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
Scalable Private Set Union, with Stronger Security

Yanxue Jia\textsuperscript{1}, Shi-Feng Sun\textsuperscript{2}, Hong-Sheng Zhou\textsuperscript{3}, and Dawu Gu\textsuperscript{2}

\textsuperscript{1}Purdue University
\textsuperscript{2}Shanghai Jiao Tong University
\textsuperscript{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. 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., $\{(X_1,Y_1),(X_2,Y_2),\cdots,(X_B,Y_B)\}$, where $B \in \mathbb{N}$; Here, $|X| = N_1$ and $|Y| = N_2$; for all $i \in [B]$, $|X_i| \ll N_1$ and $|Y_i| \ll N_2$. Then, the two parties execute $B$ numbers 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 $Z = Z_1 \cup Z_2 \cup \cdots \cup Z_B$.

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

\textsuperscript{1}Part of the work was done at Shanghai Jiao Tong University.
\textsuperscript{2}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 leakage1.

**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 leak 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 $\text{Enc}(P(\cdot))$ to the sender. Then, for each $x \in X$, the sender calculates a ciphertext based on $\text{Enc}(P(\cdot))$. Finally, the receiver can obtain the items in $X \setminus Y$ via decryption, without knowing any membership 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 scalable PSU that does NOT 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 $F^{n_1,n_2}_{\text{ePSU}}$ in Section 4, that does not allow the leakage during the execution. The formal analysis is provided in Section 6.

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

Our design: 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 $F^{n_1,n_2}_{\text{ePSU}}$ 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 $t' = t$, 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.

**Performance comparison:** 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 $F^{n_1,n_2}_{\text{ePSU}}$ 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)”, “Permuate+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 \( \mathcal{F}_{g-RPMT} \) in Figure 1.

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 \( \mathcal{F}_{OT} \) 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 \( (k, \text{hint}) \), and the receiver obtains \( \{F(k, \text{hint}, q_i)\}_{i \in [t]} \) and hint. Note that if \( q_i = x_i \), \( F(k, \text{hint}, q_i) = y_i \), otherwise, \( F(k, \text{hint}, q_i) \) is pseudorandom. The functionality \( \mathcal{F}_{OPPRF} \) is given in Figure 3.

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 \( x_0 = x_1 \), \( b_0 = b_1 = 0 \), otherwise, \( b_0 = b_1 = 1 \). The existing works [6, 9, 11] designed efficient PET protocols. We give the functionality \( \mathcal{F}_{PET} \) in Figure 4.

Permuate + Share. The Permuate + Share functionality \( \mathcal{F}_{PS} \) 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 \( \pi \) on \( n \) elements. The goal of \( \mathcal{F}_{PS} \) is to let \( P_0 \) learn the shares.
Figure 4: Private Equality Test Functionality.

```plaintext
\{s_{\pi(1)}, s_{\pi(2)}, \cdots, s_{\pi(n)}\} \text{ and } P_1 \text{ 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)}\}. \text{ As mentioned in [7], some earlier works [15, 23] can also be used for securely realizing } F_{PS}. \text{ These solutions all have computation/communication complexity } O(n \log n). \text{ The functionality } F_{PS} \text{ is shown in Figure 5.}
```

Figure 5: Permute + Share functionality.

### Simple Hashing

In the simple hashing scheme, there are \( \gamma \) 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 \( B_{h_1(x)}(x), B_{h_2(x)}, \cdots, B_{h_\gamma(x)}(x) \), regardless of whether these bins are empty. The maximum bin size \( \rho \) can be set to ensure that no bin will contain more than \( \rho \) items except with probability \( 2^{-\delta} \) 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 \( \gamma \) hash functions \( h_1, \cdots, h_\gamma \) used to map \( n \) items into \( b = \lceil n \rceil \) 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 \( B_{h_\ell(x)} \) in these \( \gamma \) bins, and evict the prior item \( y \) in \( B_{h_\ell(x)} \) where \( h_\ell(x) = h_\ell(y) \) to a new bin \( B_{h_\ell(y)}(x) \). The previous 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 \( \gamma \) and \( \delta \) 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.

### 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 \( X_1, \cdots, X_b \) (resp., \( Y_1, \cdots, Y_b \)). 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 \( X_k \not= \emptyset \land X_k \subseteq Y_k \), and Case 2 is that \( X_k = \emptyset \). According to the analysis in [19], the receiver can learn that Case 1 happens.
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 \( F_{PSU}^{n_1,n_2} \) (recalled in Figure 7), which means that the functionality \( F_{PSU}^{n_1,n_2} \) actually allows the leakage. Next, we will analyze the leakage from the perspective of ideal functionality.

### 3.2 Leakage Allowed in \( F_{PSU}^{n_1,n_2} \)

According to \( F_{PSU}^{n_1,n_2} \), 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 \( F_{PSU}^{n_1,n_2} \), 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 \( F_{PSU}^{n_1,n_2} \). 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 \( \bot \). 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. 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.
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 $f_{PSU}^{\pi_1,\pi_2}$ (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 $X \cup Y$.

