Shuffle-based Private Set Union: Faster and More Secure

Yanxue Jia†, Shi-Feng Sun†*, Hong-Sheng Zhou†*, Jiajun Du†, and Dawu Gu†*

†Shanghai Jiao Tong University, China
‡Virginia Commonwealth University, USA

Abstract

Private Set Union (PSU) allows two players, the sender and the receiver, to compute the union of their input datasets without revealing any more information than the result. While it has found numerous applications in practice, not much research has been carried out so far, especially for large datasets.

In this work, we take shuffling technique as a key to design PSU protocols for the first time. By shuffling receiver’s set, we put forward the first protocol, denoted as \( \Pi_{PSU}^1 \), that eliminates the expensive operations in previous works, such as additive homomorphic encryption and repeated operations on the receiver’s set. It outperforms the state-of-the-art design by Kolesnikov et al. (ASIACRYPT 2019) in both efficiency and security; the unnecessary leakage in Kolesnikov et al.‘s design, can be avoided in our design.

We further extend our investigation to the application scenarios in which both players may hold unbalanced input datasets. We propose our second protocol \( \Pi_{PSU}^2 \), by shuffling the sender’s dataset. This design can be viewed as a dual version of our first protocol, and it is suitable in the cases where the sender’s input size is much smaller than the receiver’s.

Finally, we implement our protocols \( \Pi_{PSU}^1 \) and \( \Pi_{PSU}^2 \) in C++ on big datasets, and perform a comprehensive evaluation in terms of both scalability and parallelizability. The results demonstrate that our design can obtain a 4-5× improvement over the state-of-the-art by Kolesnikov et al. with a single thread in WAN/LAN settings.

1 Introduction

Private set operations allow mutually distrustful parties to perform set operations (like intersection and union) on their datasets, while revealing no more information about their own private input than what can be deduced from the results. Over the past decade, much progress has been made on Private Set Intersection (PSI), which has become considerably efficient and been deployed widely in practice [6, 18, 27–29, 31, 32, 34]. In contrast, little attention has been drawn on Private Set Union (PSU).

Like the well-researched PSI, PSU also has numerous applications [3, 4, 12, 19]. For example, it can be used for cyber risk assessment and management. Specifically, “Individual blacklists today suffer from several drawbacks that limit their accuracy in malicious source identification. ... Aggregating blacklists from different maintainers and across various attack types can improve the accuracy of malicious source identification over any individual blacklist,” as pointed out by Ramanathan et al. [37]. Therefore, it is significant for organizations (namely, maintainers of IP blacklists) to compute the joint list of individual IP blacklists, which will help minimize vulnerabilities in their infrastructure. In addition, each individual IP blacklist is generated based on a detection strategy developed by the maintainer, which cannot be leaked; note that certain attacks could be launched by the adversaries via evading the detection strategy.

A straightforward way to obtain the joint list is to let the organizations simply exchange their blacklists. However, this will reveal the intersection of their blacklists. Then a curious organization may deduce the detection strategy of the other organization according to the IP addresses in the intersection. Therefore, to mitigate the privacy concerns, what we essentially need is to generate the joint blacklist with the intersection hidden, which is exactly the functionality of PSU.

According to [1, 37], most blacklists in real scenarios contain 1,000 - 500,000 IP addresses, and the update frequency is expected to be 5 - 15 minutes. With our new PSU protocol \( \Pi_{PSU}^2 \) (cf. the high level ideas in Section 1.1), a joint

---

* Corresponding authors.
blacklist can be obtained from individual blacklists of size \(2^{20}\) in 67.756s (cf. our experiments in Section 4) with a single thread in the WAN setting, which is efficient enough for this use case.

In addition, as mentioned in [17,19], PSU can be employed to compute the union of cancer patients of different hospitals while hiding the identities of the patients who had cancer treatment at multiple hospitals, which involves patient privacy. Also, it can be used for privacy-preserving aggregation of network traffic statistics [4], merger of two Internet providers without revealing the information of their existing networks [3], and private database supporting full join [19]. Therefore, it is highly desirable to develop efficient, scalable, and secure PSU protocols.

Many interesting variants of PSU (e.g., multi-party, set-size hidden, and shared-output PSU) have been proposed for real-world applications. In this paper, we focus on PSU in the two-party setting, where a sender and a receiver hold sets \(X\) and \(Y\) respectively, and aim to compute their union \(X \cup Y\) without revealing any more information than the result (especially what are the items in \(X \cap Y\)). Particularly, the goal is to enable the receiver to learn no more information than \(X \cup Y\) and the sender to learn nothing (see Section 2.1 for the formal definition).

**PSU for balanced datasets.** The state-of-the-art work is by Kolesnikov et al. [19]; there, they proposed the first scalable two-party PSU protocol based on symmetric-key techniques. Before that, except for the circuit-based PSU [2], all designs [7, 9, 17] rely on public-key operations like additive homomorphic encryption (AHE), which make the constructed protocols unscaleable in practice, especially for large datasets. To develop efficient PSU protocols, Kolesnikov et al. introduced the notion of Reverse Private Membership Test (RPMT) functionality as a basic building block.

**A unified view for PSU design with single sender’s item.** Interestingly, when focusing on a special case of two-party PSU where the sender’s data-size is 1, we observe that, the designs [7,9,19] can be presented in the same framework\(^3\). Please see Figure 1: the sender has only one item \(x\) and the receiver holds a set \(Y = \{y_1, \cdots, y_n\}\), and we denote this special case of PSU by \((1,n)\)-PSU. We note that, protocols in this design framework, consist of two phases: (1) the two parties execute a RPMT functionality, through which the receiver obtains a bit \(b\), where \(b = 0\) means \(x \notin Y\), otherwise \(x \in Y\). The receiver knows no more information about \(x\) beyond whether \(x\) belongs to \(Y\), and the sender learns nothing about \(Y\). (2) if \(b = 0\) (which means \(x \notin Y\)), the receiver obtains \(x\) (and thus \(\{x\} \cup Y\)) while the sender learns nothing.

Under this framework, in [7,9] the RPMT functionality has been realized by employing homomorphic encryption, and in this case the receiver is able to obtain \(x\) by a straightforward
decryption. As pointed out in [19], however, the public-key operations have become the workhorse of these works. Therefore, Kolesnikov et al. [19] proposed a new way of realizing RPMT in the first phase, based only on symmetric-key operations, and then they implemented the second phase of the design by using Oblivious Transfer (OT).

**PSU design: From single to multiple sender’s items.** Now let’s consider how to extend the special case \((1,n)\)-PSU into a more general \((m,n)\)-PSU, where \(m\) can be a very large integer. Two different approaches, as shown in Figure 2, have been proposed in [7,9] and in [19], respectively.

---

\(^3\)More precisely, [7,9,19] all realize a relaxed RPMT functionality where if \(x \notin Y\), the information about \(x\) can be leaked to the receiver.

---

---

**Figure 1:** Design framework for \((1,n)\)-PSU protocols.

**Figure 2:** Two design frameworks for \((m,n)\)-PSU protocols.

More concretely, in [7,9], a generalized version of RPMT, denoted as \(g\)-RPMT, has been used, to support a large input set \(X = \{x_1, x_2, \cdots, x_m\}\) from the sender. Please refer to Figure 2(a). The receiver needs only to encrypt \(Y\) once, and the ciphertext for \(x_i\) can be used to test if \(x_i \in Y\) for all \(i \in [m]\). However, we note that the schemes in [7,9] suffer from heavy public-key operations.

In contrast, in [19] based on symmetric-key operations, for each \(x_i \in X\) from the sender, a \((1,n)\)-PSU sub-protocol will be executed; to implement \((m,n)\)-PSU, thus, in total, \(m\) number of \((1,n)\)-PSU sub-protocols will be executed. Please refer to Figure 2(b). Here for each \((1,n)\)-PSU sub-protocol, the receiver needs to perform a polynomial interpolation with degree \(n\) for \(Y\), which requires time \(O(n \log^2 n)\). Thus when \(X\)’s size is \(m = n\), this approach will result in a quadratic computation complexity \(O(n^2 \log^2 n)\).

**Bucketing techniques: The gain and the loss.** To further improve the performance, Kolesnikov et al. [19] introduced the bucketing technique in PSU protocol design for the first time. Please see Figure 3: First, the sets \(X\) and \(Y\) are inserted into two simple hash tables with \(b\) bins, respectively, which means that the set \(X\) (resp., \(Y\)) is divided into \(b\) disjoint sub-
sets $X_1, \ldots, X_b$ (resp., $Y_1, \ldots, Y_b$). Then, each bin is padded with dummy items up to the maximum bin size $\rho$. Note that $b = O(n/\log n)$ and $\rho = O(\log n)$ in [19]. The two parties perform a $(\rho, \rho)$-PSU sub-protocol on the items of each bin separately. In this way, the complexity of each sub-protocol on each bin is $O(\rho^2 \log^2 \rho)$, so the total cost can be reduced to $O(b \rho^2 \log^2 \rho) = O(n \log n \log^2 \log n)$.

![Figure 3: Bucketing technique in [19].](image)

However, as mentioned by Kolesnikov et al. [19], the bucketing technique will incur certain information leakage about the items in the intersection, during the execution; intuitively, the receiver could learn if there are items in $X \cap Y$ in certain subsets $Y_i$, where $Y_i \subset Y$. Note that in the ideal PSU, from the view of receiver, any item in the entire set $Y$ could be an item in $X \cap Y$. But here, the receiver can know that some subsets have items in $X \cap Y$ and others do not. To address this issue, Kolesnikov et al. proposed to add special items to each bin, with the goal of reducing the leakage. Unfortunately, we find that their way of adding only special items, is insufficient to resolve the problem (More details are given in Section 5).

