsf{bECRG}}$ (as in Figure 15) in the $\{\mathcal{F}_{\mathsf{PET}},\mathcal{F}_{\mathsf{eqOTe}}\}$-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 P_0 and the corrupted P_1, such that any PPT environment $\mathcal{E}$ cannot distinguish the execution in the ideal world from that in the real world.

Functionality $\mathcal{F}_{\mathsf{bECRG}}$

Parameters:

- The functionality interacts with two parties, P_0 and P_1, and the simulator Sim;
- Let ℓ_1 be the bit-length of each input items, and ℓ_2 be the bit-length of each output items;

Functionality:

0. Initialize an ideal state $\mathsf{state}_{\mathsf{U}} = \emptyset$ for party U where $\mathsf{U} \in \{P_0, P_1\}$; if U is corrupted, the simulator Sim is allowed to access U's state $\mathsf{state}_{\mathsf{U}}$;
1. Upon receiving input $\{t_1, \cdots, t_m\}$ and $\{r_1, \cdots, r_m\}$ from P_0 where $t_i \in \{0,1\}^{\ell_1}$ and $r_i \in \{0,1\}^{\ell_2}$, update state $\mathsf{state}_{p_0} = \langle \{t_1, \cdots, t_m\}, \{r_1, \cdots, r_m\} \rangle$, and send $\langle \mathsf{Request}, P_0 \rangle$ to the simulator Sim;
2. Upon receiving input $\{t'_1, \cdots, t'_m\}$ from P_1 where $t'_i \in \{0,1\}^{\ell_1}$, update state $\mathsf{state}_{P_1} = \langle \{t'_1, \cdots, t'_m \rangle$, and send $\langle \mathsf{Request}, P_1 \rangle$ to the simulator Sim;
3. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, if $t_i \neq t'_i$, set $r'_i = r_i$, otherwise, randomly choose $r'_i \xleftarrow{\$} \{0,1\}^{\ell_2}$;
4. Add $\langle \{r'_1, \cdots, r'_m\} \rangle$ to P_1's state_{P_1};
5. Output $\{r'_1, \cdots, r'_m\}$ to P_1.

Figure 15: Batched Equality-Conditional Randomness Generation Functionality.

Protocol Π_{bECRG}

Inputs:

- P_0: set $\{t_1, \cdots, t_m\}$ and set $\{r_1, \cdots, r_m\}$ where $t_i \in \{0,1\}^{\ell_1}$ and $r_i \in \{0,1\}^{\ell_2}$;
- P_1: set $\{t'_1, \cdots, t'_m\}, t'_i \in \{0,1\}^{\ell_1}$;

Protocol:

- For $i \in [m]$:
  - P_0 and P_1 invoke $\mathcal{F}_{\mathsf{PET}}$ (see Figure 4):
    - P_0 inputs t_i, and P_1 inputs t'_i;
    - P_0 obtains b_i^0, and P_1 obtains b_i^1;
  - P_0 chooses $x_{i,0} \xleftarrow{\$} \{0,1\}^{\ell_2}$ and sets $x_{i,1} = r_i$;
- P_0 and P_1 invoke $\mathcal{F}_{\mathsf{eqOTe}}$ (see Figure 13):
  - P_0 inputs $\{(x_{1,0}, x_{1,1}), (x_{2,0}, x_{2,1}), \cdots, (x_{m,0}, x_{m,1})\}$ and $\boldsymbol{b}^0 = (b_1^0, b_2^0, \cdots, b_m^0)$, and P_1 inputs $\boldsymbol{b}^1 = (b_1^1, b_2^1, \cdots, b_m^1)$;
  - P_1 obtains $\{r'_1, \cdots, r'_m\}$, where $r'_i = x_{i, b_i^0 \oplus b_i^1}$;
- P_1 outputs $\{r'_1, \cdots, r'_m\}$.

Figure 16: Batched Equality-Conditional Randomness Generation Protocol.

Corrupted P_0: Simulator Sim simulates a real execution in which P_0 is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input $\{t_1, \cdots, t_m\}$ and $\{r_1, \cdots, r_m\}$ of P_0 directly, and externally send $\{t_1, \cdots, t_m\}$ and $\{r_1, \cdots, r_m\}$ to $\mathcal{F}_{\mathsf{bECRG}}$ and then receives $\langle \mathsf{Request}, P_0 \rangle$. When receiving t_i from $\mathcal{A}$, Sim randomly selects $b_i^0 \xleftarrow{\$} \{0,1\}$, and simulates the execution of Π_{PET}. When receiving $\{(x_{i,0}, x_{i,1})\}_{i \in [m]}$ and $\boldsymbol{b}^0$, the input of Π_{eqOTe}, from $\mathcal{A}$, Sim simulates the execution of Π_{eqOTe}. Finally, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\mathsf{bECRG}}$.

We argue that the outputs of Sim are indistinguishable from the real view of P_0 by the following hybrids:

Hyb_0: P_0's view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{PET} is replaced by b_i^0 chosen by Sim, and Sim runs the $\mathcal{F}_{\mathsf{PET}}$ simulator to produce the simulated view for P_0. The security of protocol Π_{PET} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_2: Same as Hyb_1 except that Sim runs the $\mathcal{F}_{\mathsf{eqOTe}}$ simulator to produce the simulated view for P_0. The security of protocol Π_{eqOTe} guarantees the view in simulation is computationally indistinguishable from the view in Hyb_1. The hybrid is the view output by Sim.

Corrupted P_1: Simulator Sim simulates a real execution in which P_1 is corrupted. Since $\mathcal{A}$ is semi-honest, Sim can obtain the input $\{t'_1, \cdots, t'_m\}$ of P_1 directly, and externally send $\{t'_1, \cdots, t'_m\}$ to $\mathcal{F}_{\mathsf{bECRG}}$ and then receives $\langle \mathsf{Request}, P_1 \rangle$. When receiving $\{t'_1, \cdots, t'_m\}$ from $\mathcal{A}$, Sim randomly selects b_i^1, and simulates the execution of Π_{PET}. Once receiving $\boldsymbol{b}^1$, the input of Π_{eqOTe}, from $\mathcal{A}$, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{\mathsf{bECRG}}$ and obtains $\{r'_1, \cdots, r'_m\}$. Finally, Sim simulates the execution of Π_{eqOTe} with $\{r'_1, \cdots, r'_m\}$ as output.

We argue that the outputs of Sim are indistinguishable from the real view of P_1 by the following hybrids:

Hyb_0: P_1's view in the real protocol.

Hyb_1: Same as Hyb_0 except that the output of Π_{PET} is replaced by b_i^1 chosen by Sim, and Sim runs the $\mathcal{F}_{\mathsf{PET}}$ simulator to produce the simulated view for P_1. The security of protocol Π_{PET} 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_{eqOTe} is replaced by $\{r'_1, \cdots, r'_m\}$ output by $\mathcal{F}_{\mathsf{bECRG}}$ and Sim runs the $\mathcal{F}_{\mathsf{eqOTe}}$ simulator to produce the simulated view for P_1. Regardless of whether r'_i is generated by Π_{eqOTe} or $\mathcal{F}_{\mathsf{bECRG}}$, it would be equal to r_i when $t'_i = t_i$, and be pseudorandom when $t'_i \neq t_i$. The security of protocol Π_{eqOTe} guarantees the view in simulation is computationally indistinguishable from the view in Hyb_1. The hybrid is the view output by Sim. $\square$

5.3 The Details of Protocol $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$

In this section, we detail our PSU protocol $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ (see Figure 17). As explained in Section 5.1, to achieve the enhanced functionality $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$, the core idea of $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ is to design the encrypted RPMT as in the AHE-based PSU: Each item $x_i \in X$ is encrypted by a one-time secret key r_i; if $x_i \in X \setminus Y$, the receiver will obtain r_i, otherwise, the receiver will obtain another random string $r'_i \neq r_i$, and thus cannot learn x_i. Due to the fact that our $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ only relies on symmetric-key operations other than base OTs, our $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ is much more efficient than the AHE-based PSU.

