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*
{jiayanxue,shifeng.sun,cqdujiajun,dwgu}@sjtu.edu.cn
[‡] *Virginia Commonwealth University, USA*
hszhou@vcu.edu

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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, 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 $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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[1]. Whereas, he cannot do this via the IP addresses not in his own blacklist[2]. 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ (cf. the high level ideas in Section 1.1), a joint

⋆ Corresponding authors.

[1] The curious organization knows his own detection strategy that is used to identify the IP addresses in the intersection. Thus, it is reasonable for him to deduce that a similar detection strategy is used by the other organization.

[2] Different organizations usually monitor different areas of the Internet, as mentioned in [37]. Therefore, the curious organization knows nothing about the traffic through these IP addresses. Even if the curious organization can monitor the traffic, it is difficult to deduce the detection strategy as e.g., some sensitive strategies are based on features that can be only extracted from encrypted data.

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 *unscalable* 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

[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.

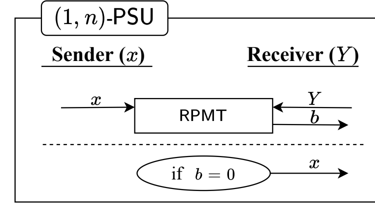
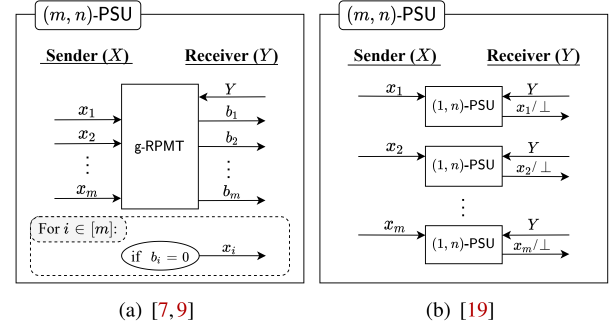


Figure 1: Design framework for $(1,n)$-PSU protocols.

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.



(a) [7,9] (b) [19]

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 $X = \{x_1, x_2, \ldots, x_m\}$ 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 $m = n$, 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 $X_1, \cdots, X_b$ (resp., $Y_1, \cdots, Y_b$). Then, each bin is padded with dummy items up to the maximum bin size ρ. Note that $b = O(n/\log n)$ and $\rho = O(\log n)$ in [19]. The two parties perform a (ρ, ρ)-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 $O(b\rho^2 \log^2 \rho) = O(n \log n \log^2 \log n)$.

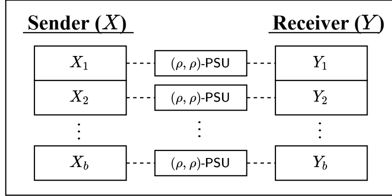


Figure 3: Bucketing technique in [19].

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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ under the design framework in Figure 2(a) by *shuffling* the receiver's set. Our protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$; this new protocol can be viewed as the dual version of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, exactly by shuffling the sender's set. More details about protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ are given in Section 1.1 and Section 3.

1.1 Technical Overview

Protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: shuffling receiver's set. We now present how to construct practical two-party PSU protocols following the design framework in Figure 2(a). A big challenge is that, we need to efficiently realize the functionality g-RPMT, which allows the receiver to perform membership tests while not revealing the receiver's set Y to the sender. In previous works [7, 9], functionality g-RPMT has been realized but not efficiently due to the heavy public-key operations.

The high-level idea of our design is shown in Figure 4 and the details are as follows. Initially, each item $y \in Y$ is split into two shares s and $y \oplus s$ by an additive secret sharing, where s is distributed uniformly at random and perfectly hides y. The set $Y = \{y_1, \cdots, y_n\}$ can be shared into two sets $\{s_1, \cdots, s_n\}$ and $\{y_1 \oplus s_1, \cdots, y_n \oplus s_n\}$; the receiver will keep the former herself, and send the latter to the sender. Now we can see that for each item $x_i \in X$, if it belongs to Y (i.e., $\exists\ j$ s.t. $x_i = y_j$), then $x_i \oplus y_j \oplus s_j = s_j$. Thus the sender can compute and send $\{x_i \oplus y_1 \oplus s_1, \cdots, x_i \oplus y_n \oplus s_n\}$ to the receiver, and the receiver can check if the sender's item x_i belongs to Y by computing the intersection of this set and $\{s_1 \cdots, s_n\}$. If empty, it means $x_i \notin Y$. Otherwise, the receiver learns that $x_i \in Y$.

Now, the receiver can learn if $x_i \in Y$ without revealing Y to the sender. However, if $x_i \in Y$, the receiver can *additionally* figure out which item of Y is exactly equal to x_i according to $s_j \in \{x_i \oplus y_1 \oplus s_1, \cdots, x_i \oplus y_n \oplus s_n\} \cap \{s_1, \cdots, s_n\}$ as she knows the mapping of the shares $\{s_1, \cdots, s_n\}$ to the items in Y. Note that such information is not allowed to be obtained by the receiver in the RPMT. Thus, next we need to find a way to defend against such information leakage.