Kolesnikov et al.’s bucketing technique [19] is significant since it improves the performance of the PSU design a lot; however, we must note that, the downside of this bucketing technique is that, the designed protocols will suffer from certain level of information leakage. This leakage may not be necessary! Recall that, in prior works [7, 9] under the design framework in Figure 2(a), all items in the receiver’s set $Y$ are processed at the same time; although the involved public-key operations make the design much less scalable (than that in [19]) for large datasets, the resulting protocol does not suffer from the information leakage issue (as that in [19]).

**Main question.** Based on above discussions, we ask the following natural question:

*Is it possible to design a PSU protocol to achieve the “best of the two worlds,” i.e., (1) fast and scalable, and at the same time, (2) without any unnecessary information leakage?*

We will give an affirmative answer to this question. In particular, we propose a practical, scalable two-party PSU protocol named $\Pi_{PSU}^{R}$ under the design framework in Figure 2(a) by shuffling the receiver’s set. Our protocol $\Pi_{PSU}^{R}$ relies only on lightweight symmetric-key primitives (along with some OTs; we note the OTs are also needed in the state-of-the-art result [19]). More details will be shown in Section 1.1 and Section 3.

**PSU for unbalanced datasets.** To the best of our knowledge, previous works on PSU mainly focus on designing efficient protocols for balanced datasets. We now investigate how to design practical PSU protocols in the unbalanced application scenarios, in which the receiver’s input size is significantly larger than the sender’s, or vice versa. In fact, existing protocols (including our $\Pi_{PSU}^{R}$ above) are already very fast when the size of receiver’s input is significantly smaller than the sender’s, as the relatively heavy operations mainly depend on the size of receiver’s set. However, in the case that the size of receiver’s input is much larger than the sender’s, the performance of these protocols is reduced significantly. Hence, the second question we ask is:

*Is it possible to design a fast and scalable PSU protocol when the sender’s input size is much smaller than the receiver’s?*

We answer this question affirmatively by presenting a new protocol named $\Pi_{PSU}^{S}$; this new protocol can be viewed as the dual version of $\Pi_{PSU}^{R}$, exactly by shuffling the sender’s set. More details about protocol $\Pi_{PSU}^{S}$ are given in Section 1.1 and Section 3.
share is to break the mapping by shuffling the receiver’s set (and shares) with a permutation not known to the receiver. Together with the additive secret sharing explained before, what we essentially need is a Permute + Share functionality: taking as input a set \( Y = \{y_1, \ldots, y_n\} \) from the receiver and a permutation \( \pi \) (over \( \{1,2,\cdots ,n\} \)) from the sender, the functionality outputs the shares \( \{s_{\pi(1)}, \ldots, s_{\pi(n)}\} \) and \( \{y_{\pi(1)} \oplus s_{\pi(1)}, \ldots, y_{\pi(n)} \oplus s_{\pi(n)}\} \) to the sender and the receiver, respectively. After executing this functionality, the sender computes \( x_i \oplus y_{\pi(i)} \oplus s_{\pi(i)} \) if \( x_i \in Y \), but she is unable to find out \( y_{\pi(i)} \) according to \( s_{\pi(i)} \), as she does not know \( \pi \).

At this point, it seems that \( x_i \) can be completely hidden from the receiver at the first glance. Unfortunately, \( x_i \oplus y_{\pi(i)} \oplus s_{\pi(i)} \) may still leak partial information about \( x_i \) to the receiver. This is because \( y_{\pi(i)} \oplus s_{\pi(i)} \) is not distributed uniformly and independently from the perspective of the receiver who knows \( s_{\pi(i)} \) and \( y_{\pi(i)} \). To overcome this obstacle, we further employ multi-point Oblivious Pseudorandom Function (OPRF)\(^4\) \( F(k,\cdot) \) to conceal \( \{x \oplus y_{\pi(1)} \oplus s_{\pi(1)}, \ldots, x \oplus y_{\pi(n)} \oplus s_{\pi(n)}\} \).

More concretely, the receiver takes \( \{s_{\pi(1)}, \ldots, s_{\pi(n)}\} \) as the input to multi-point OPRF, then the sender receives the PRF key \( k \) and the receiver obtains \( \{F(k, s_{\pi(1)}), \ldots, F(k, s_{\pi(n)})\} \).

In this case, the sender with the key \( k \) can compute \( I_{\pi} = \{F(k, x_i \oplus y_{\pi(i)} \oplus s_{\pi(i)}), \ldots, F(k, x_i \oplus y_{\pi(n)} \oplus s_{\pi(n)})\} \) for each \( x_i \) and sends it to the receiver. Then the receiver proceeds to perform the membership test as before, but learns nothing about \( x_i \) as she does not know the PRF key \( k \). For the second phase, the receiver can receive \( x_i \notin Y \) through OT as in [19].

**Optimization.** Following our core idea, it can be seen that the protocol executes Permute + Share and multi-point OPRF only once for all \( x_i \in X \), but needs to send \( m \) sets (i.e., \( I_1, \ldots, I_m \)) to receiver. In addition, the sender and receiver need to execute OT sub-protocol \( m \) times. Fortunately, we find that the functionality Permute + Share (resp. multi-point OPRF) can be securely realized by the protocols in [5, 22] (resp. [6]) with computation and communication cost \( O(n \log n) \) (resp. \( O(n) \)). However, the sender needs to compute and send \( I_1 \) containing \( n \) PRF values for each \( x_i \in X \), thus it results in a quadratic computation and communication complexity \( O(mn) \).

The main reason for this quadratic complexity is that the sender does not know which item in \( Y \) may be equal to his item \( x_i \), so he has to XOR \( x_i \) with all shares \( \{y_{\pi(1)} \oplus s_{\pi(1)}, \ldots, y_{\pi(n)} \oplus s_{\pi(n)}\} \). To improve the efficiency, our key idea is to reduce the number of items in \( Y \) that could be equal to \( x_i \) by leveraging Cuckoo hashing [26] (defined in Section 2.2). Briefly, we insert \( Y \) into a Cuckoo hash table with \( \gamma \) hash functions, \( \gamma \cdot n \) bins and stash size 0, and then execute \( \Pi_{\text{PSU}}^R \) over the hash table. To make it clear, we take a concrete example to explain the optimization via Cuckoo hashing, as illustrated in Figure 5. In particular, we assume that the sender’s item to be checked is \( x_1 \), and that the receiver’s set \( Y = \{y_1, \ldots, y_6\} \), can be inserted to the Cuckoo hash table with 8 bins and 2 hash functions \( h_1(\cdot) \) and \( h_2(\cdot) \). Then the optimized protocol works as follows: (1) The receiver inserts \( Y \) into the Cuckoo hash table and adds a dummy item \( d \) to each empty bin, then obtains the filled table denoted by \( Y^* \). (2) The receiver and the sender execute Permute + Share with a randomly chosen permutation \( \pi \) and \( Y^* \) as inputs, and obtain the shuffled secret share sets \( S_{H_1} \) and \( S_{H_2} \), respectively. Here, \( s_1^d \) and \( s_2^d \) in \( S_{H_1} \) are the shares of dummy item and \( s_j^d \neq s_i^d \). The dotted arrows mean that after permutation \( \pi \), the 4-th (resp. 7-th) item in \( S_{H_1} \) is the share of the 2-th (resp. 6-th) item in \( Y^* \), but the receiver does not know the corresponding relations. In addition, the other share is the 4-th (resp. 7-th) item in \( S_{H_2} \). (3) With \( h_1(\cdot) \) and \( h_2(\cdot) \), the sender computes

---

\(^4\)Multi-point OPRF is evaluated on different inputs with the same key, while single-point OPRF is evaluated with a different key for each input.
we summarize the computation and communication costs of Table 1: The computation and communication costs of \(\Pi^{R}_{PSU}\).

<table>
<thead>
<tr>
<th>Costs</th>
<th>(O(n \log n))</th>
<th>(O(n))</th>
<th>(O(m))</th>
<th>(O(m))</th>
</tr>
</thead>
</table>

\(m\) is the sender’s set size; \(n\) is the receiver’s set size.

Protocol \(\Pi^{S}_{PSU}\): shuffling sender’s set. From Table 1 we can see that when \(m \gg n\) the overall cost of \(\Pi^{R}_{PSU}\) is dominated by \(\{I_i\}_{i \in [m]}\) and OT that are linear in \(m\). However, when \(m \ll n\) the cost is dominated by Permute + Share that is superlinear in \(n\). Therefore, when considering unbalanced datasets, \(\Pi^{R}_{PSU}\) is more suitable for the case that the sender’s set size is much larger than the receiver’s (i.e., \(m \gg n\)). To develop efficient solutions for the other case where \(m \ll n\), we propose a second protocol \(\Pi^{S}_{PSU}\) by shuffling the sender’s set. As a whole, it can be regarded as the dual version of \(\Pi^{R}_{PSU}\). The high-level idea is shown in Figure 6.