Protocol Π_{PSU}^{bECRG}

Parameters:

- Hash functions $h_1, \cdots, h_\gamma : \{0,1\}^{\ell_1} \to [b]$;
- A Cuckoo hash table based on $h_1, \cdots, h_\gamma$, with $b = \varepsilon \cdot n_1$ bins, stash size $s = 0$;
- A simple hash table based on $h_1, \cdots, h_\gamma$, with $b = \varepsilon \cdot n_1$ bins and bin size ρ, where $\rho = O(\log(\gamma n_2))$;

Inputs:

- Sender $\mathcal{S}$: set $X = \{x_1, \cdots, x_{n_1}\}, x_i \in \{0,1\}^{\ell_1}$;
- Receiver $\mathcal{R}$: set $Y = \{y_1, \cdots, y_{n_2}\}, y_i \in \{0,1\}^{\ell_1}$;

Protocol:

1. $\mathcal{S}$ inserts set X into the Cuckoo hash table, and fills empty bins with the dummy item d, then denotes the filled Cuckoo hash table as X_C and the item in i-th bin as $X_C[i]$; $\mathcal{R}$ inserts set Y into the simple hash table, then denotes the set of items in the i-th bin as $Y_S[i]$;
2. $\mathcal{S}$ randomly chooses t'_i from $\{0,1\}^{\ell_2}$ for all $i \in [b]$;
3. $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{OPPRF}$ (see Figure 3):
   - $\mathcal{S}$ acts as P_1 with input $\{X_C[i]\}_{i\in[b]}$, and $\mathcal{R}$ acts as P_0 with input $\{(Y_S[i][1], t'_i), \cdots, (Y_S[i][\rho], t'_i)\}_{i\in[b]}$;
   - $\mathcal{R}$ obtains $\{k_i\}_{i\in[b]}$ and hint, and $\mathcal{S}$ obtains hint and $\{F(k_i, \mathsf{hint}, X_C[i]))\}_{i\in[b]}$ (note that if $X_C[i] \in Y_S[i]$, $F(k_i, \mathsf{hint}, X_C[i]) = t'_i$, otherwise, $F(k_i, \mathsf{hint}, X_C[i]) \neq t'_i$);
4. $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{bECRG}$ (see Figure 15):
   - $\mathcal{S}$ randomly chooses $r_i \xleftarrow{\$} \{0,1\}^{\ell_1}$ for $i \in [b]$;
   - $\mathcal{S}$ acts as P_0 with input $\{F(k_i, \mathsf{hint}, X_C[i])\}_{i\in[b]}$ and $\{r_1, \cdots, r_b\}$, and $\mathcal{R}$ acts as P_1 with input $\{t'_1, \cdots, t'_b\}$;
   - $\mathcal{R}$ obtains $\{r'_1, \cdots, r'_b\}$ (if $F(k_i, \mathsf{hint}, X_C[i]) = t'_i$, $r'_i \neq r_i$, otherwise, $r'_i = r_i$);
5. $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{PS}$ (see Figure 5):
   - $\mathcal{R}$ acts as P_0 with input set $\{r'_1, \cdots, r'_b\}$, and $\mathcal{S}$ acts as P_1 with a random permutation π;
   - $\mathcal{S}$ and $\mathcal{R}$ obtains the shuffled share sets $\{s^1_1, s^1_2, \cdots, s^1_b\}$ and $\{s^2_1, s^2_2, \cdots, s^2_b\}$ respectively, where $s^1_i \oplus s^2_i = r'_{\pi(i)}$;
6. $\mathcal{S}$ performs permutation π on set $\{X_C[1] \oplus r_1, X_C[2] \oplus r_2, \cdots, X_C[b] \oplus r_b\}$ and obtains $\{e_1, e_2, \cdots, e_b\}$ where $e_i = X_C[\pi(i)] \oplus r_{\pi(i)}$;
7. For $i \in [b]$:
   - $\mathcal{S}$ sends $c_i = e_i \oplus s^1_i$ to $\mathcal{R}$;
   - If $c_i \oplus s^2_i \neq d$, $\mathcal{R}$ sets $Z = Z \cup \{c_i \oplus s^2_i\}$;
8. $\mathcal{R}$ outputs $Y \cup Z$, and $\mathcal{S}$ outputs Finished;

Figure 17: A new PSU protocol that can realize $\mathcal{F}_{ePSU}^{n_1,n_2}$ and only relies on symmetric-key techniques (ignoring base OTs).

The sender and receiver first insert their sets X and Y into a Cuckoo hash table and a simple hash table, respectively, and the two filled tables are denoted as X_C and Y_S. Each bin i of the Cuckoo hash table contains only one item $X_C[i]$, whereas each bin i of the simple hash table contains a set $Y_S[i]$. By invoking $\mathcal{F}_{OPPRF}$, the receiver can set the PRF values of the items in $Y_S[i]$ as the same t'_i. If $X_C[i] \in Y_S[i]$, the sender will obtain $F(k_i, \mathsf{hint}, X_C[i]) = t'_i$, otherwise, $F(k_i, \mathsf{hint}, X_C[i]) \neq t'_i$. Then, the sender randomly chooses r_i that is used to encrypt $X_C[i]$ for all $i \in [b]$. Through $\mathcal{F}_{bECRG}$, if $F(k_i, \mathsf{hint}, X_C[i]) \neq t'_i$ (i.e., $X_C[i] \notin Y_S[i]$), the receiver can obtain $r'_i = r_i$ and thus learning $X_C[i]$ later, otherwise, the receiver can obtain $r'_i \neq r_i$. By invoking $\mathcal{F}_{PS}$, $\{r'_1, \cdots, r'_b\}$ are shuffled and shared into $\{s^1_1, \cdots, s^1_b\}$ and $\{s^2_1, \cdots, s^2_b\}$ such that $s^1_i \oplus s^2_i = r'_{\pi(i)}$. Finally, the sender sends $\{X_C[\pi(i)] \oplus r_{\pi(i)} \oplus s^1_i\}_{i\in[b]}$, and the receiver calculates $X_C[\pi(i)] \oplus r_{\pi(i)} \oplus s^1_i \oplus s^2_i$ for all $i \in [b]$. Obviously, only when $r_{\pi(i)} = s^1_i \oplus s^2_i$ (i.e., $X_C[\pi(i)] \notin Y_S[i]$), the receiver can obtain $X_C[\pi(i)]$.

The correctness of Π_{PSU}^{bECRG} shown in Figure 17 is guaranteed unless collisions occur. The collisions can only come from Π_{OPPRF}, i.e., $X_C[i] \notin Y_S[i]$ but $F(k_i, \mathsf{hint}, X_C[i]) = t'_i$. By setting the output length ℓ_2 of $F(k, \cdot, \cdot)$ as $\lambda + \log(\varepsilon n_1)$, we can bound the probability of collision happening to $2^{-\lambda}$, where λ is the statistical security parameter. Next, we state the security of Π_{PSU}^{bECRG} in Theorem 5.2.