Recall that, the receiver is able to obtain additional information is due to the fact that the receiver knows which

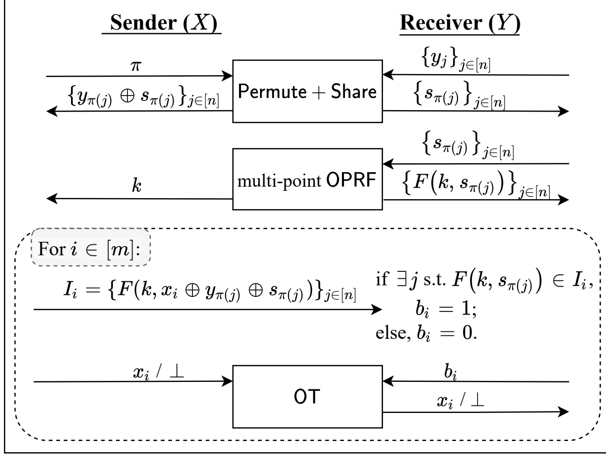


Figure 4: Core design idea of protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ for (m,n)-PSU.

share corresponds to which item in Y. Our **key design idea** now 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 $Y = \{y_1, \cdots, y_n\}$ from the receiver and a permutation π (over $\{1, 2, \cdots, n\}$) from the sender, the functionality outputs the shares $\{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ and $\{y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, y_{\pi(1)} \oplus s_{\pi(n)}\}$ to the receiver and the sender, respectively. After executing this functionality, the sender computes $\{x_i \oplus y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, x_i \oplus y_{\pi(n)} \oplus s_{\pi(n)}\}$ and sends it to the receiver. Then the receiver can check if the sender's item belongs to Y as before. Following this way, the receiver will learn that there is an item $s_{\pi(j)} \in \{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ equal to $x_i \oplus y_{\pi(j)} \oplus s_{\pi(j)}$ if $x_i \in Y$, but she is unable to find out $y_{\pi(j)}$ according to $s_{\pi(j)}$, as she does not know π.

At this point, it seems that x_i can be completely hidden from the receiver at the first glance. Unfortunately, $x_i \oplus y_{\pi(j)} \oplus s_{\pi(j)}$ may still leak partial information about x_i to the receiver. This is because $y_{\pi(j)} \oplus s_{\pi(j)}$ is not distributed uniformly and independently from the perspective of the receiver who knows $s_{\pi(j)}$ and $y_{\pi(j)}$. To overcome this obstacle, we further employ multi-point Oblivious Pseudorandom Function (OPRF)[4] $F(k, \cdot)$ to conceal $\{x \oplus y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, x \oplus y_{\pi(n)} \oplus s_{\pi(n)}\}$. More concretely, the receiver takes $\{s_{\pi(1)}, \cdots, s_{\pi(n)}\}$ as the input to multi-point OPRF, then the sender receives the PRF key k and the receiver obtains $\{F(k, s_{\pi(1)}), \cdots, F(k, s_{\pi(n)})\}$. In this case, the sender with the key k can compute $I_i = \{F(k, x_i \oplus y_{\pi(1)} \oplus s_{\pi(1)}), \cdots, F(k, x_i \oplus y_{\pi(n)} \oplus s_{\pi(n)})\}$ 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., $I_1, \cdots, I_m$) 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_i 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 $\{y_{\pi(1)} \oplus s_{\pi(1)}, \cdots, y_{\pi(n)} \oplus s_{\pi(n)}\}$. 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 γ hash functions, $\varepsilon \cdot n$ bins and stash size 0[5], and then execute $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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_i, and that the receiver's set $Y = \{y_1, \cdots, y_6\}$, can be inserted to the Cuckoo hash table with 8 bins and 2 hash functions $h_1(\cdot)$ and $h_2(\cdot)$[6]. 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 π and Y^* as inputs, and obtain the shuffled secret share sets Sh_1 and Sh_2, respectively. Here, s_d^1 and s_d^2 in Sh_1 are the shares of dummy item and $s_d^1 \neq s_d^2$. The dotted arrows mean that after permutation π, the 4-th (resp. 7-th) item in Sh_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 Sh_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.

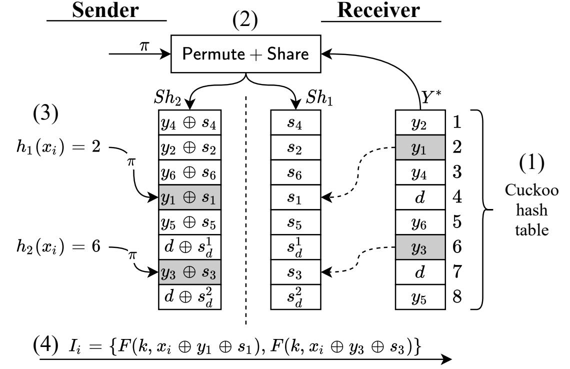


Figure 5: $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: Optimization via Cuckoo hashing.

[5]According to the empirical analysis in [34], the stash size can be reduced to 0 by increasing the number of hash functions while achieving a hashing failure probability of 2^{-40}.

[6]Note that the parameters used here are to simplify the explanation, please refer to Section 4.1 for the concrete parameter choices.

$h_1(x_i)$ and $h_2(x_i)$, say $h_1(x_i) = 2$ and $h_1(x_i) = 6$; we note that according to the principle of filling the Cuckoo hash table, the potential item of Y that is equal to x_i must be inserted to the position $h_1(x_i)$ or $h_2(x_i)$ of Y^*. Then he uses π to locate the corresponding shares (namely, 4-th and 7-th items) in Sh_2 and generates I_i with them. **(4)** The sender sends I_i to the receiver and the receiver proceeds as before.

Based on the above optimization, we can reduce the number of items in I_i to a constant γ that is the number of hash functions. Hence, the computation and communication cost incurred by $\{I_1, \cdots, I_m\}$ can be reduced to $O(\gamma m)$. In Table 1, we summarize the computation and communication costs of main steps in our $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. And we will give a more detailed complexity analysis by taking account of the error rate and security parameters in Section 3.3.