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 (Finished) 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 $r$, 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 $\langle \text{Response}, \text{OK} \rangle$ to $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$ immediately; (2) Do not send $\langle \text{Response}, \text{OK} \rangle$ 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 $\langle \text{Finished} \rangle$ to the environment in the ideal world, while the environment in the real world will not obtain $\langle \text{Finished} \rangle$ since the execution has not been completed in the real world. Therefore, the environment cannot 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 $\langle \text{Finished} \rangle$. 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 $\langle \text{Finished} \rangle$. Note that in the real world, once sending the ciphertexts, the sender completes the execution, so the environment can also obtain $\langle \text{Finished} \rangle$. Therefore, the environment cannot distinguish the two worlds. Please refer to Section 6.2 for more details.

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

In the previous section, we find that only the protocols following the AHE-based protocols do not need extra information before simulating the ciphertexts of the encrypted g-RPMT. However, the AHE-based protocols are not efficient enough in practice, especially for large datasets. In this section, we design a scalable PSU protocol $\Pi_{\text{ECRG}}^{\text{PSU}}$ that can securely realize $\mathcal{F}_{\text{ePSU}}^{n_1,n_2}$ by using symmetric-key operations. Next, we first give

\begin{itemize}
  \item 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.
\end{itemize}
an overview of \( \Pi_{\text{ECRG}}^{\text{PSU}} \), and then describe two new building blocks \( \Pi_{\text{SWOT}} \) and \( \Pi_{\text{ECRG}} \). Finally, we explain \( \Pi_{\text{ECRG}}^{\text{PSU}} \) 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 \( \{ \gamma \}^{n^2}_{\text{PSU}} \). 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 \( \{ \gamma \}^{n^2}_{\text{PSU}} \), 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 \( (1, n) - \text{PSU} \), where the sender holds an item \( x \) and the receiver holds a set \( Y = \{ y_1, \ldots, y_n \} \). The design framework for \( (1, n) - \text{PSU} \) is shown in Figure 12a.

**Simple case: \( (1, n) - \text{PSU} \).** 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 \( t' \) to the receiver, otherwise, outputs a string \( t' \) 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: \( (n, n) - \text{PSU} \).** It is natural to repeat \( (1, n) - \text{PSU} \) (shown in Figure 12a) \( n \) times to achieve \( (n, n) - \text{PSU} \), i.e., the sender’s input set is \( X = \{ x_1, \ldots, x_n \} \). 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 \( \gamma \) hash functions, such that OPPRF is performed on small subsets of set \( Y \). We give the design framework for \( (n, n) - \text{PSU} \) 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 \( n \) items in the simple hash table, and thus the overall cost for generating \( \{ (t_i, t'_i) \}_{i \in [b]} \) is \( O(n) \). Also, we designed a batched ECRG (see Section 5.2) that can generate \( \{ t'_i \}_{i \in [b]} \) 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 \( r'_i \). 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 \( \{ \gamma \}^{n^2}_{\text{PSU}} \), we leverage Permute+Share (PS) to share \( \{ r'_i \}_{i \in [b]} \) while permuting it by using a random permutation \( \pi \) only known by the sender, i.e., \( s_i' \oplus s_i = r'_i(\pi(i)) \). Finally, the sender calculates \( c_i = X_C[\pi(i)] \oplus r'_i(\pi(i)) \oplus s_i' \) for all \( i \in [b] \), then the
receiver recovers $X \setminus Y$ by using $\{s_1^2, \cdots, s_m^2\}$. Note that the receiver can not learn the corresponding bin of each item in $X \setminus Y$ without knowing the permutation $\pi$. Some earlier works [7,15,23] can be used for securely realizing $\mathcal{F}_{PS}$. These solutions all have computation/communication complexity $O(n \log n)$. Therefore, after optimization, the overall cost of our protocol is $O(\gamma n + n \log n) = O(n \log n)$.

We can see that the above process is actually an encrypted $g$-RPMT as shown in Figure 10. Specifically, the sender uses $r_{\pi(i)} \oplus s_i^2$ 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 $r_{\pi(i)} \oplus s_i^2$ 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 $\Pi^{BCRG}_{SU}$ 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 $\mathcal{F}_{bECRG}$. To design protocol $\Pi^{bECRG}_{SU}$ that UC-realize $\mathcal{F}_{bECRG}$, 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 $\{(x_i,0,x_i)\}_{i \in [m]}$ and the receiver holds $m$ bits $\{b_i\}_{i \in [m]}$. Through OT extension, the receiver obtains $\{x_i,b_i\}_{i \in [m]}$ by using only $k$ base OTs, where $m \gg k$. In $eqOTe$, the sender additionally holds $m$ bits $\{b_i^h\}_{i \in [m]}$, and the receiver obtains $\{x_i,b_i^h\}_{i \in [m]}$. An $eqOTe$ can be easily obtained from an OTe. Specifically, the sender sets $a_{i,b_i^h} = x_i,0$ and $a_{i,1\oplus b_i^h} = x_i,1$.