![Figure 6: Core idea of \(\Pi^{S}_{PSU}\) for \((m, n)\)-PSU.](image)

The basic idea of \(\Pi^{S}_{PSU}\) is to share the sender’s set \(X\) into two share sets obtained by the sender and the receiver, respectively. Then, the sender sends the shares of the items in \(X \setminus Y\) to the receiver such that the receiver can recover the items in \(X \setminus Y\). While being shared, the sender’s set \(X\) needs to be shuffled by a permutation not known by the sender such that the sender cannot know the correspondence between shares and the items in \(X\). \(\Pi^{S}_{PSU}\) can also be optimized via Cuckoo hashing following the similar idea as in \(\Pi^{R}_{PSU}\).

1.2 Our Contributions

We explore new techniques of designing two-party PSU protocols for both balanced and unbalanced datasets, and propose two efficient and secure PSU protocols \(\Pi^{R}_{PSU}\) and \(\Pi^{S}_{PSU}\) supporting big datasets in Section 3. More specifically, our main contributions are summarized as below.

New protocols. We for the first time, give a scalable and secure construction, named \(\Pi^{R}_{PSU}\), for realizing two-party PSU. Note that the state-of-the-art design by Kolesnikov et al. [19] faces the issue of partial information leakage of items in intersection. While this protocol is efficient for balanced datasets, we further extend our study to the unbalanced case in the sense that the receiver’s input size is significantly larger than the sender’s, or vice versa. Then we propose a second efficient and secure protocol, dubbed \(\Pi^{S}_{PSU}\). This protocol is suitable in the applications where the sender’s input size is much smaller than the receiver’s; this can be viewed as a dual version of our first protocol which is more suitable for the opposing case.

New design techniques. To avoid the leakage incurred by the leverager of bucketing technique on the receiver’s set, our key point is to process the receiver’s set at the same time. Then we design \(\Pi^{R}_{PSU}\) under the framework in Figure 2(a) by shuffling the receiver’s set. Regarding designing PSU protocols for unbalanced datasets, our observation is to perform heavy operations on the smaller dataset. Thus we design \(\Pi^{S}_{PSU}\) by shuffling the sender’s set for the case that the sender’s set size is much smaller than the receiver’s. With the key technique shuffling, our design avoids expensive computations like public-key operations and repeated operations on sender/receiver’s set. Furthermore, we reduce the communication and computation overhead by employing the Cuckoo hashing, which is for the first time used in PSU.

Implementation & evaluation. We implement our protocols in C++ and perform a comprehensive evaluation in Section 4. The results demonstrate that \(\Pi^{R}_{PSU}\) is \(4.5 \times\) faster than the state-of-the-art PSU protocol [19] with a single thread in WAN/LAN settings. Moreover, we show that our protocols support parallelization; \(\Pi^{R}_{PSU}\) and \(\Pi^{S}_{PSU}\) can achieve a speedup of \(2.89 \times\) and \(3.49 \times\) respectively at 8 threads in LAN setting. Beyond, the results indicate that our protocols are also...
efficient and scalable for unbalanced datasets.

New leakage analysis. In Section 5 we show the bucketing technique adopted in the state-of-the-art design [19] will leak the information of intersection. Specifically, we demonstrate that after knowing all the sender’s items in a bin belong to the intersection, the receiver can learn that her corresponding bin has items in \( X \cap Y \) with an extremely large probability.

1.3 Related Work

As a special case of secure two-party computation, privacy-preserving set operation also includes generic and custom constructions. The generic PSI protocol (also called circuit-based PSI) was firstly proposed by Huang et al. [13], and the first generic PSU protocol was proposed in [2]. In general, the generic protocols are less efficient than the custom ones but more flexible to support different functionalities. In this work, we are mainly interested in the custom constructions.

Over the past decades, a large amount of work has been done on specific PSI (e.g., [6, 18, 21, 27–29, 31, 32, 34]). The recent works are mainly based on oblivious transfer extension [25, 29, 33, 39] and various OPRF constructions [6, 18, 27, 28, 40]. The state-of-the-art protocols have become considerably efficient for practical applications.

Although PSI and PSU are similar and they share some building blocks (e.g., OPRF [18, 19, 27]), PSU cannot be obtained by directly employing existing PSI techniques, and little process has been made towards practical PSI so far.

The first PSI protocol was proposed by Kissner and Song in [17], and realized by using threshold additively homomorphic encryption (AHE) and polynomial representation. Later, Frikken [9] proposed a new PSI protocol with intersection hidden by leveraging similar techniques. Roughly, their protocol works as follows: the receiver holding the secret key of AHE sends to the sender the encrypted polynomial representation \( Enc(f) \) of her own set \( Y \), then the sender with set \( X \) calculates the tuples \( (Enc(xf(x)), Enc(f(x))) \) for each \( x \in X \), and sends them to the receiver. If \( x \in Y \), the receiver can only recover \((0, 0)\) from the tuple without learning any information about \( x \). Otherwise, the receiver can recover \((xf(x), f(x))\) and then obtain \( x \). Following the similar idea in [9], Davidson and Cid [7] presented a new protocol by replacing polynomial representation with inverted Bloom Filter.

All the above protocols encrypt the (polynomial or Bloom Filter) representation of the receiver’s set using AHE and perform a large number of operations in an encrypted manner. As pointed out by Kolesnikov et al. in [19], the public-key operations have become the workhorse of these works. Then they proposed the first scalable PSU protocol using only symmetric-key techniques. In their work, a polynomial is also used to represent the receiver’s set, but the receiver is required to re-generate her polynomial representation for testing each item of the sender. By this way, the design in [19] avoids the usage of the expensive additive homomorphic encryption, but still suffers from the repeated high-degree polynomial interpolations. To further reduce this cost, Kolesnikov et al. proposed an efficient optimization by using the bucketing technique.

Next, we summarize the asymptotic complexities of the above PSU protocols [7, 9, 17, 19] and ours in Table 2. In terms of asymptotic complexity, the scheme in [7] is the most efficient. However, according to the experimental comparison shown in [19], the protocol in [7] is 7607 \( \times \) slower than [19] due to heavy public-key operations. Note that our protocol \( \Pi^5_{PSU} \) is 4–5 \( \times \) faster than [19]. On the other hand, the PSU protocols in [19] and our work are only based on symmetric-key operations, but the complexities are super-linear. Therefore, designing a PSU protocol with linear complexity by using symmetric-key operations is still left open.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Comm. (bits)</th>
<th>Comp. (#Ops)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>pub-key</td>
<td>symm-key</td>
</tr>
<tr>
<td>[17]</td>
<td>( O(n^\log n) )</td>
<td>-</td>
</tr>
<tr>
<td>[9]</td>
<td>( O(n) )</td>
<td>( O(n^\log n) )</td>
</tr>
<tr>
<td>[7]</td>
<td>( O(n) )</td>
<td>( O(n) )</td>
</tr>
<tr>
<td>[19]</td>
<td>( O(n\log n) )</td>
<td>-</td>
</tr>
<tr>
<td>( \Pi^5_{PSU} )</td>
<td>( O(n\log n) )</td>
<td>( O(n\log n) )</td>
</tr>
</tbody>
</table>

Table 2: Comparisons of asymptotic communication (bits) and computation (operations) costs of two-party PSI protocols in the semi-honest setting. pub/symm-key: public/symmetric-key operations. Here, \( n \) is the size of the parties’ input sets. For [19] and ours, we ignore the pub-key cost of \( \kappa \) base OTs where \( \kappa \) is computational security parameter.

We now provide a concrete comparison for the state-of-the-art results on PSU and on PSI for the large datasets of size \( 2^{20} \) in LAN setting. We can see that the performance of PSI is far better than that of PSI. Our \( \Pi^5_{PSU} \) outperforms Kolesnikov et al. [19] by a factor of 5.4, but is still 20 \( \times \) lower than the PSI protocol in [18]. More research on PSI design should be encouraged to further improve the performance.

<table>
<thead>
<tr>
<th></th>
<th>PSU</th>
<th>PSI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time (s)</td>
<td>263.476</td>
<td>2470.11</td>
</tr>
<tr>
<td>Comm.(MB)</td>
<td>48.703</td>
<td>1338.79</td>
</tr>
</tbody>
</table>

Table 3: Comparisons of total runtime (in seconds) and communication (in MB) between the state-of-the-art works on PSU and PSI for set size \( 2^{20} \) in LAN setting.

Finally, we remark that in a concurrent and independent work, Garimella et al. [10] improve the results in [19] by a factor of 2 – 2.5 \( \times \) also by leveraging shuffling technique. Similar to that in [19], polynomial representation is also used for receiver’s set in [10]; the difference is that, repeated polynomial interpolations can be avoided in [10], leading to better performance than that in [19]. However, we note that in our work the polynomial interpolations can be entirely avoided; thus we can obtain even better performance than that in [10].
2 Preliminaries

Notation. We denote by $\kappa$ and $\lambda$ the computational and statistical security parameters, respectively. We use $[m]$ to denote the set $\{1, 2, \cdots, m\}$, and $X = \{x_1, \cdots, x_n\}$ to denote a set with $|X| = n$. Given a permutation $\pi$ on $n$ items, we use $\pi(X)$ to denote the set $\{x_{\pi(1)}, x_{\pi(2)}, \cdots, x_{\pi(n)}\}$.

2.1 Security Model

Our PSU protocol involves two parties, and we follow the static semi-honest security definition in [20] for secure two-party computation in this work.