Table 1: The computation and communication costs of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$.

	Permute + Share	multi-point OPRF	$\{I_i\}_{i\in[m]}$	OT
Costs	$O(n \log n)$	$O(n)$	$O(m)$	$O(m)$

m is the sender's set size; n is the receiver's set size.

Protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$: shuffling sender's set. From Table 1 we can see that when $m \gg n$ the overall cost of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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, $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ by *shuffling the sender's set*. As a whole, it can be regarded as the dual version of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. The high-level idea is shown in Figure 6.

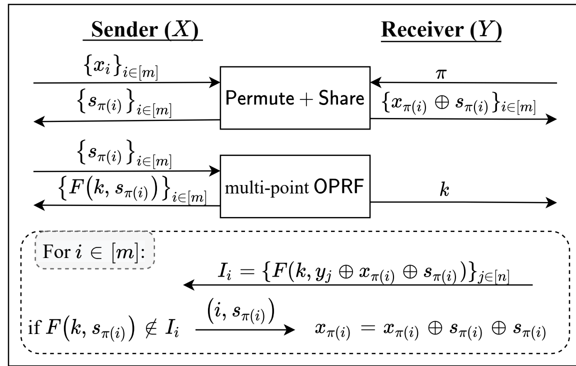


Figure 6: Core idea of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ for (m,n)-PSU.

The basic idea of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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. $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can also be optimized via Cuckoo hashing following the similar idea as in $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. More specifically, the sender's set X and the receiver's set Y are inserted into a Cuckoo hash table and a simple hash table (defined in Section 2.2), respectively. Then, the receiver uses the items stored in each bin (of the simple hash table), rather than all the items in Y, to generate I_i. Therefore, the size of I_i is reduced to the maximum bin size of the simple hash table. As pointed out in [19], however, the receiver may learn if each bin has any item in $X \cap Y$. To avoid this leakage, we use the shuffling technique again. For sake of clarity, we include an example in the full version [16] to illustrate our main idea of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$.

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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, 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 $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$. 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 leverage 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is 4-5× 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; $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can achieve a speedup of 2.89× and 3.49× 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 PSU so far.

The first PSU 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 PSU 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 $\mathsf{Enc}(f)$ of her own set Y, then the sender with set X calculates the tuples $(\mathsf{Enc}(xf(x)), \mathsf{Enc}(f(x)))$ 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[7].

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× slower than [19] due to heavy public-key operations. Note that our protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is 4-5× 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.

Protocol	Comm. (bits)	Comp. (#Ops)	
		pub-key	symm-key
[17]	$O(n^2)$	$O(n^2)$	-
[9]	$O(n)$	$O(n \log\log(n))$	-
[7]	$O(n)$	$O(n)$	-
[19]	$O(n\log(n))$	-	$O(n\log(n))$
$\Pi^{\mathsf{R/S}}_{\mathsf{PSU}}$	$O(n\log(n))$	-	$O(n\log(n))$

Table 2: Comparisons of asymptotic communication (bits) and computation (#operations) costs of two-party PSU 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 κ base OTs where κ 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 PSU. Our $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ outperforms Kolesnikov et al. [19] by a factor of 5.4, but is still 20× lower than the PSI protocol in [18]. More research on PSU design should be encouraged to further improve the performance.

	PSU		PSI	
	[19]	$\Pi^{\mathsf{R}}_{\mathsf{PSU}}$	[18]	[40]
Time (s)	263.476	48.703	2.441	5.396
Comm.(MB)	2470.11	1338.79	128.5	53.55

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× 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].

[7] If there is an item mapped to an entry of Bloom Filter, the entry will be filled with a bit 0, otherwise, with a bit 1.

2 Preliminaries

Notation. We denote by κ and λ 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 size $|X|=n$. Given a permutation π 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 Π on inputs (X,Y) be $\mathsf{view}_i^{\Pi}(X,Y)$ 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, $\mathsf{output}_i^{\Pi}(X,Y)$ is the output of P_i during an execution of Π on inputs (X,Y) and can be computed from the Π's view. And the joint output of both parties is $\mathsf{output}^{\Pi}(X,Y)=(\mathsf{output}_0^{\Pi}(X,Y),\mathsf{output}_1^{\Pi}(X,Y))$.

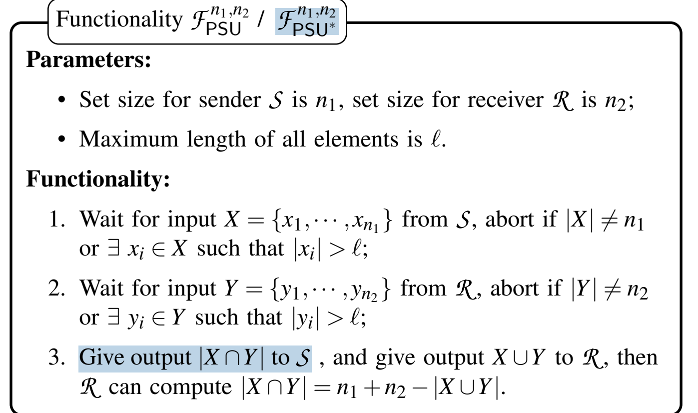
Definition 2.1. *A protocol* Π *securely computes* $\mathcal{F}$ *against static semi-honest adversaries if there exist probabilistic polynomial-time (PPT) algorithms* Sim_0 *and* Sim_1 *such that*

$$(\mathsf{Sim}_0(X,f_0(X,Y)),f(X,Y))\overset{c}{\equiv}(\mathsf{view}_0^{\Pi}(X,Y),\mathsf{output}^{\Pi}(X,Y)),$$
$$(\mathsf{Sim}_1(Y,f_1(X,Y)),f(X,Y))\overset{c}{\equiv}(\mathsf{view}_1^{\Pi}(X,Y),\mathsf{output}^{\Pi}(X,Y)).$$

PSU **Functionality.** The ideal functionality for PSU, denoted as $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$, 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 both 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 $\mathcal{F}_{\mathsf{PSU}^*}^{n_1,n_2}$; we also show it in Figure 7 while the difference from $\mathcal{F}_{\mathsf{PSU}}^{n_1,n_2}$ is marked in blue.