Theorem 5.2. *The protocol Π_{PSU}^{bECRG} shown in Figure 17 UC-realizes the functionality $\mathcal{F}_{ePSU}^{n_1,n_2}$ (as in Figure 11) in the $\{\mathcal{F}_{OPPRF}, \mathcal{F}_{bECRG}, \mathcal{F}_{PS}\}$-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 sender and the corrupted receiver, such that any PPT environment $\mathcal{E}$ cannot distinguish the execution in the ideal world from that in the real world.

Corrupted Sender: Simulator 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 $\mathcal{F}_{ePSU}^{n_1,n_2}$ and then receives $\langle \mathsf{Request}, \mathcal{S} \rangle$. When receiving $X_C[i]$ from $\mathcal{A}$, Sim randomly selects $t_i \xleftarrow{\$} \{0,1\}^{\ell_2}$ and hint, and simulates the execution of Π_{OPPRF}. Once receiving $\{t_1, \cdots, t_b\}$ and $\{r_1, \cdots, r_b\}$, the input of Π_{bECRG}, from $\mathcal{A}$, Sim simulates the execution of Π_{bECRG}. Upon receiving a permutation π from $\mathcal{A}$, Sim checks if it is a permutation of b items. If so, Sim randomly selects $\{s^1_1, \cdots, s^1_b\}$ where $s^1_i \in \{0,1\}^{\ell_1}$, and simulates the execution of Π_{PS}. After receiving $\{c_1, \cdots, c_b\}$, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}_{ePSU}^{n_1,n_2}$.

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_{OPPRF} is replaced by (hint, t_i) chosen by Sim, and Sim runs the $\mathcal{F}_{OPPRF}$ simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{OPPRF} guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Hyb_2: Same as Hyb_1 except that Sim runs the $\mathcal{F}_{bECRG}$ simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{bECRG} guarantees the view in simulation is computationally indistinguishable from the view in Hyb_1.

Hyb_3: Same as Hyb_2 except that the output of Π_{PS} is replaced by $\{s^1_1, \cdots, s^1_b\}$ chosen by Sim, and Sim runs the $\mathcal{F}_{PS}$

simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{PS} guarantees the view in simulation is computationally indistinguishable from the view in Hyb_2. The hybrid is the view output by Sim.

Note that after $\mathcal{A}$ sends $\{c_1, \cdots, c_b\}$, $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$ receives $\langle\mathsf{Response}, \mathsf{OK}\rangle$ from Sim and outputs $X \cup Y$ to the receiver $\mathcal{R}$. This guarantee that the receiver $\mathcal{R}$ outputs $X \cup Y$ after the sender $\mathcal{S}$ sends $\{c_1, \cdots, c_b\}$ in both worlds.

Corrupted Receiver: Simulator 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 $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$ and then receives $\langle\mathsf{Request}, \mathcal{R}\rangle$. When receiving $\{(Y_S[i][1], t'_i), \cdots, (Y_S[i][\rho], t'_i)\}$ from $\mathcal{A}$, Sim randomly selects (k_i, hint), and simulates the execution of Π_{OPPRF}. Once receiving $\{t'_1, \cdots, t'_b\}$, the input of Π_{bECRG}, from $\mathcal{A}$, Sim randomly selects $\{r'_1, \cdots, r'_b\}$ where $r_i \xleftarrow{\$} \{0,1\}^{\ell_1}$ and simulates the execution of Π_{bECRG}. Upon receiving $\{r'_1, \cdots, r'_b\}$ from $\mathcal{A}$, Sim randomly selects $\{s_1^2, \cdots, s_b^2\}$ where $s_i^2 \in \{0,1\}^{\ell_1}$, and simulates the execution Π_{PS}. Sim sends $\langle\mathsf{Response}, \mathsf{OK}\rangle$ to $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$. After receiving $Z = X \cup Y$ from $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$, Sim calculates $X' = X \setminus Y = Z \setminus Y$ and randomly selects a subset S' of $\{s_1^2, \cdots, s_b^2\}$, where $|S'| = |X \setminus Y|$. For each item $s'_i \in S'$, Sim sets $c_i = X'[i] \oplus s'_i$. Finally, Sim randomly chooses $c_i \xleftarrow{\$} \{0,1\}^{\ell_1}$ for $i \in \{|S'|+1, |S'|+2, \cdots, b\}$ and sends $\{c_1, \cdots, c_b\}$ to $\mathcal{A}$ in random order.

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_{OPPRF} is replaced by (hint, k_i) chosen by Sim, and Sim runs the $\mathcal{F}_{\mathsf{OPPRF}}$ simulator to produce the simulated view for $\mathcal{R}$. The security of protocol Π_{OPPRF} guarantees the view in simulation is computationally indistinguishable from the view in Hyb_0.

Hyb_2: Same as Hyb_1 except that the output of Π_{bECRG} is replaced by the $\{r'_1, \cdots, r'_b\}$ chosen by Sim, and Sim runs the $\mathcal{F}_{\mathsf{bECRG}}$ simulator to produce the simulated view for $\mathcal{S}$. The security of protocol Π_{bECRG} guarantees the view in simulation is computationally indistinguishable from the view in Hyb_1.

Hyb_3: Same as Hyb_2 except that the output of Π_{PS} is replaced by $\{s_1^2, \cdots, s_b^2\}$ chosen by Sim, and Sim runs the $\mathcal{F}_{\mathsf{PS}}$ simulator to produce the simulated view for $\mathcal{S}$. The corresponding $\{c_1, \cdots, c_b\}$ are also changed according to $\{s_1^2, \cdots, s_b^2\}$. The security of protocol Π_{PS} and the random permutation π guarantee the view in simulation is computationally indistinguishable from the view in Hyb_2. The hybrid is the view output by Sim.

Note that after $\mathcal{A}$ receives $\{s_1^2, \cdots, s_b^2\}$, $\mathcal{F}_{\mathsf{ePSU}}^{n_1,n_2}$ receives $\langle\mathsf{Response}, \mathsf{OK}\rangle$ from Sim and outputs Finished to the sender $\mathcal{S}$. This guarantees that no matter whether in the ideal world or the real world, the sender $\mathcal{S}$ outputs Finished after completing the interaction with the receiver. □

Table 1: The theoretical complexities of $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$

	Π_{OPPRF}	Π_{bECRG}	Π_{PS}	Ciphertexts
Comp.	$O(\gamma n_2 + \varepsilon n_1)$	$O(\varepsilon n_1)$	$O(\varepsilon n_1 \log(\varepsilon n_1))$	$O(\varepsilon n_1)$
Comm.	$O(\gamma n_2)$	$O(\varepsilon n_1)$	$O(\varepsilon n_1 \log(\varepsilon n_1))$	$O(\varepsilon n_1)$

1. Here, n_1 is the sender's set size and n_2 is the receiver's set size.
2. The Cuckoo hash table and the simple hash table use γ hash functions and εn_1 bins.