Static Semi-Honest Security. There are two parties denoted by $P_0$ and $P_1$. Let $f_i(X, Y)$ be the output for $P_i$ in the ideal functionality $\mathcal{F}$ and $f(X, Y) = (f_0(X, Y), f_1(X, Y))$ be the joint output. Let the view of $P_i$ during an execution of $\Pi$ on inputs $(X, Y)$ be $\text{view}$$_i^{\Pi}(X, Y)$ that consists of the input $X$ or $Y$, the contents of $P_i$’s internal random tape and the messages received during the execution. Similarly, $\text{output}^{\Pi_i}(X, Y)$ is the output of $P_i$ during an execution of $\Pi$ on inputs $(X, Y)$ and can be computed from the $\Pi$’s view. And the joint output of both parties is $\text{output}^{\Pi}(X, Y) = (\text{output}^{\Pi_0}(X, Y), \text{output}^{\Pi_1}(X, Y))$.

Definition 2.1. A protocol $\Pi$ securely computes $\mathcal{F}$ against static semi-honest adversaries if there exist probabilistic polynomial-time (PPT) algorithms $\text{Sim}_0$ and $\text{Sim}_1$ such that

$$(\text{Sim}_0(\Pi_0, f_0(X, Y)), f(X, Y)) \xrightarrow{\text{def}} (\text{view}^{\Pi_0}(X, Y), \text{output}^{\Pi_0}(X, Y)),
(\text{Sim}_1(\Pi_1, f_1(X, Y)), f(X, Y)) \xrightarrow{\text{def}} (\text{view}^{\Pi_1}(X, Y), \text{output}^{\Pi_1}(X, Y)).$$

PSU Functionality. The ideal functionality for PSU, denoted as $\mathcal{F}^{n_1,n_2}_{PSU}$, is shown in Figure 7 (except for the text marked in blue). This functionality allows two players, the sender and the receiver, who hold private datasets with size $n_1$ and $n_2$, respectively, to compute the union of the two input datasets. Note that our formulation of PSU functionality is identical to that in [19], and we allow only the receiver, not the sender, to obtain the union of the two input sets.

We remark that, in this formulation, based on the obtained output, the receiver can easily calculate the size of the intersection of the two input sets. However, the receiver is not allowed to learn any additional information about the data items in the intersection. On the other hand, the sender is not allowed to learn any information about the union or the intersection of the two private input sets.

With the goal of investigating PSU design comprehensive, we further consider a natural relaxation of the ideal PSU functionality, by allowing the sender to learn the size of the intersection; as mentioned above, the receiver by default, is allowed to obtain the intersection size. We denote the relaxed ideal functionality as $\mathcal{F}^{n_1,n_2}_{PSU}$; we also show it in Figure 7 while the difference from $\mathcal{F}^{n_1,n_2}_{PSU}$ is marked in blue.

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

Parameters:
- Set size for sender $S$ is $n_1$, set size for receiver $R$ is $n_2$;
- Maximum length of all elements is $\ell$.

Functionality:
1. Wait for input $X = \{x_1, \cdots, x_{n_1}\}$ from $S$, abort if $|X| \neq n_1$ or $\exists x_i \in X$ such that $|x_i| > \ell$;
2. Wait for input $Y = \{y_1, \cdots, y_{n_2}\}$ from $R$, abort if $|Y| \neq n_2$ or $\exists y_i \in Y$ such that $|y_i| > \ell$;
3. Give output $X \cap Y$ to $S$, and give output $X \cup Y$ to $R$, then $R$ can compute $|X \cup Y| = n_1 + n_2 - |X \cap Y|$.

2.2 Building Blocks

We briefly recollect the main cryptographic tools, including Permuate + Share, multi-point Oblivious PRF, 1-out-of-2 Oblivious Transfer, simple hashing and Cuckoo hashing.

Permuate + Share. The Permuate + Share functionality $\mathcal{F}_{PS}$ is defined by Chase et al. in [5]. There are two parties $P_0$ and $P_1$ in this functionality, where $P_0$ possesses a set $X = \{x_1, \cdots, x_{n_1}\}$ of size $n_1$ and $P_1$ picks a permutation $\pi$ on $n$ elements. The goal of $\mathcal{F}_{PS}$ is to let $P_0$ learn the shares $\{s_{\pi(1)}, s_{\pi(2)}, \cdots, s_{\pi(n)}\}$ and $P_1$ learn nothing but the other shares $\{s_{\pi(1)} + s_{\pi(2)} + s_{\pi(3)} + \cdots, s_{\pi(n)}\}$.

Multi-Point Oblivious PRF. Oblivious PRF (OPRF) is a protocol involving two parties $P_0$ and $P_1$, where $P_0$ obtains the key of the PRF $F(\cdot, \cdot)$ and $P_0$ takes as input $x$ and obtains $F(k, x)$. OPRF has been widely used in PSI protocols, and extensive efforts have been made to develop efficient single-point OPRF protocols [8, 11, 18, 30]. Most recently, Pinkas et al. [27] proposed for the first time to use multi-point OPRF to realize more efficient PSI protocols. Particularly, in a multi-point OPRF, $P_0$ takes as input $\{x_1, \cdots, x_{n_1}\}$ and obtains $\{F(k, x_1), \cdots, F(k, x_{n_1})\}_{n \geq 1}$ while $P_1$ obtains the PRF key $k$. Later, Chase et al. [6] designed a more efficient multi-point OPRF with computation complexity $O(n)$ while the computation cost of [27] is $O(n \log^2 n)$. Moreover, the scheme in [6] only involves efficient OT extension and AES operations, rather than the high-degree polynomial interpolation/evaluation over a large field as in [27]. The functionality $\mathcal{F}_{mpOPRF}$ is shown in Figure 12 in Appendix A.

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.
ing about \( b \). The first OT protocol was proposed by Rabin in [36]. And due to the lower bound in [14], all the OT protocols require expensive public-key operations. To improve the performance, Ishai et al. [15] introduced the concept of OT extension that enables us to carry out many OTs based on a small number of basic OTs. The functionality \( f_{\text{OT}} \) is shown in Figure 13 in Appendix A.

**Simple Hashing.** In the simple hashing scheme, there are \( \gamma \) hash functions \( h_1, \ldots, h_\gamma : \{0, 1\}^* \to \{b\} \) used to map \( n \) items into \( b \) bins \( B_1, \ldots, B_b \). An item \( x \) will be added into \( B_{h_1(x)}, B_{h_2(x)}, \ldots, B_{h_\gamma(x)} \), regardless of whether these bins are empty. According to the following inequality [23], the maximum bin size \( \rho \) can be set to ensure that no bin will contain more than \( \rho \) items except with probability \( 2^{-k} \) when hashing \( n \) items into \( b \) bins.

\[
\Pr[\exists \text{ bin with } \geq \rho \text{ items}] \leq b \left[ \sum_{i=0}^{\rho} \binom{n}{i} \cdot \left( \frac{1}{b} \right)^i \cdot \left( 1 - \frac{1}{b} \right)^{n-i} \right]
\]

**Cuckoo Hashing.** Cuckoo hashing was introduced by Pagh and Rodler in [26]. In this hashing scheme, there are \( \gamma \) hash functions \( h_1, \ldots, h_\gamma \) 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 \( B_{h_1(x)}, B_{h_2(x)}, \ldots, B_{h_\gamma(x)} \). If there are no empty bins in the \( k \) bins, randomly select a bin \( B_{h_i(x)} \) in these \( \gamma \) bins, and evict the prior item \( y \) in \( B_{h_i(x)} \) where \( h_i(x) = h_i(y) \) to a new bin \( B_{h_j(y)} \) where \( i \neq j \). 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 [34], we can adjust the values of \( \gamma \) and \( \varepsilon \) to reduce the stash size to 0 while achieving a hashing failure probability of \( 2^{-40} \).

### 3 Private Set Union via Shuffling

In this section, we propose two scalable PSU protocols \( \Pi_{PSU}^R \) and \( \Pi_{PSU}^S \) by leveraging shuffling and Cuckoo hashing techniques. The first protocol \( \Pi_{PSU}^R \) realizes \( f_{\text{PSU}}^{n_1, n_2} \), which is obtained by shuffling the receiver’s set. In contrast, the second \( \Pi_{PSU}^S \) realizes \( f_{\text{PSU}}^{n_1, n_2} \), which is obtained by shuffling the sender’s set. To ease the understanding of our main idea, we also present the simplified versions of \( \Pi_{PSU}^R \) and \( \Pi_{PSU}^S \) (without using the Cuckoo hashing) denoted by \( \Pi_{PSU}^R \) and \( \Pi_{PSU}^S \), respectively, in the full version [16].

#### 3.1 Protocol \( \Pi_{PSU}^R \): Shuffling Receiver’s Set

The first protocol \( \Pi_{PSU}^R \) is designed under the framework in Figure 2(a). Our basic idea is to realize the functionality \( g \)-RPM by shuffling the secret shares of the receiver’s set. In the following, we first give a brief description of this protocol and then present the details in Figure 8.

We assume that the sender’s set is \( X = \{x_1, \ldots, x_{n_1}\} \) and the receiver’s set is \( Y = \{y_1, \ldots, y_{n_2}\} \). Then the protocol proceeds as follows. Firstly, the receiver chooses the parameters of Cuckoo hash table without a stash, including the number of bins \( b = \varepsilon \cdot n_2 \) and \( \gamma \) hash functions \( h_1, \ldots, h_\gamma \). Then she inserts \( Y \) into this table and pads each empty bin with a dummy item \( d \). Please refer to Section 2.2 for the details of Cuckoo hashing. After successfully inserting \( Y \) into the Cuckoo hash table, the receiver sends the parameters to the sender. Hereafter, we denote by \( Y_C \) the Cuckoo hash table filled with \( Y \) and \( Y_C[i] \) the item in the \( i \)-th bin of the table.