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

Parameters:

- Set size for sender $\mathcal{S}$ is n_1, set size for receiver $\mathcal{R}$ is n_2;
- Maximum length of all elements is ℓ.

Functionality:

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

Figure 7: Ideal Functionalities for PSU (The difference is marked in blue).

2.2 Building Blocks

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

Permute + Share. The Permute + Share functionality $\mathcal{F}_{\mathsf{PS}}$ 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 $X=\{x_1,\cdots,x_n\}$ of size n and P_1 picks a permutation π on n elements. The goal of $\mathcal{F}_{\mathsf{PS}}$ 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 $\{x_{\pi(1)}\oplus s_{\pi(1)},x_{\pi(2)}\oplus s_{\pi(2)},\cdots,x_{\pi(n)}\oplus s_{\pi(n)}\}$. As mentioned in [5], some earlier works [13, 22] can also be used for securely realizing $\mathcal{F}_{\mathsf{PS}}$. These solutions all have computation/communication complexity $O(n\log n)$. The functionality $\mathcal{F}_{\mathsf{PS}}$ is shown in Figure 11 in Appendix A.

Multi-Point Oblivious PRF**.** Oblivious PRF (OPRF) is a protocol involving two parties P_0 and P_1, where P_1 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 $\{x_1,\cdots,x_n\}_{n\geq 1}$ and obtains $\{F(k,x_1),\cdots,F(k,x_n)\}_{n\geq 1}$ 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 $\mathcal{F}_{\mathsf{mpOPRF}}$ 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 noth-

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 $\mathcal{F}_{\mathsf{OT}}$ is shown in Figure 13 in Appendix A.

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

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

Cuckoo Hashing. Cuckoo hashing was introduced by Pagh and Rodler in [26]. In this hashing scheme, there are γ hash functions $h_1, \cdots, h_\gamma$ used to map n items into $b = \varepsilon n$ bins and a stash, and we denote the i-th bin as B_i. Unlike the simple hashing, the Cuckoo hashing can guarantee that there is only one item in each bin, and the approach to avoid collisions is as follows: For an item x, it can be inserted into any empty bin of $B_{h_1(x)}, B_{h_2(x)}, \cdots, B_{h_\gamma(x)}$. If there are no empty bins in the k bins, randomly select a bin $B_{h_r(x)}$ in these γ bins, and evict the prior item y in $B_{h_r(x)}$ where $h_r(x) = h_r(y)$ to a new bin $B_{h_i}(y)$ where $i \neq r$. The above procedure is repeated until no more evictions are necessary, or until the number of evictions has reached a threshold. In the latter case, the last item will be put in the stash. According to the empirical analysis in [34], we can adjust the values of γ and ε to reduce the stash size to 0 while achieving a hashing failure probability of 2^{-40}.

3 Private Set Union via Shuffling

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

3.1 Protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: Shuffling Receiver's Set

The first protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ is designed under the framework in Figure 2(a). Our basic idea is to realize the functionality g-RPMT 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 $X = \{x_1, \cdots, x_{n_1}\}$ and the receiver's set is $Y = \{y_1, \cdots, y_{n_2}\}$. 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 γ hash functions $h_1, \cdots, h_\gamma$. 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 $\mathcal{F}_{\mathsf{PS}}$ with inputs Y_C and π randomly chosen by the receiver. After this, the sender and receiver obtain the shares $\{a'_1, a'_2, \cdots, a'_b\}$ and $\{a_1, a_2, \cdots, a_b\}$ respectively, where $a'_i \oplus a_i = Y_C[\pi(i)]$. Further through $\mathcal{F}_{\mathsf{mpOPRF}}$ with $\{a_i\}_{i \in [b]}$ as the input, the sender receives PRF key k and the receiver obtains $\{F(k, a_i)\}_{i \in [b]}$ where $F(k, a_i) \in \{0,1\}^{\ell_2}$.

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[8]. To avoid this leakage, the sender permutes his set X to $\pi'(X) = \{x'_1, \cdots, x'_{n_1}\}$ by a randomly chosen permutation π', 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 $x'_i \in \{Y_C[h_1(x'_i)], \cdots, Y_C[h_\gamma(x'_i)]\}$. To do so, the sender first uses the permutation π of $\mathcal{F}_{\mathsf{PS}}$ to identify the shares of $Y_C[h_1(x'_i)], \cdots, Y_C[h_\gamma(x'_i)]$ from $\{a'_1, a'_2, \cdots, a'_b\}$, say $\{a'_{q_1}, a'_{q_2}, \cdots, a'_{q_\gamma}\}$, where $q_j = \pi^{-1}(h_j(x'_i))$ for $j \in [\gamma]$. Then he computes I_i as $I_i = \{F(k, x'_i \oplus a'_{q_1}), \cdots, F(k, x'_i \oplus a'_{q_\gamma})\}$. However, we notice that if there are distinct hash functions, say h_{j_s} and h_{j_t} s.t. $h_{j_s}(x'_i) = h_{j_t}(x'_i)$, then we have $I_i[j_s] = I_i[j_t]$, from which the receiver may learn partial information about x'_i. To overcome this shortcoming, I_i is generated in a slightly different way: $I_i[j_s]$ is computed and recorded in I_i as before, but $I_i[j_t]$ is replaced with a random value $r \xleftarrow{\$} \{0,1\}^{\ell_2}$. 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 $\{a'_1, \cdots, a'_b\}$) 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