![Figure 13: Equality Oblivious Transfer Extension Functionality.](image)

![Figure 14: Equality Oblivious Transfer Extension Protocol.](image)

Then, the two parties perform OTe with $\{(a_i,0,a_i,1)\}_{i \in [m]}$ and $\{b_i^h\}_{i \in [m]}$ as inputs, respectively. Obviously, through OTe, if $b_i^h = b_i^h$ (i.e., $b_i^h \oplus b_i^h = 0$), the receiver can obtain $a_{i,b_i^h} = a_{i,1\oplus b_i^h} = x_i,0$, otherwise ($b_i^h = b_i^h + 1$), the receiver can obtain $a_{i,b_i^h} = a_{i,1\oplus b_i^h+1} = x_i,1$. We give the functionality $\mathcal{F}_{eqOTe}$ and protocol $\Pi^{eqOTe}_{SU}$ 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 $\mathcal{F}_{ECRG}$ of a batched version, where $m$ pairs $\{(t_i,t'_i)\}_{i \in [m]}$ are as input, and $\{r'_i\}_{i \in [m]}$ are output to $P_1$ based on whether $t_i = t'_i$.

We design protocol $\Pi^{ECRG}_{SU}$ (shown in Figure 16) by using $\mathcal{F}_{eqOTe}$ and $\mathcal{F}_{PET}$ (see Figure 4) as building blocks. More specifically, for each pair $(t_i,t'_i)$, the two parties $P_0$ and $P_1$ invoke $\mathcal{F}_{PET}$ and obtain $b_i^h$ and $b_i$, respectively, such that if $t_i = t'_i$, $b_i^h \oplus b_i = 0$, otherwise, $b_i^h \oplus b_i = 1$. To realize the functionality $\mathcal{F}_{ECRG}$ (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 $\mathcal{F}_{eqOTe}$ such that if $b_i^h \oplus b_i = 0$, $P_0$ obtains $x_i,0$ (namely, a random string), otherwise, $P_0$ obtains $x_i,1 = r_i$. We show the security of $\Pi^{ECRG}_{SU}$ in Theorem 5.1.

**Theorem 5.1.** The protocol $\Pi^{ECRG}_{SU}$ shown in Figure 16 UC-realizes the functionality $\mathcal{F}_{eqOTe}$ (as in Figure 15) in the $\{F_{PET}, \mathcal{F}_{eqOTe}\}$-hybrid model, against static, semi-honest adversaries.

**Proof.** We will show that for any adversary $\mathcal{A}$, we can construct a simulator $\mathcal{Sim}$ 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.
Corrupted $P_0$: Simulator $\Pi_{\text{PET}}$ simulates a real execution in which $P_0$ is corrupted. Since $A$ is semi-honest, $Sim$ can obtain the input $\{t_1, \ldots, t_m\}$ and $\{r_1, \ldots, r_m\}$ of $P_0$ directly, and externally send $\{t_1, \ldots, t_m\}$ and $\{r_1, \ldots, r_m\}$ to $f_{\text{BEGR}}$ and then receives $\langle \text{Request}, P_1 \rangle$. When receiving $t_i$, from $A$, $Sim$ randomly selects $b_i^0 \overset{\$}{\leftarrow} \{0, 1\}$, and simulates the execution of $\Pi_{\text{PET}}$. When receiving $\{\langle x_{i0,0}, x_{i1,0} \rangle \}_{i \leftarrow [m]}$ and $b^0_i$, the input of $\Pi_{\text{eqOTe}}$, from $A$, $Sim$ simulates the execution of $\Pi_{\text{eqOTe}}$. Finally, $Sim$ sends $\langle \text{Response}, OK \rangle$ to $f_{\text{BEGR}}$.

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 $\Pi_{\text{PET}}$ is replaced by $b^0_i$ chosen by Sim, and $Sim$ runs the $f_{\text{PET}}$ simulator to produce the simulated view for $P_0$. The security of protocol $\Pi_{\text{PET}}$ 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 $f_{\text{eqOTe}}$ simulator to produce the simulated view for $P_0$. The security of protocol $\Pi_{\text{eqOTe}}$ guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.

Corrupted $P_1$: Simulates a real execution in which $P_1$ is corrupted. Since $A$ is semi-honest, $Sim$ can obtain the input $\{t_1', \ldots, t_m'\}$ of $P_1$ directly, and externally send $\{t_1', \ldots, t_m'\}$ to $f_{\text{BEGR}}$ and then receives $\langle \text{Request}, P_1 \rangle$. When receiving $\{t_1', \ldots, t_m'\}$ from $A$, $Sim$ randomly selects $b^1_i$, and simulates the execution of $\Pi_{\text{PET}}$. Once receiving $b^1_i$, the input of $\Pi_{\text{eqOTe}}$, from $A$, $Sim$ sends $\langle \text{Response}, OK \rangle$ to $f_{\text{BEGR}}$ and obtains $\{r_1', \ldots, r_m'\}$. Finally, $Sim$ simulates the execution of $\Pi_{\text{eqOTe}}$ with $\{t_1', \ldots, t_m'\}$ 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 $\Pi_{\text{PET}}$ is replaced by $b^0_i$ chosen by Sim, and $Sim$ runs the $f_{\text{PET}}$ simulator to produce the simulated view for $P_1$. The security of protocol $\Pi_{\text{PET}}$ 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 $\Pi_{\text{eqOTe}}$ is replaced by $\{r_1', \ldots, r_m'\}$ output by $f_{\text{BEGR}}$ and $Sim$ runs the $f_{\text{eqOTe}}$ simulator to produce the simulated view for $P_1$. Regardless of whether $r_i$ is generated by $\Pi_{\text{eqOTe}}$ or $f_{\text{BEGR}}$, 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 $\Pi_{\text{eqOTe}}$ guarantees the view in simulation is computationally indistinguishable from the view in $Hyb_2$. The hybrid is the view output by $Sim$.