5.4 Cost Analysis

We describe the theoretical complexities of our new construction $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ in Table 1; here, we assume that the sender's set size is n_1 and the receiver's set size is n_2. Recall that in $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ (see Figure 17), the sender's set and the receiver's set are initially inserted into a Cuckoo hash table and a simple hash table, respectively, using γ hash functions, and the number of bins is εn_1. After insertion, the Cuckoo hash table contains εn_1 items, and the simple hash table includes γn_2 items. Then, the two parties proceed with the subsequent steps on the two tables, which involve performing Π_{OPPRF}, Π_{bECRG}, Π_{PS}, as well as computing and sending ciphertexts.

We use the batched OPPRF in [6] that can hide the number of items in each bin, to implement Π_{OPPRF}. Thus, padding each bin in the simple hash table up to the maximum bin size, is not required. Specifically, the complexity for hint computation and communication is linear (i.e., $O(\gamma n_2)$) and the sender additionally needs $O(\varepsilon n_1)$ to compute the PRF values. As in Figure 16, our Π_{bECRG} consists of Π_{PET} and Π_{eqOTe}, both exhibiting linear complexity; thus the computation and communication costs of Π_{bECRG} are $O(\varepsilon n_1)$. We use the construction in [23] to implement Π_{PS}, leading to $O(\varepsilon n_1 \log(\varepsilon n_1))$ costs. Finally, the sender needs to compute and send a ciphertext for each item in the Cuckoo hash table, and the costs are $O(\varepsilon n_1)$. Overall, the computation and communication complexity of our protocol $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ is $O(n_1 \log n_1 + n_2)$, making $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ more suitable for balanced sets, i.e., the sizes of the two sets are comparable.

5.5 Performance Evaluation

In this section, we experimentally evaluate our protocol $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ and compare with the previous works.

Benchmarking Environment. We implement our protocol $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ in C++, which is available on GitHub: https://github.com/yanxue820/SecurePSU.git. Our experiments are conducted on a server equipped with two Intel Xeon Silver 4116 CPUs (2.10GHz) and 128GB RAM, running Ubuntu. We evaluate our protocol $\Pi_{\mathsf{PSU}}^{\mathsf{bECRG}}$ in two network settings, LAN network with 10Gbps bandwidth and 0.02 ms RTT and WAN network with 100Mbps and 80ms RTT, which are emulated using Linux tc command. We leverage the constructions in [6] to implement Π_{OPPRF} and Π_{PET} (the building block of Π_{bECRG}). We implement Π_{PS} using the

		Protocol		2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
				set size n							
Time (s)	LAN	[10]		11.78	44.73	175.7	702.4	2836.5	11341.2	-	-
		[31]		-	-	-	0.68	2.70	10.82	44.78	-
		$\Pi_{\text{PSU}}^{\text{bECRG}}$	Total	1.6	1.64	1.77	2.23	4.61	12.98	49.38	202.49
			w/o setup	1.04	1.08	1.2	1.61	3.73	11.12	43.49	180.55
	WAN	[10]		-	-	-	-	-	-	-	-
		[31]		-	-	-	12.87	16.04	28.58	86.31	-
		$\Pi_{\text{PSU}}^{\text{bECRG}}$	Total	5.31	5.86	7.06	10.22	21.07	77.56	225.32	987.945
			w/o setup	2.92	3.47	4.62	7.51	16.59	71.19	200.32	889.88
Comm.(MB)	[10]			2.83	11.32	45.28	181.12	724.49	2897.97	-	-
	[31]			-	-	-	6.52	26.03	103.85	414.43	-
	$\Pi_{\text{PSU}}^{\text{bECRG}}$			2.03	2.88	8.59	32.92	137.42	577.73	2430.47	10204.7

Table 2: Comparisons of total runtime (in seconds) and communication (in MB) between $\Pi_{\text{PSU}}^{\text{bECRG}}$ and [10] in WAN (100Mbps bandwidth, 80 ms RTT) and LAN (10Gbps bandwidth, 0.02 ms RTT) settings, where $n_1 = n_2 = n$. The results of [10] and [31] are both sourced from their papers. The implementation of [10] is in Go using 8 threads. The implementation of SKE-PSU [31] is in Java using a single thread; when using the OKVS in [1], the communication and runtime can be improved 2% and 6-9%, respectively. Our protocol $\Pi_{\text{PSU}}^{\text{bECRG}}$ is implemented in C++, using a single thread. Setup in our protocol refers to base OTs and generating triples for PET. Our $\Pi_{\text{PSU}}^{\text{bECRG}}$ and [10] can achieve the enhanced functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$, while SKE-PSU [31] can not.

code from [19] that is based on Oblivious Switching Network (OSN) [23]. As for OT extension, we use libOTe library [28].

Parameters. We set the computational security parameter $\kappa = 128$ and the statistical security parameter $\lambda = 40$, and item length is 128 bits. We use $\gamma = 3$ hash functions to insert sets X and Y into the Cuckoo hash table and simple hash table, respectively, and ε for Cuckoo hash table is set as 1.27.

Comparisons. We show performance comparisons with the previous works [10,31] in Table 2. Before our work, only two AHE-based schemes in [10,12] that can achieve the enhanced PSU functionality, i.e., $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$. The design by Davidson et al. [10] can be seen as an improvement of the one in [12]. Moreover, only Davidson et al. [10] provided experimental results. Therefore, we compare the performance of our $\Pi_{\text{PSU}}^{\text{bECRG}}$ with that of the scheme in [10] in Table 2. We can see that, for $n_1 = n_2 = 2^{18}$, our total runtime is 873.74$\times$ faster than that of [10] in the LAN setting, while their implementation is in 8 threads and ours is in a single thread; our total communication cost is 5$\times$ less than theirs.

Currently, the PSU by Zhang et al. [31] is the state-of-the-art work for large balanced sets, but it can not achieve the enhanced PSU functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$. Zhang et al. [31] gave two designs: (i) SKE-PSU, using symmetric key techniques, and (ii) PKE-PSU[4], using public key techniques. Given that SKE-PSU demonstrates better performance than PKE-PSU when the Internet speed exceeds 100Mbps, we compare the performance of our $\Pi_{\text{PSU}}^{\text{bECRG}}$ with that of SKE-PSU in [31] in Table 2. It shows that the performance of our protocol $\Pi_{\text{PSU}}^{\text{bECRG}}$ is *comparable to that of SKE-PSU* in the LAN setting. However, the performance of SKE-PSU in the WAN setting is better than ours, due to the less communication cost. In addition, the new OKVS structure designed by Bienstock et al. [1] could be used as a building block to improve the performance of the design in [31]. Specifically, as reported in [1], in a 1Gbps network, the combination can obtain a 16-22% improvement in communication and a 28-40% reduction in runtime compared to PKE-PSU; it also shows a 2% improvement in communication and a 6-9% reduction in runtime compared to SKE-PSU. Nonetheless, it is important to note that the designs in [31] cannot achieve the enhanced PSU functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$, whereas our $\Pi_{\text{PSU}}^{\text{bECRG}}$ can.

To the best of our knowledge, the PSU protocol by Blanton et al. [2] is the only one based on generic MPC techniques, and it can achieve the enhanced PSU functionality $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$. Since its security is guaranteed by the generic MPC techniques, it is out of scope for this work. Nonetheless, here, we still compare the performance of our protocol with that of their scheme. Their paper provided experimental results on small input sets in a three-party and honest majority setting for 32-bit sized elements, using 1Gbps bandwidth. For $n_1 = n_2 = 2^{12}$, their runtime is 24.88s, while ours is 8.42s in the WAN setting. In addition, Kolesnikov et al. [21] calculated the communication cost of [2] for 2PC and 128-bit items; for $n_1 = n_2 = 2^{18}$, its communication cost would be 163208.76MB, which is 282.5$\times$ higher than ours.