[8]For example, assuming that X consists of the ages of a group 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ using Cuckoo hashing

Parameters:

- Hash functions $h_1, \cdots, h_\gamma : \{0,1\}^{\ell_1} \to [b]$;
- A Cuckoo hash table based on $h_1, \cdots, h_\gamma$ with $b = \varepsilon \cdot n_2$ bins, stash size $s = 0$;
- Ideal functionalities $\mathcal{F}_{\mathsf{PS}}$, $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$ (the underlying PRF is $F(k, \cdot) : \{0,1\}^{\ell_1} \to \{0,1\}^{\ell_2}$);

Inputs:

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

Protocol:

1. $\mathcal{R}$ inserts set Y into the Cuckoo hash table based on $h_1, \cdots, h_\gamma$ as shown in Section 2.2, and adds a dummy item d in each empty bin, then denotes the filled Cuckoo hash table as Y_C and the item in i-th bin as $Y_C[i]$;
2. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{PS}}$:
   - $\mathcal{R}$ acts as P_0 with input set Y_C, and $\mathcal{S}$ acts as P_1 with a permutation π;
   - $\mathcal{R}$ obtains the shuffled shares $\{a_1, a_2, \cdots, a_b\}$, and $\mathcal{S}$ obtains the other shuffled shares $\{a'_1, a'_2, \cdots, a'_b\}$ where $Y_C[\pi(i)] = a'_i \oplus a_i$;
3. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{mpOPRF}}$:
   - $\mathcal{R}$ acts as P_0 with her shuffled shares $\{a_i\}_{i \in [b]}$, and obtains the outputs $\{F(k, a_i)\}_{i \in [b]}$;
   - $\mathcal{S}$ obtains the key k;
4. $\mathcal{R}$ initializes set $Z = \emptyset$, $\mathcal{S}$ randomly selects a permutation π', and obtains $\pi'(X) = \{x'_1, x'_2, \cdots, x'_{n_1}\}$;
5. For $i \in [n_1]$:
   - $\mathcal{S}$ initializes sets $Q_i = \emptyset$ and $I_i = \emptyset$;
   - For $j \in [\gamma]$:
     - $\mathcal{S}$ computes $q_j = \pi^{-1}(h_j(x'_i))$;
     - if $q_j \notin Q_i$,
       $Q_i = Q_i \cup \{q_j\}$, $I_i = I_i \cup \{F(k, x'_i \oplus a'_{q_j})\}$,
       else,
       $r \xleftarrow{\$} \{0,1\}^{\ell_2}$ and $I_i = I_i \cup \{r\}$;
   - $\mathcal{S}$ sends I_i to $\mathcal{R}$;
   - $\mathcal{R}$ checks if $\{F(k, a_j)\}_{j \in [b]} \cap I_i \neq \emptyset$. If so, $\mathcal{R}$ sets $b_i = 1$, otherwise, sets $b_i = 0$;
   - $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{OT}}$:
     - $\mathcal{S}$ acts as P_0 with input $\{x'_i, \perp\}$;
     - $\mathcal{R}$ acts as P_1 with input b;
     - if $b_i = 0$, $\mathcal{R}$ obtains x'_i, otherwise, obtains $\perp$;
   - Once receiving x'_i, $\mathcal{R}$ sets $Z = Z \cup \{x'_i\}$;
6. $\mathcal{R}$ outputs $Y \cup Z$;

Figure 8: Protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ using Cuckoo Hashing.

$r \in \{0,1\}^{\ell_2}$ and records it into I_i. At the end, the sender sends I_i to the receiver. Recall that if $x'_i \in Y$, then there is an item in I_i that belongs to $\{F(k, a_i)\}_{i \in [b]}$.

Finally, upon receiving I_i, the receiver checks if the intersection of I_i and $\{F(k, a_j)\}_{j \in [b]}$ is non-empty. If not, the receiver sets $b_i = 0$ and obtains x_i through $\mathcal{F}_{\mathsf{OT}}$ and adds it to an initially empty set Z, otherwise sets $b_i = 1$ and obtains nothing. At last, the receiver outputs $Y \cup Z$.

Next we first argue that the protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ in Figure 8 realizes the functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}}$ correctly, and then show it satisfies the security properties.