5.3 The Details of Protocol $\Pi^{bECRG}_{\text{PSU}}$

In this section, we detail our PSU protocol $\Pi^{bECRG}_{\text{PSU}}$ (see Figure 17). As explained in Section 5.1, to achieve the enhanced functionality $f_{\text{PSU}}^{bECRG}$, the core idea of $\Pi^{bECRG}_{\text{PSU}}$ 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 $\Pi^{PSU}_{\text{PSU}}$ only relies on symmetric-key operations other than base OTs, our $\Pi^{bECRG}_{\text{PSU}}$ is much more efficient than the AHE-based PSU.
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 $\mathcal{F}_{\text{OPPRF}}$, 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 $F(k_s, \text{hint}, X_C[i]) = t'_i$, otherwise, $F(k_s, \text{hint}, X_C[i]) \neq t'_i$.

Then, the sender randomly chooses $r_i$ that is used to encrypt $X_C[i]$ for all $i \in [b]$. Through $\mathcal{F}_{\text{SECRG}}$, if $F(k_s, \text{hint}, X_C[i]) \neq t'_i$ (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 $\mathcal{F}_{\text{PS}}$, $\{r'_1, \cdots, r'_b\}$ are shuffled and shared into $\{s'_1, \cdots, s'_b\}$ and $\{s''_1, \cdots, s''_b\}$ such that $s'_1 \oplus s''_1 = r'_0$. Finally, the sender sends $\{X_C[\pi(i)] \oplus \pi(i) + s'_1 \}$ for all $i \in [b]$. Obviously, only when $r_0 = s'_1 \oplus s''_1$ (i.e., $X_C[\pi(i)] \notin Y_S[i]$), the receiver can obtain $X_C[\pi(i)]$.

The correctness of $\Pi_{\text{SECRG}}$ shown in Figure 17 is guaranteed unless collisions occur. The collisions can only come from $\Pi_{\text{OPPRF}}$, i.e., $X_C[i] \notin Y_S[i]$ but $F(k_s, \text{hint}, X_C[i]) = t'_i$. By setting the output length $\ell_2$ of $F(k_s, \cdot)$ as $\lambda + \log(n_r)$, we can bound the probability of collision happening to $2^{-\lambda}$, where $\lambda$ is the statistical security parameter. Next, we state the security of $\Pi_{\text{SECRG}}$ in Theorem 5.2.

**Theorem 5.2.** The protocol $\Pi_{\text{SECRG}}$ shown in Figure 17 UC-realizes the functionality $f_{\text{epSU}}^{n, r_0}$ (as in Figure 11) in the $\{\mathcal{F}_{\text{OPPRF}}, \mathcal{F}_{\text{SECRG}}, \mathcal{F}_{\text{PS}}\}$-hybrid model, against static, semi-honest adversaries.

**Proof.** We will show that for any adversary $A$, we can construct a simulator $\text{Sim}$ 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 $\text{Sim}$ simulates a real execution in which the sender $S$ is corrupted. Since $A$ is semi-honest, $\text{Sim}$ can obtain the input $X$ of $S$ directly, and externally send the set $X$ to $f_{\text{epSU}}^{n, r_0}$ and then receives (Request, $S$). When receiving $X_C[i]$ from $A$, $\text{Sim}$ randomly selects $t'_i \in \{0, 1\}^{\ell_2}$ and hint, and simulates the execution of $\Pi_{\text{OPPRF}}$. Once receiving $\{t_1, \cdots, t_b\}$ and $\{r_1, \cdots, r_b\}$, the input of $\Pi_{\text{SECRG}}$, from $A$, $\text{Sim}$ simulates the execution of $\Pi_{\text{SECRG}}$. Upon receiving a permutation $\pi$ from $A$, $\text{Sim}$ checks if it is a permutation of $b$ items. If so, $\text{Sim}$ randomly selects $\{s'_1, \cdots, s'_b\}$ where $s'_i \in \{0, 1\}^{\ell_1}$, and simulates the execution of $\mathcal{F}_{\text{PS}}$. After receiving $\{c_1, \cdots, c_b\}$, $\text{Sim}$ sends (Response, OK) to $f_{\text{epSU}}^{n, r_0}$.

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

- $\text{Hyb}_0$: $S$’s view in the real protocol.
- $\text{Hyb}_1$: Same as $\text{Hyb}_0$ except that the output of $\Pi_{\text{OPPRF}}$ is replaced by $(\text{hint}, t_i)$ chosen by $\text{Sim}$, and $\text{Sim}$ runs the $\mathcal{F}_{\text{OPPRF}}$ simulator to produce the simulated view for $S$. The security of protocol $\Pi_{\text{OPPRF}}$ guarantees the view in simulation is computationally indistinguishable from the view in the real protocol.
- $\text{Hyb}_2$: Same as $\text{Hyb}_1$ except that $\text{Sim}$ runs the $\mathcal{F}_{\text{SECRG}}$ simulator to produce the simulated view for $S$. The security of protocol $\Pi_{\text{SECRG}}$ guarantees the view in simulation is computationally indistinguishable from the view in $\text{Hyb}_1$.