6 Proofs for Previous Works

In this section, we formally show that the OT-based PSU protocols cannot UC-realize $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$, whereas the AHE-based PSU protocols can. The proof for OT-based protocols can be naturally extended to the protocol [21] relying on the "split-execute-assemble" paradigm, whose proof can be found in full version [18]. Given that the OT-based design framework is currently mainstream, we define a relaxed PSU functionality $\mathcal{F}_{\text{rPSU}}^{\text{b}}$ to capture its security.

[4]Zhang et al. [31] used PKE-PSU* to represent the version that does not perform point compression.

Protocol $\Pi^{\text{OT}}_{\text{Unified}}$

Inputs:

- Sender $\mathcal{S}$: set $X = \{x_1, \cdots, x_{n_1}\}$, where $x_i \in \{0,1\}^\ell$
- Receiver $\mathcal{R}$: set $Y = \{y_1, \cdots, y_{n_2}\}$, where $y_i \in \{0,1\}^\ell$;

Protocol:

1. The sender $\mathcal{S}$ randomly permutes the set X into the set X^*;
2. The two players $\mathcal{S}$ and $\mathcal{R}$ invoke the "generalized Reversed Private Membership Test" $\mathcal{F}_{\text{g-RPMT}}$ (see Figure 1):
   - $\mathcal{S}$ acts as sender with input set X^*;
   - $\mathcal{R}$ acts as receiver with input set Y;
   - $\mathcal{R}$ obtains output b_i for each $i \in [n_1]$;
3. $\mathcal{R}$ initializes set $Z = Y$;
4. The two players $\mathcal{S}$ and $\mathcal{R}$ **simultaneously**[a] invoke n_1 number of $\mathcal{F}_{\text{OT}}$ (see Figure 2) instances.
   In the i-th instance, where $i \in [n_1]$,
   - $\mathcal{R}$ acts as receiver with input b_i;
   - $\mathcal{S}$ acts as sender with input $(X^*[i], \perp)$;
   - If $\mathcal{R}$ obtains $X^*[i]$, $\mathcal{R}$ sets $Z = Z \cup \{X^*[i]\}$;
5. $\mathcal{R}$ outputs Z, and $\mathcal{S}$ outputs Finished.

[a]Note that, in practice, the n_1 number of OT instances can be implemented by OT extension [17], so that the receiver can obtain all the items in $X \setminus Y$ at the same time.

Figure 18: The design framework unifying PSU protocols in [1,8,13,19,21,29,31]. Here, the protocol in [21] is the basic scheme without using "split-execute-assemble" paradigm.

6.1 OT-based PSU Protocols

We unify the OT-based protocols [1, 8, 13, 21, 29, 31] into the same framework $\Pi^{\text{OT}}_{\text{Unified}}$ as shown in Figure 18. Next, we show that the protocols unified in Figure 18 cannot UC-realize the new enhanced PSU functionality $\mathcal{F}^{n_1,n_2}_{\text{ePSU}}$. Formally, the security is stated in Theorem 6.1.

Theorem 6.1. *The protocol following the framework $\Pi^{\text{OT}}_{\text{Unified}}$ in Figure 18 cannot UC-realize functionality $\mathcal{F}^{n_1,n_2}_{\text{ePSU}}$ (as in Figure 11) in the $\{\mathcal{F}_{\text{g-RPMT}}, \mathcal{F}_{\text{OT}}\}$-hybrid model, against static, semi-honest adversaries.*

Proof. To complete the proof, we first construct an environment $\mathcal{E}$. Then we show that for any simulator Sim, this constructed $\mathcal{E}$ can tell the difference of the execution in the real world from that in the ideal world, with non-negligible probability.

Construction of environment $\mathcal{E}$. The environment $\mathcal{E}$ chooses sets X and Y as the inputs of the sender and the receiver, respectively. Since the environment $\mathcal{E}$ knows both X and Y, $\mathcal{E}$ of course knows the size of $X \cap Y$, denoted as m. In other words, the environment $\mathcal{E}$ knows the number of 1's in $\{b_1, \cdots, b_{n_1}\}$ is m. The environment $\mathcal{E}$ instructs the dummy adversary $\mathcal{A}$ to corrupt the receiver at the beginning of the protocol execution, and then chooses a time t. Finally, if the number of 1's in $\{b_1, \cdots, b_{n_1}\}$ reported by the dummy adversary $\mathcal{A}$ is m and the message Finished has not been reported by the honest sender $\mathcal{S}$ at the time t, the environment $\mathcal{E}$ outputs 1, otherwise, outputs 0.

The real world execution. In the real world, the Π_{OT} subprotocol instances will not be executed until the execution of $\Pi_{\text{g-RPMT}}$ sub-protocol instance is finished. Therefore, there is a time t when the receiver obtains $\{b_1, \cdots, b_{n_1}\}$ but not the items in the set $X \setminus Y$. Note that, the receiver is corrupted and under the control by the semi-honest real world adversary $\mathcal{A}$, the bit set $\{b_1, \cdots, b_{n_1}\}$ must be reported to the environment at the time t. Note also that, the protocol execution has not been finished, the honest sender $\mathcal{S}$ is *not supposed to return* the message Finished to the environment $\mathcal{E}$ at the time t.

The ideal world execution. In the ideal world, since the (dummy) receiver is corrupted, the simulator Sim is allowed to access the ideal state $\text{state}_{\mathcal{R}} = \langle Y \rangle$. After receiving from the functionality $\mathcal{F}^{n_1,n_2}_{\text{ePSU}}$ the message $\langle \text{Request}, \mathcal{R} \rangle$, to simulate $\{b^{\text{ideal}}_1, \cdots, b^{\text{ideal}}_{n_1}\}$, the simulator Sim *must* face the following two simulation strategies:

- *Do send $\langle \text{Response}, OK \rangle$ to the functionality $\mathcal{F}^{n_1,n_2}_{\text{ePSU}}$, immediately.* Note that, now the functionality will update the ideal states into $\text{state}_{\mathcal{S}} = \langle X, \text{Finished} \rangle$ and $\text{state}_{\mathcal{R}} = \langle Y, Z \rangle$, and immediately report Finished to the environment $\mathcal{E}$.
- *Do not send $\langle \text{Response}, OK \rangle$ to the functionality $\mathcal{F}^{n_1,n_2}_{\text{ePSU}}$, immediately.* Note that, now the functionality will **not** update the ideal states into $\text{state}_{\mathcal{S}} = \langle X, \text{Finished} \rangle$ and $\text{state}_{\mathcal{R}} = \langle Y, Z \rangle$; of course, no output Finished will be reported to $\mathcal{E}$ immediately.

Security analysis. Now, we can see, if the simulator follows the first simulation strategy, the environment will tell the difference with probability 1, since in the real world, no output Finished will be reported while there is Finished in the ideal world. If the simulator follows the second simulation strategy, the probability that $\mathcal{E}$ outputs 1 in the real world is 1 except negligible probability. However, in the ideal world, the simulator Sim does not know m. The probability that there are m 1's in $\{b^{\text{ideal}}_1, \cdots, b^{\text{ideal}}_{n_1}\}$ is $1/n_1$, which is far less than 1 when n_1 is large enough. Thus, the probability that $\mathcal{E}$ outputs 1 in the ideal world is far less than 1. Therefore, $\mathcal{E}$ can distinguish between the two worlds with non-negligible probability.