Correctness. The receiver obtains the Cuckoo hash table Y_C filled with the set Y and dummy items d. Then through $\mathcal{F}_{\mathsf{PS}}$, the receiver obtains the shuffled secret share set $\{a_1, \cdots, a_b\}$ of Y_C and the sender receives the other shares $\{a'_1, \cdots, a'_b\}$, where $a'_i \oplus a_i = Y_C[\pi(i)]$. For an item $x^* \in X$, if $x^* \in Y$, say $x^* = y_i$, then y_i must be inserted into one of the bins of Y_C indexed by $\{h_j(x^*)\}_{j \in [\gamma]}$, and so the share of y_i held by the sender must belong to $\{a'_{q_1}, \cdots, a'_{q_\gamma}\}$ where $q_j = \pi^{-1}(h_j(x^*))$ for all $j \in [\gamma]$. Without loss of generality, we assume that the share of y_i is a'_{q_w}, then $x^* \oplus a'_{q_w} = y_i \oplus a'_{q_w} = a_{q_w}$, and thus $F(k, x^* \oplus a'_{q_w}) = F(k, a_{q_w})$. So in this case the intersection of $I^* = \{F(k, x^* \oplus a'_{q_j})\}_{j \in [\gamma]}$ and $\{F(k, a_i)\}_{i \in [b]}$ is non-empty, the receiver sets $b^* = 1$ and receives nothing from $\mathcal{F}_{\mathsf{OT}}$. Otherwise (i.e., $x^* \notin Y$), we have $x^* \oplus a'_{q_j} \neq a_{q_j}$ for all $j \in [\gamma]$. Moreover, for any a_t where $t \in [b] \setminus \{q_j\}_{j \in [\gamma]}$, $x^* \oplus a'_{q_j} \neq a_t$ with an overwhelming probability, as long as the length ℓ_1 of the share is sufficiently large. Thus the intersection of $\{F(k, x^* \oplus a'_{q_j})\}_{j \in [\gamma]}$ and $\{F(k, a_i)\}_{i \in [b]}$ is empty except for a negligible probability, then the receiver will set $b^* = 0$ and receive x^* through $\mathcal{F}_{\mathsf{OT}}$.

We remark that the correctness error comes from the following two types of collisions. Specifically, the first type is incurred by the secret shares, that is, for $x^* \notin Y$ and some $j \in [\gamma]$, $x^* \oplus a'_{q_j} \in \{a_1, \cdots, a_b\}$ holds. The other case is incurred by PRF, that is $F(k, x^* \oplus a'_{q_j}) = F(k, a_t)$ for some $x^* \oplus a'_{q_j} \neq a_t$. To make the correctness hold with an overwhelming probability, we need to ensure the probability of collisions happening is less than $2^{-\lambda}$. To this end, we set both the share length ℓ_1 and the PRF output length ℓ_2 to be at least $\lambda + \log(\varepsilon n_2) + \log(\gamma n_1)$.

Security. Now we proceed to show the semi-honest security of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ in the $\{\mathcal{F}_{\mathsf{PS}}, \mathcal{F}_{\mathsf{mpOPRF}}, \mathcal{F}_{\mathsf{OT}}\}$-hybrid model.

Theorem 3.1. *The protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ presented in Figure 8 securely realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}}$ in the $\{\mathcal{F}_{\mathsf{PS}}, \mathcal{F}_{\mathsf{mpOPRF}}, \mathcal{F}_{\mathsf{OT}}\}$-hybrid model, in the presence of semi-honest adversaries.*

Proof (sketch). We construct $\mathsf{Sim}_{\mathcal{S}}$ and $\mathsf{Sim}_{\mathcal{R}}$ to simulate the views of corrupted sender $\mathcal{S}$ and corrupted receiver $\mathcal{R}$, respectively. Roughly speaking, $\mathsf{Sim}_{\mathcal{S}}$ randomly chooses $\{a'_1, \cdots, a'_b\}$ and a key k as the outputs of $\mathcal{F}_{\mathsf{PS}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$, and receives the input of $\mathcal{F}_{\mathsf{OT}}$. Then $\mathsf{Sim}_{\mathcal{S}}$

can leverage the simulators of these subroutine functionalities to simulate the view of corrupted sender $\mathcal{S}$. On the other hand, $\mathsf{Sim}_{\mathcal{R}}$ also randomly chooses $\{a_1, \cdots, a_b\}$ and $\{F(k,a_1), \cdots, F(k,a_b)\}$ as the outputs of $\mathcal{F}_{\mathsf{PS}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$. When constructing $\{I_i\}_{i\in[n_1]}$, $\mathsf{Sim}_{\mathcal{R}}$ needs to simulate that there are $|X\cap Y|$ items in $\{I_i\}_{i\in[n_1]}$ that have an intersection with $\{F(k,a_1), \cdots, F(k,a_b)\}$. Once receiving b_i for $\mathcal{F}_{\mathsf{OT}}$, if $b_i = 0$, $\mathsf{Sim}_{\mathcal{R}}$ takes one item from $\{X\cup Y\} - Y$ in random order as the output, otherwise, $\mathsf{Sim}_{\mathcal{R}}$ takes $\perp$ as the output. Likewise, $\mathsf{Sim}_{\mathcal{S}}$ can leverage the simulators of these subroutine functionalities to simulate the remaining view of corrupted receiver $\mathcal{R}$. Due to lack of space, more details of the proof are given in the full version [16]. □

The protocol $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ as below.

3.2 Protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$: Shuffling Sender's Set

In contrast to $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, the core idea of designing $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 $X = \{x_1, \cdots, x_{n_1}\}$ and $Y = \{y_1, \cdots, y_{n_2}\}$, respectively. Then the protocol works as follows. Firstly, the sender inserts X into the Cuckoo hash table with b bins by using γ hash functions $\{h_1, \cdots, h_\gamma\}$ 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 by using the same hash functions. In general, when n items are inserted into a simple hash table with m bins using γ hash functions, the maximum bin size is $O(\gamma n/m)$[9] when $n > m\log m$ according to [35]. Therefore, the maximum bin size ρ of simple hash table is $O(\gamma n_2/b)$. 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 $\mathcal{F}_{\mathsf{PS}}$ with input X_C and a random permutation π, respectively. Then the sender obtains the shuffled secret shares $\{a_1, \cdots, a_b\}$ of X while the receiver obtains other shares $\{a'_1, \cdots, a'_b\}$. Further, by running $\mathcal{F}_{\mathsf{mpOPRF}}$ 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 as the receiver selects the permutation π, 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

[9] To be precise, the maximum bin size should be $\Theta(\gamma n/m)$.