$\text{Hyb}_3$: Same as $\text{Hyb}_2$ except that the output of $\mathcal{F}_{\text{PS}}$ is replaced by $\{s'_1, \cdots, s'_b\}$ chosen by $\text{Sim}$, and $\text{Sim}$ runs the $\mathcal{F}_{\text{PS}}$
simulator to produce the simulated view for $\mathcal{S}$. The security of protocol $\Pi_{PS}$ 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 $\{c_1, \ldots, c_b\}$, $\mathcal{R}$$_{PSU}$ receives $(\text{Response, OK})$ 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 $\{c_1, \ldots, c_b\}$ 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 $\mathcal{R}$. Then receives $(\text{Request, } \mathcal{R})$, when receiving $\{(Y_s[i][1], t'_i), \ldots, (Y_s[p][1], t'_i)\}$ from $\mathcal{A}$, Sim randomly selects $(k_i, \text{hint})$, and simulates the execution of $\Pi_{OPPRF}$. Once receiving $\{t'_1, \ldots, t'_b\}$, the input of $\Pi_{ECCRG}$ from $\mathcal{A}$, Sim randomly selects $\{r'_1, \ldots, r'_b\}$ where $r'_i \in \{0, 1\}$ and simulates the execution of $\Pi_{ECCRG}$. Upon receiving $\{r'_1, \ldots, r'_b\}$ from $\mathcal{A}$, Sim randomly selects $\{s'_1, \ldots, s'_b\}$ where $s'_i \in \{0, 1\}$ and simulates the execution of $\Pi_{PS}$. Sim sends $(\text{Response, OK})$ to $\mathcal{R}$. After receiving $Z = X \cup Y$ from $\mathcal{R}$$_{PSU}$, Sim calculates $X' = X \setminus Y$ and randomly selects a subset $S'$ of $\{s'_1, \ldots, s'_b\}$, where $|S'| = |X' \setminus Y|$. For each item $X'_i \in S'$, Sim sets $c_i = X'_i \oplus s'_i$. Finally, Sim randomly chooses $c_i \in \{0, 1\}$ for $i \in \{1, 1 + |S'| + 2, \ldots, b\}$ and sends $\{c_1, \ldots, c_b\}$ 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 $\Pi_{OPPRF}$ is replaced by $(\text{hint}, k_i)$ chosen by Sim, and $\mathcal{A}$ runs the $\mathcal{F}_{\Pi_{OPPRF}}$ simulator to produce the simulated view for $\mathcal{R}$. The security of protocol $\Pi_{OPPRF}$ 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 $\Pi_{ECCRG}$ is replaced by the $\{r'_1, \ldots, r'_b\}$ chosen by Sim, and $\mathcal{A}$ runs the $\mathcal{F}_{\Pi_{ECCRG}}$ simulator to produce the simulated view for $\mathcal{S}$. The security of protocol $\Pi_{ECCRG}$ guarantees the view in simulation is computationally indistinguishable from the view in Hyb$_2$.

Hyb$_3$: Same as Hyb$_2$ except that the output of $\Pi_{PS}$ is replaced by $\{s'_1, \ldots, s'_b\}$ chosen by Sim, and $\mathcal{A}$ runs the $\mathcal{F}_{\Pi_{PS}}$ simulator to produce the simulated view for $\mathcal{S}$. The corresponding $\{c_1, \ldots, c_b\}$ are also changed according to $\{s'_1, \ldots, s'_b\}$. The security of protocol $\Pi_{PS}$ and the random permutation $\pi$ 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 $\{s'_1, \ldots, s'_b\}$, $\mathcal{R}$$_{PSU}$ receives $(\text{Response, OK})$ 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 $\Pi_{ECCRG}$

<table>
<thead>
<tr>
<th>Component</th>
<th>$\Pi_{OPPRF}$</th>
<th>$\Pi_{ECCRG}$</th>
<th>$\Pi_{PS}$</th>
<th>Ciphertexts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp.</td>
<td>$O(</td>
<td>\Pi_2 + \text{en}_1</td>
<td>$</td>
<td>$O(\text{en}_1)$</td>
</tr>
<tr>
<td>Comm.</td>
<td>$O(</td>
<td>\Pi_2</td>
<td>$</td>
<td>$O(\text{en}_1)$</td>
</tr>
</tbody>
</table>

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 $\gamma$ hash functions and $\text{en}_1$ bins.

### 5.4 Cost Analysis

We describe the theoretical complexities of our new construction $\Pi_{ECCRG}$ 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 $\Pi_{ECCRG}$ (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 $\gamma$ hash functions, and the number of bins is $\text{en}_1$. After insertion, the Cuckoo hash table contains $\text{en}_1$ items, and the simple hash table includes $\gamma n_2$ items. Then, the two parties proceed with the subsequent steps on the two tables, which involve performing $\Pi_{OPPRF}$, $\Pi_{ECCRG}$, $\Pi_{PS}$, 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 $\Pi_{OPPRF}$. 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., $O(|\mathcal{E}_2|)$) and the sender additionally needs $O(\text{en}_1)$ to compute the PRF values. As in Figure 16, our $\Pi_{ECCRG}$ consists of $\Pi_{PET}$ and $\Pi_{PS}$, both exhibiting linear complexity; thus the computation and communication costs of $\Pi_{ECCRG}$ are $O(\text{en}_1)$. We use the construction in [23] to implement $\Pi_{PS}$, leading to $O(\text{en}_1 \log(\text{en}_1))$ costs. Finally, the sender needs to compute and send a ciphertext for each item in the Cuckoo hash table, and the costs are $O(\text{en}_1)$. Overall, the computation and communication complexity of our protocol $\Pi_{ECCRG}$ is $O(\text{en}_1 \log(\text{en}_1) + n_2)$, making $\Pi_{ECCRG}$ 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 $\Pi_{ECCRG}$ and compare with the previous works.