Secondly, the two parties invoke \( f_{\text{PSU}} \) with inputs \( Y_C \) and \( \pi \) randomly chosen by the receiver. After this, the sender and receiver obtain the shares \( \{a'_1, a'_2, \ldots, a'_b\} \) and \( \{a_1, a_2, \ldots, a_b\} \) respectively, where \( a_i \oplus a'_i = Y_C[\pi(i)] \). Further through \( f_{\text{mopPRF}} \) with \( \{a_i\}_{i \in [b]} \) as input, the sender receives PRF key \( k \) and the receiver obtains \( \{F(k, a_i)\}_{i \in [b]} \) where \( F(k, a_i) \in \{0, 1\}^2 \).

Next, for each \( x_i \in X \), the sender generates a set \( I_i \) so that the receiver can test if \( x_i \in Y \) via \( I_i \). If not, the receiver can obtain the item \( x_i \). However, we observe that if the sender picks items from \( X \) in a special order, then when the receiver obtains a certain item \( x_i \) she can obtain extra information about \( X \) according to the order. To avoid this leakage, the sender permutes his set \( X \) to \( \pi(X) = \{x'_1, \ldots, x'_{n_1}\} \) by a randomly chosen permutation \( \pi \), and then generates \( I_i \) for each item \( x'_i \) of \( \pi(X) \) in turn, the details of which are shown below.

Note that for each item \( x'_i \in \pi(X) \), if there is an item \( y \in Y \) equal to \( x'_i \), then \( y \) must be inserted into one of the positions of \( Y_C \) indexed by \( \{h_j(x'_i)\}_{j \in [\gamma]} \), according to the property of the Cuckoo hashing. Hence, to test if \( x'_i \in Y \), we need only to check if \( x'_i \in \{Y_C[h_j(x'_i)] \} \), \( j \in [\gamma] \). To do so, the sender first uses the permutation \( \pi \) of \( Y_C \) to identify the shares of \( Y_C[h_j(x'_i)] \) from \( \{a'_1, a'_2, \ldots, a'_b\} \), say \( \{a'_{q_1}, a'_{q_2}, \ldots, a'_{q_{\gamma}}\} \), where \( q_j = \pi^{-1}(h_j(x'_i)) \) for \( j \in [\gamma] \). Then he computes \( I_i \) as \( I_i = \{F(k, x'_i \oplus a'_{q_1}), \ldots, F(k, x'_i \oplus a'_{q_{\gamma}})\} \).

However, we notice that if there are distinct hash functions, say \( h_j \) and \( h_k \) s.t. \( h_j(x'_i) = h_k(x'_i) \), then we have \( I_i[j_i] = I_i[j_i] \), from which the receiver may learn partial information about \( x'_i \). To overcome this shortcoming, \( I_i \) is generated in a slightly different way: \( I_i[j_i] \) is computed and recorded in \( I_i \) as before, but \( I_i[j_i] \) is replaced with a random value \( r \) \( \overset{\$}{\leftarrow} \{0, 1\}^2 \). In particular, for each \( x'_i \), the sender initializes a set \( I_i = \emptyset \) and a set \( Q_i = \emptyset \), where \( Q_i \) is used to record the indices of the shares (in \( \{a'_1, \ldots, a'_b\} \) that are XORed with \( x'_i \). Then for each \( j \in [\gamma] \), the sender computes \( q_j = \pi^{-1}(h_j(x'_i)) \). If it does not appear before (i.e., \( q_j \notin Q_i \)), the sender adds it into \( Q_i \) and records \( F(k, x'_i \oplus a'_{q_j}) \) into \( I_i \). Otherwise, the sender randomly chooses \( r \) for example, assuming that \( X \) consists of the ages of a group of people, if the sender picks the items of \( X \) in an ascending order and the receiver obtains an item \( x = 16 \) through the third OT, then the receiver can learn that two people in \( X \) are under the age of 16.
Protocol $\Pi_{\text{PSU}}$ using Cuckoo hashing

Parameters:
- Hash functions $h_1, \ldots, h_t : \{0,1\}^{f_i} \rightarrow [b]$;
- A Cuckoo hash table based on $h_1, \ldots, h_t$ with $b = e \cdot n_2$ bins, stash size $s = 0$;
- Ideal functionalities $\mathcal{F}_{\text{PSU}}, \mathcal{F}_{\text{OT}}$ and $\mathcal{F}_{\text{mpOPRF}}$ (the underlying PRF is $F'(k, \cdot) : \{0,1\}^{f_i} \rightarrow \{0,1\}^{f_i}$);

Inputs:
- Sender $S$: set $X = \{x_1, \ldots, x_{n_1}\}, x_i \in \{0,1\}^{f_i}$;
- Receiver $R$: set $Y = \{y_1, \ldots, y_{n_1}\}, y_i \in \{0,1\}^{f_i}$;

Protocol:
1. $R$ inserts set $Y$ into the Cuckoo hash table based on $h_1, \ldots, h_t$ as shown in Section 2.2, and adds a dummy item $d$ in each empty bin, then the filled Cuckoo hash table as $Y_C$ and the item in $i$-th bin as $Y_C[i]$;
2. $S$ and $R$ invoke the ideal functionality $\mathcal{F}_{\text{PSU}}$:
   - $R$ acts as $P_0$ with input set $Y_C$, and $S$ acts as $P_1$ with a permutation $\pi$;
   - $R$ obtains the shuffled shares $\{a_1, a_2, \ldots, a_b\}$, and $S$ obtains the other shuffled shares $\{a'_1, a'_2, \ldots, a'_b\}$ where $Y_C[(\pi(i))] = a'_i + a_i$;
3. $S$ and $R$ invoke the ideal functionality $\mathcal{F}_{\text{mpOPRF}}$:
   - $R$ acts as $P_0$ with her shuffled shares $\{a_i\}_{i \in [b]}$, and obtains the outputs $\{F(k, a_i)\}_{i \in [b]}$;
   - $S$ obtains the key $k$;
4. $R$ initializes set $Z = \emptyset$, $S$ randomly selects a permutation $\pi'$, and obtains $\pi'(X) = \{x'_1, x'_2, \ldots, x'_{n_1}\}$;
5. For $i \in [n_1]$:
   - $S$ initializes sets $Q_i = \emptyset$ and $I_i = \emptyset$;
   - For $j \in [y]$:
     - $S$ computes $q_j = \pi^{-1}(h_j(x'_j))$;
     - If $q_j \notin Q_j$, $Q_j = Q_j \cup \{q_j\}$, $I_i = I_i \cup \{F(k, x'_i + a'_q)\}$, else, $r \leftarrow \{0,1\}^{f_i}$ and $I_i = I_i \cup \{r\}$;
   - $S$ sends $I_i$ to $R$;
   - $R$ checks if $\{F(k, a_j)\}_{j \in [y]} \cap I_i \neq \emptyset$. If so, $R$ sets $b_i = 1$, otherwise, sets $b_i = 0$;
   - $S$ and $R$ invoke the ideal functionality $\mathcal{F}_{\text{OT}}$:
     - $S$ acts as $P_0$ with input $\{x'_i, \bot\}$;
     - $R$ acts as $P_1$ with input $b$;
     - If $b_i = 0$, $R$ obtains $x'_i$, otherwise, obtains $\bot$;
   - Once receiving $x'_i$, $R$ sets $Z = Z \cup \{x'_i\}$;
6. $R$ outputs $Y \cup Z$.

Figure 8: Protocol $\Pi_{\text{PSU}}$ using Cuckoo Hashing.
can leverage the simulators of these subroutine functionalities to simulate the view of corrupted sender $S$. On the other hand, $\text{Sim}_R$ also randomly chooses \(\{a_1, \ldots, a_b\}\) and \(\{F(k, a_1), \ldots, F(k, a_b)\}\) as the outputs of $f_{\text{PS}}$ and $f_{\text{mpOPRF}}$. When constructing \(\{I_i\}_{i \in [n]}\), $\text{Sim}_R$ needs to simulate that there are \(|X \cap Y|\) items in \(\{I_i\}_{i \in [n]}\) that have an intersection with \(\{F(k, a_1), \ldots, F(k, a_b)\}\). Once receiving $b_i$ for $f_{\text{PS}}$, if $b_i = 0$, $\text{Sim}_R$ takes one item from $\{X \cup Y\} - Y$ in random order as the output, otherwise, $\text{Sim}_R$ takes $\bot$ as the output. Likewise, $\text{Sim}_T$ can leverage the simulators of these subroutine functionalities to simulate the remaining view of corrupted receiver $R$. Due to lack of space, more details of the proof are given in the full version [16].

The protocol $\Pi_{\text{PSU}}^R$ is very scalable for balanced datasets, as demonstrated in Section 4. When considering unbalanced datasets, we observe that it is already considerably efficient for the case that the sender’s set size is much larger than the receiver’s, but not so friendly for the opposite case. To deal with this case, we propose a second protocol $\Pi_{\text{PSU}}^S$ as below.

### 3.2 Protocol $\Pi_{\text{PSU}}^S$: Shuffling Sender’s Set

In contrast to $\Pi_{\text{PSU}}^R$, the core idea of designing $\Pi_{\text{PSU}}^S$ is to shuffle the sender’s set, rather than the receiver’s. A brief description is given below and the details are shown in Figure 9.