$\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ Protocol using Cuckoo hashing

Parameters:

- Hash functions $h_1, \cdots, h_\gamma : \{0,1\}^{\ell_1} \to [b]$;
- A Cuckoo hash table based on $h_1, \cdots, h_\gamma$, with $b = \varepsilon \cdot n_1$ bins, stash size $s = 0$;
- A simple hash table based on $h_1, \cdots, h_\gamma$, with $b = \varepsilon \cdot n_1$ bins and bin size ρ, where $\rho = O(\gamma n_2/b)$;
- Ideal functionalities $\mathcal{F}_{\mathsf{PS}}$ and $\mathcal{F}_{\mathsf{mpOPRF}}$ (the underlying PRF is $F(k,\cdot) : \{0,1\}^{\ell_1} \to \{0,1\}^{\ell_2}$);

Inputs:

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

Protocol:

1. $\mathcal{S}$ inserts set X into the Cuckoo hash table, and fills empty bins with the dummy item d, then denotes the filled Cuckoo hash table as X_C and the item in i-th bin as $X_C[i]$; $\mathcal{R}$ inserts set Y into the simple hash table, and deletes the duplicates in each bin, then denotes the set of items in the i-th bin as $Y_B[i]$;
2. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{PS}}$:
   - $\mathcal{S}$ acts as P_0 with input set X_C, and $\mathcal{R}$ acts as P_1 with a permutation π;
   - $\mathcal{S}$ obtains the shuffled shares $\{a_1, a_2, \cdots, a_b\}$, $\mathcal{R}$ obtains the another shuffled shares $\{a'_1, a'_2, \cdots, a'_b\}$ where $X_C[\pi(i)] = a'_i \oplus a_i$;
3. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{mpOPRF}}$:
   - $\mathcal{S}$ acts as P_0 with his shuffled shares $\{a_i\}_{i\in[b]}$, and obtains the outputs $\{F(k,a_i)\}_{i\in[b]}$;
   - $\mathcal{R}$ obtains the key k;
4. $\mathcal{R}$ initializes a set $Z = \emptyset$, $\mathcal{S}$ initializes a string $U = 0^b$;
5. For $i \in [b]$:
   - $\mathcal{R}$ initializes a set $I_i = \emptyset$;
   - For each $y_j \in Y_B[\pi(i)]$, $\mathcal{R}$ adds $F(k, y_j \oplus a'_i)$ to I_i;
   - $\mathcal{R}$ pads I_i up to bin size ρ by different $r \xleftarrow{\$} \{0,1\}^{\ell_2}$, and sends I_i to $\mathcal{S}$;
   - $\mathcal{S}$ checks if $F(k,a_i)$ is in I_i, if not, $\mathcal{S}$ sets $U[i] = 1$, otherwise, sets $U[i] = 0$;
6. $\mathcal{S}$ and $\mathcal{R}$ invoke the ideal functionality $\mathcal{F}_{\mathsf{PS}}$:
   - $\mathcal{R}$ acts as P_0 with input set $\{a'_i\}_{i\in[b]}$, and $\mathcal{S}$ acts as P_1 with a random permutation π';
   - $\mathcal{S}$ and $\mathcal{R}$ obtains the shuffled share sets $\{s^1_1, s^1_2, \cdots, s^1_b\}$ and $\{s^2_1, s^2_2, \cdots, s^2_b\}$ respectively, where $s^1_i \oplus s^2_i = a'_{\pi'(i)}$;
7. For $i \in [b]$:
   - If $U[\pi'(i)] = 1$, $\mathcal{S}$ sets $z_i = a_{\pi'(i)} \oplus s^1_i$, otherwise, sets $z_i = \perp$, then sends z_i to $\mathcal{R}$;
   - If $z_i \neq \perp$ and $z_i \oplus s^2_i \neq d$, $\mathcal{R}$ sets $Z = Z \cup \{z_i \oplus s^2_i\}$;
8. $\mathcal{R}$ outputs $Y \cup Z$;

Figure 9: Protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ using Cuckoo Hashing.

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 $\rho - |Y_B[\pi(i)]|$ random values from $\{0,1\}^{\ell_2}$ if the number (i.e., $|Y_B[\pi(i)]|$) of items in I_i is less than the maximum bin size ρ. Note that if there is a y_j that is mapped to the bin multiple times using different hash functions, $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 $\perp$. 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 $\perp$, 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 $\mathcal{F}_{\mathsf{PS}}$ again. Roughly speaking, the receiver takes her shares $\{a'_1, a'_2, \cdots, a'_b\}$ as the input and the sender randomly chooses a permutation π' as his input. After $\mathcal{F}_{\mathsf{PS}}$, the sender receives the shuffled shares $\{s^1_1, s^1_2, \cdots, s^1_b\}$, and the receiver obtains the other shares $\{s^2_1, s^2_2, \cdots, s^2_b\}$ where $s^1_i \oplus s^2_i = a'_{\pi'(i)}$. As the sender knows the permutation π', he can check if $a_{\pi'(i)}$ should be sent to the receiver according to $U[\pi'(i)]$. If so, the sender sends $z_i = a_{\pi'(i)} \oplus s^1_i$ to the receiver, otherwise sends $z_i = \perp$. Once receiving z_i from the sender, if it is not $\perp$, the receiver can calculate $z_i \oplus s^2_i = a_{\pi'(i)} \oplus s^1_i \oplus s^2_i = a_{\pi'(i)} \oplus a'_{\pi'(i)}$. If the recovered item is not the dummy item d, 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 $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ in Figure 9 and then argue that it securely realizes the functionality $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}^*}$.