**Benchmarking Environment.** We implement our protocol $\Pi_{ECCRG}$ 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 $\Pi_{ECCRG}$ in two network settings, LAN network with 10Gbps bandwidth and 0.02 ms RTT and WAN network with 100Mbps and 0.02ms RTT, which are emulated using Linux tc command. We leverage the constructions in [6] to implement $\Pi_{OPPRF}$ and $\Pi_{PET}$ (the building block of $\Pi_{ECCRG}$). We implement $\Pi_{PS}$ using the
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 \( \varepsilon \) 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] can achieve the enhanced PSU functionality, i.e., \( F_{ePSU}^{\Pi_{ePSU}} \). 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 protocol \( \Pi_{ePSU}^{\text{bECRG}} \) with that of the scheme in [10] in Table 2. We can see that, for \( n_1 = n_2 = 2^8 \), our total runtime is 873.74× 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× less than theirs.

Currently, the PSU by Zhang et al. [31] is the state-of-the-art work for large balanced sets, but it cannot achieve the enhanced PSU functionality \( F_{ePSU}^{\Pi_{ePSU}} \). Zhang et al. [31] gave two designs: (i) SKE-PSU, using symmetric key techniques, and (ii) PKE-PSU\(^\dagger\), 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 protocol \( \Pi_{ePSU}^{\text{bECRG}} \) with that of SKE-PSU in [31] in Table 2. It shows that the performance of our protocol \( \Pi_{ePSU}^{\text{bECRG}} \) 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 \( F_{ePSU}^{\Pi_{ePSU}} \), whereas our protocol \( \Pi_{ePSU}^{\text{bECRG}} \) 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 \( F_{ePSU}^{\Pi_{ePSU}} \). 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 \( n_1 = n_2 = 2^{18} \), 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× higher than ours.

### 6 Proofs for Previous Works

In this section, we formally show that the OT-based PSU protocols cannot UC-realize \( F_{ePSU}^{\Pi_{ePSU}} \), 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 \( F_{bPSU}^{\text{bECRG}} \) to capture its security.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>( 2^8 )</th>
<th>( 2^{10} )</th>
<th>( 2^{12} )</th>
<th>( 2^{14} )</th>
<th>( 2^{16} )</th>
<th>( 2^{18} )</th>
<th>( 2^{20} )</th>
<th>( 2^{22} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAN</td>
<td>[10]</td>
<td>11.78</td>
<td>44.73</td>
<td>175.7</td>
<td>702.4</td>
<td>2836.5</td>
<td>11341.2</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>[31]</td>
<td>-</td>
<td>-</td>
<td>0.68</td>
<td>2.70</td>
<td>10.82</td>
<td>44.78</td>
<td>-</td>
</tr>
<tr>
<td>( \Pi_{ePSU}^{\text{bECRG}} )</td>
<td>Total w/o setup</td>
<td>1.6</td>
<td>1.64</td>
<td>1.77</td>
<td>2.23</td>
<td>4.61</td>
<td>12.98</td>
<td>49.38</td>
</tr>
<tr>
<td></td>
<td>( \Pi_{ePSU}^{\text{bECRG}} )</td>
<td>1.04</td>
<td>1.08</td>
<td>1.2</td>
<td>1.61</td>
<td>3.73</td>
<td>11.12</td>
<td>43.49</td>
</tr>
<tr>
<td>WAN</td>
<td>[10]</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>[31]</td>
<td>-</td>
<td>-</td>
<td>12.87</td>
<td>16.04</td>
<td>28.58</td>
<td>86.31</td>
<td>-</td>
</tr>
<tr>
<td>( \Pi_{ePSU}^{\text{bECRG}} )</td>
<td>Total w/o setup</td>
<td>5.31</td>
<td>5.86</td>
<td>7.06</td>
<td>10.22</td>
<td>21.07</td>
<td>77.56</td>
<td>225.32</td>
</tr>
<tr>
<td></td>
<td>( \Pi_{ePSU}^{\text{bECRG}} )</td>
<td>2.92</td>
<td>3.47</td>
<td>4.62</td>
<td>7.51</td>
<td>16.59</td>
<td>71.19</td>
<td>200.32</td>
</tr>
<tr>
<td>Comm.(MB)</td>
<td>[10]</td>
<td>2.83</td>
<td>11.32</td>
<td>45.28</td>
<td>181.12</td>
<td>724.49</td>
<td>2897.91</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>[31]</td>
<td>-</td>
<td>-</td>
<td>6.52</td>
<td>26.03</td>
<td>103.85</td>
<td>414.43</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Comparisons of total runtime (in seconds) and communication (in MB) between \( \Pi_{ePSU}^{\text{bECRG}} \) 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 \( \Pi_{ePSU}^{\text{bECRG}} \) is implemented in C++, using a single thread. Setup in our protocol refers to base OTs and generating triples for PET. Our \( \Pi_{ePSU}^{\text{bECRG}} \) and \[10\] can achieve the enhanced functionality \( F_{ePSU}^{\Pi_{ePSU}} \), while SKE-PSU [31] can not.