Similarly, the sender’s set and the receiver’s set are assumed to be $X = \{x_1, \ldots, x_n\}$ and $Y = \{y_1, \ldots, y_m\}$, respectively. Then the protocol works as follows. Firstly, the sender inserts $X$ into the Cuckoo hash table with $b$ bins using $\gamma$ hash functions \(\{h_1, \ldots, h_t\}\) and fills each empty bin with a dummy item $d$. After that, the sender sends to the receiver the parameters of the Cuckoo hashing. Then the receiver inserts $Y$ into a simple hash table with $b$ bins using the same hash functions. In general, when $n$ items are inserted into a simple hash table with $m$ bins using $\gamma$ hash functions, the maximum bin size is $O(m \log m)$ when $n > m \log m$ according to [35]. Therefore, the maximum bin size $\rho$ of simple hash table is $O(n \log_2 b)$. For simplicity, we denote by $X_C$ and $Y_B$ the filled Cuckoo hash table and simple hash table, respectively.

Secondly, the sender and the receiver invoke $f_{\text{PS}}$ with input $X_C$ and a random permutation $\pi$, respectively. Then the sender obtains the shuffled secret shares \(\{a_1, \ldots, a_b\}\) of $X$ while the receiver obtains other shares \(\{a'_1, \ldots, a'_b\}\). Further, by running $f_{\text{mpOPRF}}$ with the input \(\{a_i\}_{i \in [b]}\) the sender obtains \(\{F(k, a_i)\}_{i \in [b]}\) and the receiver obtains the PRF key $k$.

Thirdly, for $i \in [b]$ the receiver generates a set $I_i$ with her $i$-th share $a_i'$ and sends it to the sender, so that the sender can test if the item of $X$ associated with $a_i'$ belongs to $Y$. Note that the receiver selects the permutation $\pi$, she knows that $a_i'$ is the share of the $\pi(i)$-th item of $X_C$, whereas the sender does not know which item of $X_C$ is being tested. Moreover, if $X_C[\pi(i)] \in Y$, then the item in $Y$ equal to $X_C[\pi(i)]$ must be

---

9To be precise, the maximum bin size should be $\Theta(n/m)$.
contained in the $\pi(i)$-th bin of $Y_B$. So to check if $X_C[\pi(i)] \in Y$, we need only to check if $X_C[\pi(i)] \in Y_B[\pi(i)]$. To do so, the receiver computes the XOR of $a_i$ and each item in $Y_B[\pi(i)]$, then evaluates PRFs over them, namely $\{F(k, y_j \oplus a_i)\}_{y_j \in Y_B[\pi(i)]}$, and adds them into set $I_i$. To further hide from the sender the actual number of items in $Y$ mapped to the bin, the receiver pads $I_i$ with $p - |Y_B[\pi(i)]|$ random values from $\{0, 1\}^\ell$ if the number (i.e., $|Y_B[\pi(i)]|$) of items in $I_i$ is less than the maximum bin size $p$. Note that if there is a $y_i$ that is mapped to the bin multiple times using different hash functions, $\Pi_{PSU}^B$ only puts it to the bin once, and thus there are no duplicates in $I_i$.

Finally, after receiving $I_i \supseteq \{F(k, y_j \oplus a'_i)\}_{y_j \in Y_B[\pi(i)]}$, the sender checks if $F(k, a_i) \in I_i$. If not, the sender sends $a_i$ to the receiver in order for the receiver to obtain $X_C[\pi(i)] = a_i \oplus a'_i$, otherwise sends $\bot$. By this way, however, the receiver will learn that an item belonging to $X \cap Y$ is in the bin $Y_B[\pi(i)]$ if she receives $\bot$, which will leak the information about intersection to the receiver as mentioned in [19]. On the other hand, if the receiver obtains the sender’s item, she may learn partial information about the whole sender’s set as she knows the position in Cuckoo hash table to which the item is mapped.

To solve the above problems, we opt to postpone sending the sender’s shares. More specifically, instead of sending the share directly, the sender first records which shares should be sent using a bit-string $U$ and then sends them in a new order. Since the receiver needs to match the shares from the sender with her shares, the receiver also needs to permute her shares in the same order. To this end, we leverage $\Pi_{PSU}$ again. Roughly speaking, the receiver takes her shares $\{a'_1, a'_2, \cdots, a'_k\}$ as the input and the sender randomly chooses a permutation $\pi'$ as his input. After $\Pi_{PSU}$, the sender receives the shuffled shares $\{s'_1, s'_2, \cdots, s'_k\}$, and the receiver obtains the other shares $\{s''_1, s''_2, \cdots, s''_k\}$ where $s'_i \oplus s''_i = a'_i$. As the sender knows the permutation $\pi'$, he can check if $\sigma_{\pi'}(i)$ should be sent to the receiver according to $U[\pi'(i)]$. If so, the sender sends $z_i = a_{\pi'(i)} \oplus s'_i$ to the receiver, otherwise sends $z_i = \bot$. Once receiving $z_i$ from the sender, if it is not $\bot$, the receiver can calculate $z_i \oplus s''_i = a_{\pi'(i)} \oplus s'_i \oplus s''_i = a_{\pi'(i)} \oplus a'_{\pi'(i)}$. If the recovered item is not the dummy item $a$, the receiver will add it into $Z$. Finally, the receiver outputs $Y \cup Z$.

In what follows, we first show the correctness of the protocol $\Pi_{PSU}^B$ in Figure 9 and then argue that it securely realizes the functionality $\Pi_{PSU}^{n_2 \cdot m_2}$.

### Correctness

The analysis is similar to that of $\Pi_{PSU}^{n_1 \cdot m_2}$, and we show it in the full version [16] due to the limited space.

### Security

Now we show that $\Pi_{PSU}^B$ securely realizes $\Pi_{PSU}^{n_1 \cdot m_2}$.

**Theorem 3.2.** The protocol $\Pi_{PSU}^B$ presented in Figure 9 securely realizes $\Pi_{PSU}^{n_1 \cdot m_2}$ in the $(\Pi_{PSU}, \Pi_{OPRF})$-hybrid model, in the presence of semi-honest adversaries.

The proof of Theorem 3.2 is similar to Theorem 3.1. Due to the limited space, the details are given in the full version [16].

### 3.3 Cost Analysis

Given the statistical security parameter $\lambda$, according to [34], we choose the parameters of Cuckoo hashing without stash, exactly including $b = \varepsilon \cdot n$ bins (where $n = n_2$ for $\Pi_{PSU}^B$ and $n = n_1$ for $\Pi_{PSU}^R$) and $\gamma$ hash functions, to ensure that the hashing failure probability is less than $2^{-\lambda}$. Besides, to guarantee the error rate incurred by collisions is less than $2^{-\lambda}$ in $\Pi_{PSU}^R$, we set the share/item length $\ell_1$ and the output length $\ell_2$ of $F(k, \cdot)$ to be at least $\lambda + \log(\gamma n_1) + \log(\gamma n_2)$. Likewise, the output length $\ell_2$ of $F(k, \cdot)$ in $\Pi_{PSU}^B$ is at least $\lambda + \log(\gamma n_1) + \log(\gamma n_2)$. In addition, the costs of $\Pi_{PSU}^R$ and $\Pi_{PSU}^B$ also rely on the sub-protocols used to realize the building blocks. Particularly, we realize $\Pi_{PSU}, \Pi_{OPRF}$ and $\Pi_{OT}$ with the protocols in [22], [6] and [15], respectively.

| Table 4: The costs of $\Pi_{PSU}^R$ and $\Pi_{PSU}^B$ |
| Part-1 | Part-2 | Part-3 | Part-4 |
| Comp. | $O(b \log b)$ | $O(b)$ | $O(\gamma n)$ | $O(n_1)$ |
| Comm. | $O(\ell_1 b \log b)$ | $O(b)$ | $O(\ell_2 \gamma n)$ | $O(\ell_1 n_1)$ |

Since $\Pi_{PSU}^R$ and $\Pi_{PSU}^B$ mainly consist of 4 parts: (1) Permuted Share, (2) multi-point OPRF, (3) computing and sending $\{b\}$, and (4) obtaining items in $X \setminus Y$, we for clarity summarize their complexities in Table 4 according to each part. In $\Pi_{PSU}^R$ (resp. $\Pi_{PSU}^B$), Part-1 and Part-2 are performed on the receiver’s set $Y$ (resp. the sender’s set $X$) while Part-3 is performed on the sender’s set $X$ (resp. the receiver’s set $Y$), and thus $b = \gamma n_2, n = n_1$ (resp. $b = \gamma n_1, n = n_2$).

### 4 Performance Evaluation

In this section, we experimentally evaluate our PSU protocols $\Pi_{PSU}^R$ and $\Pi_{PSU}^B$. In Section 4.1, we first give our benchmarking environment. In section 4.2, we compare our protocols with the state-of-the-art work [19] in terms of communication cost and single-threaded runtime on different networks, and the results are reported in Table 5. To demonstrate the scalability and parallelizability of our protocols, we evaluate our two protocols on small and large sets with different threads in Section 4.3, and show the results in Table 6, Table 7 and Table 8. Besides the equal set sizes, we also consider the unbalanced sets in Section 4.4. We perform $\Pi_{PSU}^R$ in the cases where the sender’s set is larger than the receiver’s set, and $\Pi_{PSU}^B$ in the opposite cases, and show the results in Table 9. Our complete implementation is available on GitHub: https://github.com/dujiajun/PSU.