Note that, all simulations must follow one of the two strategies. Therefore, for all simulators, our constructed environment can tell the difference between the two worlds with non-negligible probability. This completes the proof. □

We define a relaxed PSU ideal functionality $\mathcal{F}^{\text{b}}_{\text{rPSU}}$ in Figure 19 to capture the security of OT-based design framework, and formally show the security in Theorem 6.2.

Theorem 6.2. *The protocol following the framework $\Pi^{\text{OT}}_{\text{Unified}}$ in Figure 18 UC-realizes the functionality $\mathcal{F}^{\text{b}}_{\text{rPSU}}$ (as in Figure 19) in the $\{\mathcal{F}_{\text{g-RPMT}}, \mathcal{F}_{\text{OT}}\}$-hybrid model, against static, semi-honest adversaries.*

Functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$

Parameters:

- The functionality interacts with two parties, the sender $\mathcal{S}$ and the receiver $\mathcal{R}$, and the simulator Sim;
- Set size for sender $\mathcal{S}$ is n_1; set size for receiver $\mathcal{R}$ is n_2.

Functionality:

0. Initialize an ideal state $\mathsf{state}_\mathsf{U} = \emptyset$ for party U where $\mathsf{U} \in \{\mathcal{S}, \mathcal{R}\}$; if U is corrupted, the simulator Sim is allowed to access to U's state $\mathsf{state}_\mathsf{U}$;
1. Upon receiving input $X = \{x_1, \cdots, x_{n_1}\}$ from the sender $\mathcal{S}$, abort if $|X| \neq n_1$; otherwise, update state $\mathsf{state}_\mathcal{S} = \langle X \rangle$, and send $\langle \mathsf{Request}, \mathcal{S} \rangle$ to the simulator Sim;
2. Upon receiving input $Y = \{y_1, \cdots, y_{n_2}\}$ from the receiver $\mathcal{R}$, abort if $|Y| \neq n_2$; otherwise, update state $\mathsf{state}_\mathcal{R} = \langle Y \rangle$, and send $\langle \mathsf{Request}, \mathcal{R} \rangle$ to the simulator Sim;
3. Upon receiving $\langle \mathsf{Response}, \mathsf{OK} \rangle$ from Sim, send $\langle \mathsf{Request_If} \rangle$ to the simulator Sim;
4. Upon receiving $\langle \mathsf{Response_If}, \mathsf{OK} \rangle$ from Sim, for each $i \in [n_1]$, set $b_i = 1$ if $X[i] \in Y$, otherwise set $b_i = 0$, then record $\{b_1, \cdots, b_{n_1}\}$ to the receiver's state $\mathsf{state}_\mathcal{R}$ and send $\langle \mathsf{Request_Item} \rangle$ to Sim;
5. Upon receiving $\langle \mathsf{Response_Item}, \mathsf{OK} \rangle$ from Sim, compute $Z = X \cup Y$, and record $\langle \mathsf{Finished} \rangle$ and $\langle Z \rangle$ to the sender's sate $\mathsf{state}_\mathcal{S}$ and the receiver's state $\mathsf{state}_\mathcal{R}$, respectively;
6. Output Z to $\mathcal{R}$, and Finished to $\mathcal{S}$.

Figure 19: A relaxed PSU ideal functionality leaking **set membership** in advance. Compared to the enhanced PSU functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$ in Figure 11, $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ additionally adds $\{b_1, \cdots, b_{n_1}\}$ into the receiver's state $\mathsf{state}_\mathcal{R}$ as shown in steps 3 - 4.

Proof. To prove this theorem, we will show that for any efficient adversary $\mathcal{A}$, we can construct a simulator Sim to properly simulate the view of the corrupted sender and the corrupted receiver, such that any PPT environment $\mathcal{E}$ cannot distinguish between the execution in the ideal world from that in the real world. In particular, according to the modular design of $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ from the sub-protocols $\Pi_{\mathsf{g\text{-}RPMT}}$ and Π_{OT}, the simulator Sim can be constructed by invoking the simulator Sim$'$ in [1, 8, 13, 19, 21, 29, 31].

Corrupted Sender: Simulator Sim first sends the input set X to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$. After receiving $\langle \mathsf{Request_If} \rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim first invokes Sim$'$ to simulate the execution of the sub-protocol $\Pi_{\mathsf{g\text{-}RPMT}}$. Then, Sim sends $\langle \mathsf{Response_If}, \mathsf{OK} \rangle$ to the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$. Once receiving $\langle \mathsf{Request_Item} \rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim simulates the execution of sub-protocol Π_{OT} by invoking Sim$'$. When $\mathcal{A}$ sends items in all Π_{OT} instances, Sim sends $\langle \mathsf{Response_Item}, \mathsf{OK} \rangle$ to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and then obtains $\langle \mathsf{Finished} \rangle$ from the sender's state $\mathsf{state}_\mathcal{S}$. Compared to the simulator for corrupted sender in [21], [19], [8], [29] or [13], Sim just additionally receives some request messages and additionally sends some response messages. Moreover, due to the request/response messages, the environment $\mathcal{E}$ will receive the honest receiver's output $X \cup Y$ at the same time in the real and ideal worlds. Therefore, Sim can simulate $\mathcal{A}$'s view such that $\mathcal{E}$ cannot distinguish the two worlds.

Corrupted Receiver: Likewise, simulator Sim first sends the input set Y to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$. After receiving $\langle \mathsf{Request_If} \rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim first invokes Sim$'$ to simulate the execution of the sub-protocol $\Pi_{\mathsf{g\text{-}RPMT}}$ except for the last step. To simulate the last step of $\Pi_{\mathsf{g\text{-}RPMT}}$, Sim sends $\langle \mathsf{Response_If}, \mathsf{OK} \rangle$ to the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and then obtains $\{b_1, b_2, \cdots, b_{n_1}\}$. Given $\{b_1, b_2, \cdots, b_{n_1}\}$, Sim can invoke Sim$'$ to simulate the last step of $\Pi_{\mathsf{g\text{-}RPMT}}$. Once receiving $\langle \mathsf{Request_Item} \rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim invokes Sim$'$ to simulate the execution of the sub-protocol Π_{OT} except for the last step (i.e., sending items). Then, Sim sends $\langle \mathsf{Response_Item}, \mathsf{OK} \rangle$ to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and then obtains set Z. By using the items in set $X \setminus Y$, Sim can simulate the last step of each Π_{OT} instance. Compared to the simulator for corrupted receiver in [1, 8, 13, 19, 21, 29, 31], Sim just additionally receives some request messages and additionally sends some response messages. Likewise, due to the request/response messages, the environment $\mathcal{E}$ will receive the honest sender's output Finished at the same time in the real and ideal worlds. Therefore, Sim can simulate $\mathcal{A}$'s view such that $\mathcal{E}$ cannot distinguish the two worlds. □

6.2 AHE-based PSU Protocols

In Figure 20, we show more details on how to achieve encrypted g-RPMT in AHE-based protocols [10, 12]. We can see that if the receiver $\mathcal{R}$ is corrupted, the simulator does not need to simulate anything for $\mathcal{R}$ before simulating $\{c_1, \cdots, c_{n_1}\}$. Therefore, the simulator does not need to obtain information from $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$ before simulating $\{c_1, \cdots, c_{n_1}\}$. Intuitively, the AHE-based protocols can UC-realize $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$. The formal security is stated in Theorem 6.3.