\(^\dagger\)Zhang et al. [31] used PKE-PSU\(^\dagger\) to represent the version that does not perform point compression.
We unify the OT-based protocols in [1, 8, 13, 21, 29, 31] into Theorem 6.1. The protocol following the framework in Figure 11) in the \( \Pi_{\text{OT}} \)-hybrid model, against static, semi-honest adversaries.

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

Construction of environment \( E \). The environment \( E \) chooses sets \( X \) and \( Y \) as the inputs of the sender and the receiver, respectively. Since the environment \( E \) knows both \( X \) and \( Y \), \( E \) of course knows the size of \( X \cap Y \), denoted as \( m \). In other words, the environment \( E \) knows the number of 1’s in \( \{b_1, \ldots, b_n\} \) is \( m \). The environment \( E \) instructs the dummy adversary \( A \) 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 \( \{b_1, \ldots, b_n\} \) reported by the dummy adversary \( A \) is \( m \) and the message \text{Finished} has not been reported by the honest sender \( S \) at the time \( t \), the environment \( E \) outputs 1, otherwise, outputs 0.

The real world execution. In the real world, the \( \Pi_{\text{OT}} \) sub-protocol instances will not be executed until the execution of \( \Pi_{\text{RPMT}} \) sub-protocol instance is finished. Therefore, there is a time \( t \) when the receiver obtains \( \{b_1, \ldots, b_n\} \) 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 \( A \), the bit set \( \{b_1, \ldots, b_n\} \) must be reported to the environment at the time \( t \). Note also that, the protocol execution has not been finished, the honest sender \( S \) is not supposed to return the message Finished to the environment \( E \) at the time \( t \).

The ideal world execution. In the ideal world, since the (dummy) receiver is corrupted, the simulator \( \hat{S} \) is allowed to access the ideal state \( \text{state}_{\text{ePSU}} = \langle X, \text{Finished} \rangle \). After receiving from the functionality \( \mathcal{F}^{ePSU}_{\text{RPMT}} \) the message \( \langle \text{Request}, R \rangle \), to simulate \( \{b_1^{\text{ideal}}, \ldots, b_n^{\text{ideal}}\} \), the simulator \( \hat{S} \) must face the following two simulation strategies:

- **Do send** \( \langle \text{Response}, \text{OK} \rangle \) to the functionality \( \mathcal{F}^{ePSU}_{\text{ePSU}} \) immediately. Note that, now the functionality will update the ideal states into \( \text{state}_{\text{ePSU}} = \langle X, \text{Finished} \rangle \) and \( \text{state}_{\text{g}} = \langle X, Y, \text{Finished} \rangle \), and immediately report \text{Finished} to the environment \( E \).

- **Do not send** \( \langle \text{Response}, \text{OK} \rangle \) to the functionality \( \mathcal{F}^{ePSU}_{\text{ePSU}} \) immediately. Note that, now the functionality will not update the ideal states into \( \text{state}_{\text{ePSU}} = \langle X, \text{Finished} \rangle \) and \( \text{state}_{\text{g}} = \langle X, Y, \text{Finished} \rangle \); of course, no output Finished will be reported to \( E \) 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 \text{Finished} will be reported while there is Finished in the ideal world. If the simulator follows the second simulation strategy, the probability that \( E \) outputs 1 in the real world is 1 except negligible probability. However, in the ideal world, the simulator \( \hat{S} \) does not know \( m \). The probability that there are \( m \) 1’s in \( \{b_1^{\text{ideal}}, \ldots, b_n^{\text{ideal}}\} \) is \( 1/n \), which is far less than 1 when \( n \) is large enough. Thus, the probability that \( E \) outputs 1 in the ideal world is far less than 1. Therefore, \( E \) 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 \( \mathcal{F}^{b}_{\text{PSU}} \) 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 in Figure 18 UC-realizes the functionality \( \mathcal{F}^{b}_{\text{PSU}} \) in the \( \mathcal{F}_{\text{RPMT}}, \mathcal{F}_{\text{OT}} \)-hybrid model, against static, semi-honest adversaries.
Corrupted Sender: Simulator \( \text{Sim} \) first sends the input set \( X \) to \( \mathcal{F}_{\text{PSU}}^b \). After receiving \( \langle \text{Request}_l, Y \rangle \) from \( \mathcal{F}_{\text{PSU}}^b \), \( \text{Sim} \) first invokes \( \text{Sim}' \) to simulate the execution of the sub-protocol \( \Pi_{\text{RPMT}}^b \). Then, \( \text{Sim} \) sends \( \langle \text{Response}_i, \text{OK} \rangle \) to \( \mathcal{F}_{\text{PSU}}^b \). Once receiving \( \langle \text{Response}_e, \text{OK} \rangle \) from \( \mathcal{F}_{\text{PSU}}^b \), \( \text{Sim} \) simulates the execution of sub-protocol \( \Pi_{\text{OT}} \) by invoking \( \text{Sim}' \). When \( \mathcal{A} \) sends items in all \( \Pi_{\text{OT}} \) instances, \( \text{Sim} \) sends \( \langle \text{Response}_e, \text{OK} \rangle \) to \( \mathcal{F}_{\text{PSU}}^b \) and then obtains \( \langle \text{Finished} \rangle \) from the sender’s state \( S \). Compared to the simulator for corrupted sender in [21], [19], [8], [29] or [13], \( \text{Sim} \) 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, \( \text{Sim} \) can simulate \( \mathcal{A} \)’s view such that \( \mathcal{E} \) cannot distinguish the two worlds.