### 4.1 Benchmarking Environment

We implement $\Pi_{PSU}^R$ and $\Pi_{PSU}^B$ in C++, and run our experiments on a single Intel Xeon with 2.39GHz and 128GB RAM. We evaluate our protocols in two networks settings, LAN network with 10Gbps bandwidth and 0.02 ms RTT and WAN.
network with 400Mbps and 80ms RTT, which are emulated using Linux tc command. We set the computational security parameter $\kappa = 128$ and statistical security parameter $\lambda = 40$, and the item length in bits $\ell_1$ is 128.

Our protocols are built on Permute + Share, multi-point OPRF, and OT extension. We implement Permute + Share with the design in [22] and OT extension [15] using libOTe library [38] with Naor-Pinkas Base OT [24]. For multi-point OPRF, we use the source code from [6].

**Parameters about Cuckoo Hashing.** For the equal set sizes, Permute + Share sub-protocol costs most of the runtime, and thus we need to minimize the number of items to be shuffled as far as possible. Moreover, items in the stash of Cuckoo hashing need to be compared with each item of the other party rather than certain items picked out by the hash functions. Hence, we also need to limit the stash size to be 0.

The empirical analysis in [34] shows that increasing the number of hash functions can drastically reduce the number of bins and the required stash size. According to the results reported in [34], we decide to use 4 hash functions to implement Cuckoo hashing with 1.09 $\cdot n$ bins and 0 stash. However, for the unbalanced set sizes, calculating PRFs will dominate, to reduce the number of PRFs calls and keep the stash size 0, we choose to use 3 hash functions and 1.27 $\cdot n$ bins.

### 4.2 Performance Comparisons

In this section, we compare $\Pi_{PSU}^R$, $\Pi_{PSU}^S$ and [19] in terms of runtime and communication, and the results are reported in Table 5. More concretely, compared to [19], our $\Pi_{PSU}^R$ can obtain a 4-5× improvement in runtime for large datasets ($n_1 = n_2 \geq 2^{14}$) in both WAN and LAN settings. And the communication is about 50% communication of [19].

Although we also consider $\Pi_{PSU}^R$, in this comparison, it is worth noting that $\Pi_{PSU}^R$ realises a different functionality than the other two protocols, because the sender can obtain the intersection size in advance. Moreover, to avoid the leakage of intersection information, $\Pi_{PSU}^S$ has to execute the Permute + Share sub-protocol twice (cf. Section 3.2 for more details). Therefore, the runtime of $\Pi_{PSU}^S$ is longer than that of $\Pi_{PSU}^R$. Nevertheless, compared to [19], our $\Pi_{PSU}^S$ can still obtain a 3-4× improvement in runtime for large datasets ($n_1 = n_2 \geq 2^{14}$) in both WAN and LAN settings. But for communication, the cost of $\Pi_{PSU}^S$ is almost equal to that of [19].

### 4.3 Scalability and Parallelizability

In this section, we show that our two protocols can be efficiently executed on small datasets without Cuckoo hashing, which can simplify development in practice. And using Cuckoo hashing, our protocols can be scaled to large sets. Moreover, we show that our protocols can be executed in parallel. Specifically, for the set of size $2^2$, $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ can achieve a speedup of 3.49× and 2.89× using 8 threads, respectively.

**The cases with small sets.** In Table 6, we show that $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ can be executed on small sets directly without Cuckoo hashing (please see the full version [16] for more details of $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$). In LAN setting, they cost about 10 seconds with 8 threads on the set of size $2^{12}$. However, we can see that as the set size increases, the runtime and communication increase dramatically. For set size of $2^{14}$, the two protocols both need more than 3 minutes with 8

<table>
<thead>
<tr>
<th>Protocol</th>
<th>$2^8$</th>
<th>$2^{10}$</th>
<th>$2^{12}$</th>
<th>$2^{14}$</th>
<th>$2^8$</th>
<th>$2^{10}$</th>
<th>$2^{12}$</th>
<th>$2^{14}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi_{PSU}^R$</td>
<td>1.064</td>
<td>1.379</td>
<td>2.164</td>
<td>5.326</td>
<td>17.541</td>
<td>86.358</td>
<td>333.073</td>
<td>1459.539</td>
</tr>
<tr>
<td>$\Pi_{PSU}^S$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Pi_{PSU}^{\text{Permute}}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Comparisons of total runtime (in seconds) and communication (in MB) between $\Pi_{PSU}^R$, $\Pi_{PSU}^S$ and [19] with a single thread in WAN/LAN settings where $n_1 = n_2 = n$. Best results are marked in bold.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>$2^8$</th>
<th>$2^{10}$</th>
<th>$2^{12}$</th>
<th>$2^{14}$</th>
<th>$2^8$</th>
<th>$2^{10}$</th>
<th>$2^{12}$</th>
<th>$2^{14}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi_{PSU}^R$</td>
<td>0.671</td>
<td>0.892</td>
<td>1.132</td>
<td>1.778</td>
<td>4.412</td>
<td>16.104</td>
<td>67.756</td>
<td>341.758</td>
</tr>
<tr>
<td>$\Pi_{PSU}^S$</td>
<td>0.712</td>
<td>0.993</td>
<td>1.238</td>
<td>2.214</td>
<td>6.233</td>
<td>22.78</td>
<td>102.039</td>
<td>458.731</td>
</tr>
<tr>
<td>$\Pi_{PSU}^{\text{Permute}}$</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Runtime (in seconds) and communication (in MB) of $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ for small set ($n_1 = n_2 \in \{2^8, 2^{10}, 2^{12}, 2^{14}\}$) and threads $T \in \{1, 2, 4, 8\}$ threads in LAN setting.
In this section, we show that $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ can be chosen according to the sizes of the two sets. Considering that the set to be shuffled is small ($2^8$ or $2^{12}$) and the items that will be calculated PRFs are too many (more than $2^{16}$, $2^{20}$ or $2^{24}$), we adjust the parameters of Cuckoo hashing to 3 hash functions and $1.27 \cdot n$ bins with stash size $s = 0$ according to the results in [34]. Table 9 shows the performances of $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ with 8 threads in WAN/LAN settings.

When the receiver’s set is much smaller than the sender’s set (i.e., $n_2 \ll n_1$), we perform $\Pi_{PSU}^S$ to obtain the union. We can see that for the sender’s size of set $2^{24}$, $\Pi_{PSU}^R$ only needs about 50 seconds in LAN setting, and about 65 seconds in WAN setting, which is reasonable in practice. As for the opposite unbalanced cases (i.e., $n_1 \ll n_2$), $\Pi_{PSU}^S$ can obtain a better performance since it replaces OT related to the larger set with Permute + Share just related to the smaller set.

### 4.4 Design for Unbalanced Datasets

The cases with large sets. In both $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$, PRF values in set $T$ are independent of each other, and thus can be calculated in parallel. In addition, the Permute + Share sub-protocol in [22] and the multi-point OPRF sub-protocol in [6] can be partially parallelized. We demonstrate the scalability and parallelizability of $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ by evaluating it on the large set sizes $n_1 = n_2 = n \in \{2^{12}, 2^{14}, 2^{16}, 2^{20}, 2^{22}\}$ in parallel with $T \in \{1, 2, 4, 8\}$ threads. Table 7 shows the experimental results of $\Pi_{PSU}^R$ in both WAN/LAN settings, and the last row presents the ratio between the runtime of the single thread and 8 threads. We can see that, the speedup becomes better as the set size increases. Specifically, when the set size is $2^{22}$, we can obtain a speedup of $2.08 \times$ in WAN setting and $3.49 \times$ in LAN setting. Similarly, we report the results of $\Pi_{PSU}^S$ in Table 8. On the whole, as the set sizes and number of threads increase, the runtime of $\Pi_{PSU}^S$ changes in the same way as that of $\Pi_{PSU}^R$. However, the speedup of $\Pi_{PSU}^S$ is less than that of $\Pi_{PSU}^R$, since $\Pi_{PSU}^S$ performs Permute + Share sub-protocol twice, which is not completely parallelized.
As pointed out by Kolesnikov et al. in [19], however, we further explain why the protocol in [19] cannot benefit from Cuckoo hashing in Appendix B.2.

In order to improve the performance, Kolesnikov et al. [19] proposed to optimize their protocol by using the bucketing technique, as shown in Figure 10. More specifically, the sender and receiver in [19] 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 [19], however, the bucketing technique will leak the information “which bins contain items in $X \cap Y$” to the receiver. To avoid this leakage, in [19] 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 10, 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, we observe that this strategy is insufficient to avoid the leakage incurred by the bucketing technique. A detailed analysis is given below.

### Leaks 

We denote the probability that $\text{Case}_2$ occurs by $\Pr[\text{Case}_1] = 1 - \Pr[\text{Case}_2]$, we now bound $\Pr[\text{Case}_2]$. In general, assuming that there are $\alpha n$ bins and $n$ items and the hash mapping is a random oracle, the probability $\text{Case}_2$ occurs is $P = \Pr[\text{Case}_2] = (1 - \frac{1}{\alpha n})^n \approx e^{-1/\alpha}.

Based on the parameters in [19], we calculate the probability $P$ for different set sizes as shown in Table 10. From the results, we can see that the probability $P$ is very small. For example, when the set size is $n = 2^{20}$, $P = 5.778 \times 10^{-8}$. 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.
belonging to $X \cap Y$ are in a bin (namely, a subset of $Y$) with an overwhelming probability.