Theorem 6.3. *Given an IND-CPA secure AHE scheme, the protocol following the framework $\Pi^{\mathsf{AHE}}_{\mathsf{Unified}}$ in Figure 20 UC-realizes the functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$ (as in Figure 11), 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 sender and the corrupted receiver, such that any PPT environment $\mathcal{E}$ cannot distinguish the execution in the ideal world from that in the real world.

Corrupted Sender: The simulator Sim for the corrupted sender first sends the input set X to $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$. After receiving $\langle \mathsf{Request}, \mathcal{S} \rangle$ from $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$, Sim generates a key pair (pk, sk) and sends pk to $\mathcal{A}$. To simulate the ciphertext c from $\mathcal{R}$, Sim randomly generates a $f_Y(\cdot)$ according to the set size n_2 of Y, then encrypts it to c by using pk and sends c to $\mathcal{A}$. After $\mathcal{A}$ sends back $\{c_1, \cdots, c_{n_1}\}$ to $\mathcal{R}$, Sim sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to

Protocol $\Pi^{\mathsf{AHE}}_{\mathsf{Unified}}$

Parameters:

- An AHE scheme includes an encryption algorithm $\mathsf{Enc}_{pk}(\cdot)$ and a decryption algorithm $\mathsf{Dec}_{sk}(\cdot)$.

Inputs:

- Sender $\mathcal{S}$: set $X = \{x_1, \cdots, x_{n_1}\}$, where $x_i \in \{0,1\}^\ell$
- Receiver $\mathcal{R}$: set $Y = \{y_1, \cdots, y_{n_2}\}$, where $y_i \in \{0,1\}^\ell$;

Protocol:

1. The receiver $\mathcal{R}$ generates a key pair (pk, sk) and sends pk to the sender $\mathcal{S}$;
2. $\mathcal{R}$ represents set Y as $f_Y(\cdot)$, generates $c = \mathsf{Enc}_{pk}(f_Y)$ and then sends c to $\mathcal{S}$;
3. $\mathcal{S}$ randomly permutes set X to X^*;
4. $\mathcal{R}$ initializes set $Z = \emptyset$;
5. For each $i \in [n_1]$, $\mathcal{S}$ chooses a uniformly random value r_i, then generates $c_i = (\mathsf{Enc}_{pk}(r_i f_Y(X^*[i])), \mathsf{Enc}_{pk}(r_i X^*[i] f_Y(X^*[i])))$ based on the additive homomorphic property; $\mathcal{S}$ sends $\{c_1, \cdots, c_{n_1}\}$ to $\mathcal{R}$;
6. For each $i \in [n_1]$, $\mathcal{R}$ decrypts c_i to get (d_i^1, d_i^2); if $d_i^1 \neq 0$, $\mathcal{R}$ obtains $X^*[i] = d_i^2/d_i^1$ and sets $Z = Z \cup \{X^*[i]\}$, otherwise $\mathcal{R}$ obtains nothing;
7. $\mathcal{R}$ outputs Z, and $\mathcal{S}$ outputs $\mathsf{Finished}$.

Figure 20: The design framework unifying PSU protocols in [10, 12]. Note that $f_Y(\cdot)$ in [12] is a polynomial $P(x) = \prod_{i=1}^{n_2}(x - y_i)$ such that $P(x^*) = 0$ if $x^* \in Y$, and $f_Y(\cdot)$ in [10] is an inverted Bloom Filter B where Y is inserted by using hash functions $h_1, \cdots, h_\gamma$ such that $\sum_{i=1}^{\gamma} B[h_i(x^*)] = 0$ if $x^* \in Y$ (the "inverted" means that each bit value of the Bloom Filter containing Y is flipped).

$\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$. We can see that the only difference between the ideal world and the real world is that $f_Y(\cdot)$ is randomly generated in the ideal world while $f_Y(\cdot)$ is generated based on Y in the real world. The IND-CPA security of AHE scheme guarantees that any PPT environment $\mathcal{E}$ cannot distinguish between the real world from the ideal world.

Corrupted Receiver: The simulator Sim for the corrupted receiver first sends the input set Y to $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$ and then receives $\langle \mathsf{Request}, \mathcal{R} \rangle$. Sim will receive a public key pk and a ciphertext c from $\mathcal{A}$. To simulate $\{c_1, \cdots, c_{n_1}\}$, the simulator sends $\langle \mathsf{Response}, \mathsf{OK} \rangle$ to $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$ and obtains the union $Z = X \cup Y$. Then for $i \in \{1, 2, \cdots, |Z \setminus Y|\}$, Sim randomly picks α_i and generates $c_i = (\mathsf{Enc}_{pk}(\alpha_i), \mathsf{Enc}_{pk}(\alpha_i x_i))$, where $x_i \in Z \setminus Y$. After that, for all $i \in \{|Z \setminus Y| + 1, \cdots, n_1\}$, Sim generates $c_i = (\mathsf{Enc}_{pk}(0), \mathsf{Enc}_{pk}(0))$ by using pk. After randomly permuting the set $\{c_1, \cdots, c_{n_1}\}$, the simulator Sim sends the ciphertexts to $\mathcal{A}$. In both the ideal world and the real world, if $x_i \in X \setminus Y$, the corresponding c_i is a pair of ciphertexts for two messages α_i and $\alpha_i x_i$, otherwise it is the encryption of 0's. Moreover, $\mathcal{A}$ receives the items in $X \setminus Y$ in a random order in both worlds. Therefore, the ideal world and the real world are indistinguishable. □

7 Conclusion

In this work, we conduct a thorough analysis of the leakage in the typical PSU protocols. We identify a prevalent form of leakage in current PSU designs, called "during-execution leakage", which is implied by the output but can be obtained before the complete output is received. In addition, we find that the commonly used functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}}$ cannot capture the security without during-execution leakage. Therefore, we define a new enhanced functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$ to capture it.

Concretely, our investigation reveals that although symmetric key-based PSU constructions offer scalability, they are vulnerable to the during-execution leakage. On the other hand, only AHE-based PSU solutions can avoid the during-execution leakage, but their performance falls short of meeting practical requirements. To bridge the gap, we design a new PSU protocol $\Pi^{\mathsf{bECRG}}_{\mathsf{PSU}}$ that is the first scalable PSU protocol to UC-realize the enhanced PSU functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{ePSU}}$ in the semi-honest setting, by using a new building block Π_{bECRG}. Like OT-based PSU protocols, our $\Pi^{\mathsf{bECRG}}_{\mathsf{PSU}}$ only relies on symmetric key operations other than base OTs, obtaining significant performance improvement over AHE-based protocols.

Acknowledgments

We thank the anonymous reviewers for their insightful suggestions and comments. Yanxue Jia was supported in part by Supra Research. Shi-Feng Sun and Dawu Gu were supported in part by the National Key Research and Development Project 2020YFA0712300, and the National Natural Science Foundation of China (Grant No. 62272294). Hong-Sheng Zhou was supported in part by NSF grant CNS-1801470.