**Corrupted Receiver:** Likewise, simulator \( \text{Sim} \) first sends the input set \( Y \) to \( \mathcal{F}_{\text{PSU}}^b \). After receiving \( \langle \text{Request}_l, Y \rangle \) from \( \mathcal{F}_{\text{PSU}}^b \), \( \text{Sim} \) first invokes \( \text{Sim}' \) to simulate the execution of the sub-protocol \( \Pi_{\text{RPMT}} \) except for the last step. To simulate the last step of \( \Pi_{\text{RPMT}} \), \( \text{Sim} \) sends \( \langle \text{Response}_l, \text{OK} \rangle \) to the functionality \( \mathcal{F}_{\text{PSU}}^b \) and then obtains \( \{b_1, b_2, \cdots, b_{n_1}\} \). Given \( \{b_1, b_2, \cdots, b_{n_1}\} \), \( \text{Sim} \) can invoke \( \text{Sim}' \) to simulate the last step of \( \Pi_{\text{RPMT}} \). Once receiving \( \langle \text{Request}_e, \text{OK} \rangle \) from \( \mathcal{F}_{\text{PSU}}^b \), \( \text{Sim} \) invokes \( \text{Sim}' \) to simulate the execution of the sub-protocol \( \Pi_{\text{OT}} \) except for the last step (i.e., sending items). Then, \( \text{Sim} \) sends \( \langle \text{Response}_e, \text{OK} \rangle \) to \( \mathcal{F}_{\text{PSU}}^b \) and then obtains set \( Z \). By using the items in set \( X \setminus Y \), \( \text{Sim} \) can simulate the last step of each \( \Pi_{\text{OT}} \) instance. Compared to the simulator for corrupted receiver in [1, 8, 13, 19, 21, 29, 31], \( \text{Sim} \) 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 \( Z \) at the same time in the real and ideal worlds. Therefore, \( \text{Sim} \) 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 \( \mathcal{F}_{\text{PSU}}^b \) before simulating \( \{c_1, \cdots, c_{n_1}\} \). Intuitively, the AHE-based protocols can UC-realize \( \mathcal{F}_{\text{PSU}}^b \). The formal security is stated in Theorem 6.3.

**Theorem 6.3.** Given an IND-CPA secure AHE scheme, the protocol following the framework \( \Pi_{\text{Unified AHE}} \) in Figure 20 UC-realizes the functionality \( \mathcal{F}_{\text{PSU}}^b \) (as in Figure 11), against static, semi-honest adversaries.

**Proof.** We will show that for any adversary \( \mathcal{A} \), we can construct a simulator \( \text{Sim} \) that can 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 \( \Pi_{\text{Unified}} \) from the sub-protocols \( \Pi_{\text{RPMT}}^b \) and \( \Pi_{\text{OT}} \), the simulator \( \text{Sim} \) can be constructed by invoking the simulator \( \text{Sim}' \) in [1, 8, 13, 19, 21, 29, 31].
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 \( f_{\text{PSU}}^{n_1,n_2} \) cannot capture the security without during-execution leakage. Therefore, we define a new enhanced functionality \( f_{\text{ePSU}}^{n_1,n_2} \) 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 \( \Pi_{\text{BECRG}} \) that is the first scalable PSU protocol to UC-realize the enhanced PSU functionality \( f_{\text{ePSU}}^{n_1,n_2} \) in the semi-honest setting, by using a new building block \( \Pi_{\text{BECRG}} \). Like OT-based PSU protocols, our \( \Pi_{\text{BECRG}} \) 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 Program 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


[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.


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 $\rho_1$ and $\rho_2$, respectively. Then they perform the $(\rho_1, \rho_2)$-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 $(\rho_1, \rho_2)$-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].](https://example.com/bucketing.png)

For ease of exposition, we take the 4th $(\rho_1, \rho_2)$-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:

- Case1: all the real items that are mapped to the sender’s bin (say \( x_5 \) in Figure 21) belong to \{y_5, y_7\};
- Case2: no real items are mapped to the sender’s bin (i.e., all items are special item \( \perp \)).

The probabilities that Case1 and Case2 occur are denoted as $\Pr[\text{Case1}]$ and $\Pr[\text{Case2}]$, respectively. Clearly, if the receiver is able to determine that Case1 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 $\Pr[\text{Case2}]$. Note that $\Pr[\text{Case1}] = 1 - \Pr[\text{Case2}]$, and the probability $\Pr[\text{Case2}]$ is very small. For example, when the set size is $n = 2^{20}$, $\Pr[\text{Case2}] = 5.778 \times 10^{-8}$. 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.