6 Discussion

In this work, we focus on designing efficient PSU protocols for both balanced and unbalanced datasets. Somewhat surprisingly, our techniques can also be used for designing PSI protocols with only slight modification. With our techniques, it is extremely convenient to design fast protocols when both functionalities, set intersection and set union, are required; details are below.

Recall that in $\Pi_{PSU}^R$, the receiver sets $b_i$ depending on whether the sender’s item $x_i$ belongs to her set $Y$; if $x_i \in Y$, sets $b_i = 1$, otherwise $b_i = 0$. Then through $f_{OT}$ the receiver obtains $x_i$ if $b_i = 0$, and nothing otherwise. To obtain a PSI protocol from $\Pi_{PSU}^R$, the receiver only needs to set $b'_i = b_i \oplus 1$ and obtains the sender’s items through $f_{OT}$ according to $b'_i$, rather than $b_i$. In this way, the receiver will obtain the sender’s items belonging to $Y$. Thus we obtain a PSI protocol, denoted by $\Pi_{PSU}^R$. In $\Pi_{PSU}^S$, the sender sends the shares of the items in $X \setminus Y$ according to the bit string $U$, then the receiver uses them to recover the items in $X \setminus Y$. Therefore, the sender can send the shares associated with the items in $X \setminus Y$ by flipping each bit in $U$, and then the receiver will obtain items in $X \setminus Y$. Thus we obtain a PSI protocol, denoted by $\Pi_{PSU}^S$.

It can be seen that $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ are obtained from the PSU protocols with almost no extra overhead. Therefore, it is believed that they have nearly the same performance as the proposed PSU protocols. Due to the page limit, we leave their formal descriptions and security analysis to future work.

In addition, as stated in Section 1.3, it is desirable to have better PSU protocols designed. Notice that our protocols, $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ are designed in a modular manner; thus, a natural way to achieve better performance is to improve the performance of the underlying building blocks. More concretely, if the underlying Permuto + Share protocol can be designed with linear complexity, the cost of $\Pi_{PSU}^R$ and $\Pi_{PSU}^S$ can be reduced to be linear. Finally, it is also interesting to design PSU with better security (e.g., defending against malicious adversaries) and/or with better functionalities (e.g., new variants of PSU including multi-party PSU and PSU with payload).

Acknowledgments

We thank the anonymous reviewers and especially our shepherd, Mayank Varia, for their insightful suggestions and comments, that substantially helped in improving the paper. This work was supported in part by the National Key Research and Development Project 2020YFA0712300. Hong-Sheng Zhou acknowledges support by NSF grant CNS-1801470, a Google Faculty Research Award and a research gift from Ergo Platform.

References


A Additional Materials for Section 2: Building blocks

In this section, we show the formal functionalities of the main building blocks, including Permute + Share functionality $F_{PS}$ in Figure 11, multi-point OPRF functionality $F_{mpOPRF}$ in Figure 12, and 1-out-of-2 Oblivious Transfer functionality $F_{OT}$ in Figure 13.

---

B Additional Materials for Section 5

In this section, we first describe the protocol in [19], and then further explain why the protocol in [19] and earlier constructions [7, 9] cannot benefit from Cuckoo hashing.
B.1 Details of the protocol in [19]

According to the design idea in [19], we first give the protocol for \((1,n)\)-PSU where the sender only holds an item \(x^*\) and the receiver holds a set \(Y = \{y_1, \ldots, y_n\}\) in Figure 15. More specifically, their protocol works in the following way: the two parties first execute an OPRF sub-protocol for \(F\), then the sender obtains \(F(k, x)\) without knowing \(k\) and the receiver obtains the PRF key \(k\). After that, the receiver interpolates a polynomial \(P\) over points \(\{(y, s \oplus F(k, y))\}_{y \in Y}\), where \(s\) is a random value chosen by the receiver, and sends \(P\) to the sender. Once receiving the polynomial \(P\), the sender calculates \(s' = P(x) \oplus F(k, x)\) and sends \(s'\) to the receiver. Then the receiver checks if \(s' = s\). If not, meaning that \(x \notin Y\), the receiver obtains \(x\) through OT, otherwise obtains a dummy item.

Kolesnikov et al. [19] extend \((1,n)\)-PSU to the general case (namely, \((n_1,n_2)\)-PSU) by repeatedly using \((1,n)\)-PSU. Then in Figure 15, we give the protocol for \((n_1,n_2)\)-PSU where the sender and receiver hold \(X = \{x_1, \ldots, x_{n_1}\}\) and \(Y = \{y_1, \ldots, y_{n_2}\}\), respectively. Note that for each sender’s item, both the secret value \(x\) and the key \(k\) for OPRF need to be refreshed, otherwise the sender can learn information about the intersection.

\((1,n)\)-PSU protocol in [19]

**Parameters:**
- A bit-length \(\ell\);
- Ideal functionalities \(\mathcal{F}_{\text{OT}}\) and \(\mathcal{F}_{\text{OPRF}}\) \((F(k, x) \in \{0,1\}^\sigma)\);
- A collision-resistant hash function \(h(x) : \{0,1\}^\ell \rightarrow \{0,1\}^{\sigma}\);

**Inputs:**
- Sender \(S\): \(x^* \in \{0,1\}^{\ell}\);
- Receiver \(R\): set \(Y = \{y_1, \ldots, y_n\}\), \(y_i \in \{0,1\}^{\ell}\);

**Protocol:**
1. \(S\) acts as \(\mathcal{F}_{\text{OPRF}}\) receiver, sends \(x^*\) to \(\mathcal{F}_{\text{OPRF}}\), and \(S\) receives \(q^* = F(k, x^*)\) and \(R\) receives \(k\);
2. \(R\) randomly picks \(s \in \{0,1\}^\sigma\), and interpolates a polynomial \(P(y)\) over points \(\{(b(y_i), s \oplus q_i)\}\) where \(q_i = F(k, y_i)\), \(\forall i \in [n]\). Here \(s \oplus q_i\) is computed as operation on \(\sigma\)-bit strings.
3. \(R\) sends the coefficients of \(P(y)\) to \(S\);
4. \(S\) computes \(s' = P(h(x^*)) \oplus q^*\) and sends it to \(R\);
5. \(S\) and \(R\) invoke \(\mathcal{F}_{\text{OT}}\):
   - \(R\) acts as receiver with input 1 if \(s' = s\) and input 0 otherwise;
   - \(S\) acts as sender with input \((x^*, \bot)\);
6. If \(s' = s\), then \(R\) gives output \(Y\). Otherwise, it learns \(x^*\) and outputs \(Y \cup x^*\).

Figure 14: \((1,n)\)-PSU protocol in [19].

B.2 Discussion about Cuckoo hashing

As for Cuckoo hashing, Kolesnikov et al. [19] pointed out that “this hashing scheme (and the corresponding performance improvement) does not immediately fit in the PSU case.” Recall that the protocols [7, 9, 19] share the same design framework as in Figure 1. We can see that the sender’s set can be inserted into Cuckoo hash table. Then, for the item in each bin of Cuckoo hash table, the receiver will check if it belongs to the intersection, if not, the receiver will get the item. Note that the receiver also knows the item’s position in Cuckoo hash table. Since the position of an item in Cuckoo hash table is also affected by other items, the receiver can obtain partial information about the sender’s entire input set based on the received item and its position in Cuckoo hash table.

\((n_1,n_2)\)-PSU protocol in [19]

**Parameters:**
- A bit-length \(\ell\) and \(n = \max(n_1,n_2)\);
- Number of bins \(\beta = \beta(n)\), hash function \(H : \{0,1\}^{\ell} \rightarrow [\beta]\), and max bin size \(m\);
- A special item \(\bot \in \{0,1\}^{\ell}\);

**Inputs:**
- Sender \(S\): \(X = \{x_1, \ldots, x_{n_1}\}, x_i \in \{0,1\}^{\ell}\);
- Receiver \(R\): \(Y = \{y_1, \ldots, y_{n_2}\}, y_i \in \{0,1\}^{\ell}\);

**Protocol:**
1. \(S\) and \(R\) hash items of their sets \(X\) and \(Y\) into \(\beta\) bins under hash function \(H\). Let \(B_{S}^{\ell}[\cdot]\) and \(B_{R}^{\ell}[\cdot]\) denote the set of items in the sender’s and receiver’s \(i\)-th bin, respectively;
2. \(S\) pads each bin \(B_{S}^{\ell}[i]\) with the special item \(\bot\) up to the maximum bin size \(m + 1\), and randomly permutes all items in this bin;
3. \(R\) pads each bin \(B_{R}^{\ell}[i]\) with one special item \(\bot\) and different dummy items to the maximum bin size \(m + 1\);
4. \(R\) initializes set \(Z = \emptyset\);
5. For each bin \(i \in [\beta]\), for each item \(x_j \in B_{S}^{\ell}[i]\):
   - \(S\) and \(R\) invoke the \((1,n)\)-PSU sub-protocol in Figure 14 and \(n = m + 1\):
     * \(S\) acts as sender with input \(x_j\);
     * \(R\) acts as receiver with input set \(B_{R}^{\ell}[i]\);
     * \(R\) obtains output \(Z_{i,j}\) and sets \(Z = Z \cup Z_{i,j}\);
6. \(R\) outputs \(Z\).

Figure 15: \((n_1,n_2)\)-PSU protocol in [19].