In this project, the third author mainly contributed to the definitions of ideal functionalities for PSU and the separation proofs between different versions of PSU functionalities.

References

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

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

[3] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[4] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[5] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

[6] Nishanth Chandran, Divya Gupta, and Akash Shah. Circuit-psi with linear complexity via relaxed batch OPPRF. *Proc. Priv. Enhancing Technol.*, 2022(1):353–372, 2022.

[7] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part III*, volume 12493 of *LNCS*, pages 342–372. Springer, Heidelberg, December 2020.

[8] Yu Chen, Min Zhang, Cong Zhang, Minglang Dong, and Weiran Liu. Private set operations from multi-query reverse private membership test. In Qiang Tang and Vanessa Teague, editors, *PKC 2024, Part III*, volume 14603 of *LNCS*, pages 387–416. Springer, 2024. https://eprint.iacr.org/2022/652.

[9] Geoffroy Couteau. New protocols for secure equality test and comparison. In Bart Preneel and Frederik Vercauteren, editors, *ACNS 18*, volume 10892 of *LNCS*, pages 303–320. Springer, Heidelberg, July 2018.

[10] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In Josef Pieprzyk and Suriadi Suriadi, editors, *ACISP 17, Part II*, volume 10343 of *LNCS*, pages 261–278. Springer, Heidelberg, July 2017.

[11] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS 2015*. The Internet Society, February 2015.

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

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

[14] 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 Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 1487–1504. USENIX Association, August 2022.

[15] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS 2012*. The Internet Society, February 2012.

[16] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *21st ACM STOC*, pages 44–61. ACM Press, May 1989.

[17] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 145–161. Springer, Heidelberg, August 2003.

[18] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, and Dawu Gu. Scalable private set union, with stronger security. Cryptology ePrint Archive, Report 2024/922, 2024. https://eprint.iacr.org/2024/922.

[19] Yanxue Jia, Shifeng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022*, pages 2947–2964. USENIX Association, August 2022.

[20] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1257–1272. ACM Press, October / November 2017.

[21] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 636–666. Springer, Heidelberg, December 2019.

[22] Brice Minaud and Charalampos Papamanthou. Generalized cuckoo hashing with a stash, revisited. *Inf. Process. Lett.*, 181(C), mar 2023.

[23] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 557–574. Springer, Heidelberg, May 2013.

[24] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms - ESA 2001*, volume 2161 of *Lecture Notes in Computer Science*, pages 121–133. Springer, 2001.

[25] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. Efficient circuit-based PSI with linear communication. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part III*, volume 11478 of *LNCS*, pages 122–153. Springer, Heidelberg, May 2019.

[26] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on ot extension. *ACM Trans. Priv. Secur.*, 21(2), January 2018.

[27] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. https://eprint.iacr.org/2005/187.

[28] Peter Rindal. libote: an efficient, portable, and easy to use oblivious transfer library. https://github.com/osu-crypto/libOTe.

[29] Binbin Tu, Yu Chen, Qi Liu, and Cong Zhang. Fast unbalanced private set union from fully homomorphic encryption. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 2959–2973, 2023.

[30] Kevin Yeo. Cuckoo hashing in cryptography: Optimal parameters, robustness and applications. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 197–230. Springer, Heidelberg, August 2023.

[31] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. Linear private set union from multi-query reverse private membership test. In *32st USENIX Security Symposium (USENIX Security 23)*, Anaheim, CA, August 2023. USENIX Association.

A The leakage in Kolesnikov et al.'s design [21]

In order to improve the performance, Kolesnikov et al. [21] proposed to optimize their protocol by using the bucketing technique, as shown in Figure 21. More specifically, the sender and receiver in [21] first assign their items in X and in Y, into two simple hash tables with the same number of bins, and the maximum bin sizes are assumed to be ρ_1 and ρ_2, respectively. Then they perform the (ρ_1,ρ_2)-PSU sub-protocol on the items of each bin separately. As pointed out by Kolesnikov et al. in [21], however, *the bucketing technique will leak the information "which bins contain items in $X \cap Y$" to the receiver.* To avoid this leakage, in [21] the receiver is required to put a special item $\perp$ into each bin, and to pad the bins with different dummy items d, while the sender pads his bins with the special item $\perp$. For example, in Figure 21, the items $\{x_6,x_2,x_{10}\}$ of X are mapped to the first bin of the sender's simple hash table, and the items $\{y_3,y_8\}$ of Y are mapped to the first bin of the receiver's hash table. Without the special item $\perp$, if $x_2 = y_3$, the receiver can learn that an item belonging to $X \cap Y$ is in $\{y_3,y_8\}$ after executing the (ρ_1,ρ_2)-PSU. By adding the special item $\perp$ to both sides, if the receiver learns that an item from the sender belongs to $\{y_3,\perp,y_8,d\}$, it seems that the receiver cannot know whether the item is a real item (namely, in X) or the special item $\perp$. Unfortunately, Jia et al. [19] pointed out that this strategy is insufficient to avoid the leakage incurred by the bucketing technique, and the detailed analysis is given below.

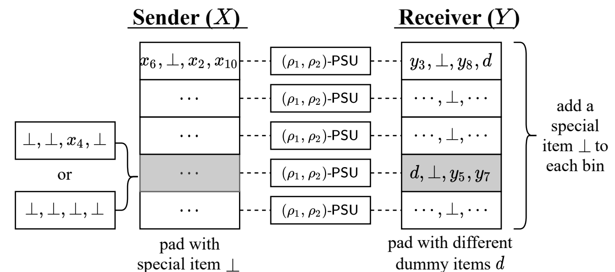


Figure 21: The bucketing technique in [21].

For ease of exposition, we take the 4th (ρ_1,ρ_2)-PSU sub-protocol in Figure 21 as an example to explain why the optimization in [21] fails to hide the intersection information. After the execution of the sub-protocol over the 4th bins, if the receiver does not obtain any items from the sender (that is, all items in the sender's 4th bin belong to the subset in the receiver's 4th bin i.e., $\{d,\perp,y_5,y_7\}$), then the receiver could obtain additional information about the intersection. Concretely, one of the following will occur:

- Case_1: all the real items that are mapped to the sender's bin (say x_4 in Figure 21) belong to $\{y_5,y_7\}$;
- Case_2: no real items are mapped to the sender's bin (i.e., all items are special item $\perp$).

The probabilities that Case_1 and Case_2 occur are denoted as $\Pr[\mathsf{Case}_1]$ and $\Pr[\mathsf{Case}_2]$, respectively. Clearly, if the receiver is able to determine that Case_1 occurs with certain (high) probability, she will know that items belonging to $X \cap Y$ are in $\{y_5,y_7\}$ with the same probability. According to the parameters in [21], Jia et al. [19] estimated $\Pr[\mathsf{Case}_2]$. Note that $\Pr[\mathsf{Case}_1] = 1 - \Pr[\mathsf{Case}_2]$, and the probability $\Pr[\mathsf{Case}_2]$ is very small. For example, when the set size is $n = 2^{20}$, $\Pr[\mathsf{Case}_2] = 5.778 \times 10^{-8}$. This means that when the receiver finds that all items in a bin belong to the intersection, she can learn that this bin has at least one real item with probability $1 - 5.778 \times 10^{-8}$, and that her corresponding bin contains at least an item in $X \cap Y$ with the same probability. Hence, their approach is insufficient to avoid the leakage incurred by the bucketing technique.