Correctness. The analysis is similar to that of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, and we show it in the full version [16] due to the limited space.

Security. Now we show that $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ securely realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}^*}$.

Theorem 3.2. *The protocol $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ presented in Figure 9 securely realizes $\mathcal{F}^{n_1,n_2}_{\mathsf{PSU}^*}$ in the $\{\mathcal{F}_{\mathsf{PS}}, \mathcal{F}_{\mathsf{mpOPRF}}\}$-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 λ, according to [34], we choose the parameters of Cuckoo hashing without stash, exactly including $b = \varepsilon \cdot n$ bins (where $n = n_2$ for $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $n = n_1$ for $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$) and γ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, we set the share/item length ℓ_1 and the output length ℓ_2 of $F(k,\cdot)$ to be at least $\lambda + \log(\gamma n_1) + \log(\varepsilon n_2)$. Likewise, the output length ℓ_2 of $F(k,\cdot)$ in $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ is at least $\lambda + \log(\varepsilon n_1) + \log(\gamma n_2)$. In addition, the costs of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ also rely on the sub-protocols used to realize the building blocks. Particularly, we realize $\mathcal{F}_{\mathsf{PS}}$, $\mathcal{F}_{\mathsf{mpOPRF}}$ and $\mathcal{F}_{\mathsf{OT}}$ with the protocols in [22], [6] and [15], respectively.

Table 4: The costs of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$

	Part-1	Part-2	Part-3	Part-4
Comp.	$O(b \log b)$	$O(b)$	$O(\gamma n)$	$O(n_1)$
Comm.	$O(\ell_1 b \log b)$	$O(b)$	$O(\ell_2 \gamma n)$	$O(\ell_1 n_1)$

$\Pi^{\mathsf{R}}_{\mathsf{PSU}}$: $b = \varepsilon n_2$, $n = n_1$; $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$: $b = \varepsilon n_1$, $n = n_2$ and Part-1 is executed twice.

Since $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ mainly consist of 4 parts: (1) Permute + Share, (2) multi-point OPRF, (3) computing and sending $\{I_i\}$, and (4) obtaining items in $X \setminus Y$, we for clarity summarize their complexities in Table 4 according to each part. In $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ (resp. $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$), 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 $b = \varepsilon n_2, n = n_1$ (resp. $b = \varepsilon n_1, n = n_2$).

4 Performance Evaluation

In this section, we experimentally evaluate our PSU protocols $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$. 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ in the cases where the sender's set is larger than the receiver's set, and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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

		Protocol	set size n 2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
Time (s)	WAN	[19]	1.064	1.379	2.164	5.326	17.541	86.358	333.073	1459.539
		Π^R_{PSU}	**0.671**	**0.892**	**1.132**	**1.778**	**4.412**	**16.104**	**67.756**	**341.758**
		Π^S_{PSU}	0.712	0.993	1.238	2.214	6.233	22.78	102.039	458.731
	LAN	[19]	0.578	0.69	1.278	3.551	13.285	69.19	263.476	1191.703
		Π^R_{PSU}	**0.265**	**0.308**	**0.412**	**0.87**	**2.702**	**10.751**	**48.703**	**251.091**
		Π^S_{PSU}	0.274	0.32	0.434	1.051	3.452	13.382	60.16	279.97
Comm.(MB)		[19]	0.41	1.86	7.72	31.8	131.17	600.62	2470.11	10233.28
		Π^R_{PSU}	**0.22**	**0.814**	**3.576**	**15.848**	**70.198**	**307.192**	**1338.79**	**5779.599**
		Π^S_{PSU}	0.376	1.554	7.019	31.381	140.604	617.654	2725.932	11746.69

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

		$\hat{\Pi}^R_{PSU}$				$\hat{\Pi}^S_{PSU}$			
		2^8	2^{10}	2^{12}	2^{14}	2^8	2^{10}	2^{12}	2^{14}
Time (s)	T=1	0.620	5.582	89.862	1423.955	0.526	5.827	86.037	1425.376
	T=2	0.432	3.108	45.295	722.29	0.358	2.862	44.967	719.325
	T=4	0.356	1.722	23.094	363.270	0.295	1.861	22.231	360.131
	T=8	0.349	1.067	11.713	183.181	0.261	0.986	11.640	183.838
Comm. (MB)		0.665	9.624	162.759	2828.476	0.641	9.588	162.63	2827.972

Table 6: Runtime (in seconds) and communication (in MB) of $\hat{\Pi}^R_{PSU}$ and $\hat{\Pi}^S_{PSU}$ 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.

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 ℓ_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 Π^R_{PSU}, Π^S_{PSU} and [19] in terms of runtime and communication, and the results are reported in Table 5. More concretely, compared to [19], our Π^R_{PSU} 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 Π^S_{PSU} in this comparison, it is worth noting that Π^S_{PSU} 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, Π^S_{PSU} has to execute the Permute + Share sub-protocol twice (cf. Section 3.2 for more details). Therefore, the runtime of Π^S_{PSU} is longer than that of Π^R_{PSU}. Nevertheless, compared to [19], our Π^S_{PSU} 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 Π^S_{PSU} 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 sets 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^{22}, Π^R_{PSU} and Π^S_{PSU} 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 $\hat{\Pi}^R_{PSU}$ and $\hat{\Pi}^S_{PSU}$ can be executed on small sets directly without Cuckoo hashing (please see the full version [16] for more details of $\hat{\Pi}^R_{PSU}$ and $\hat{\Pi}^S_{PSU}$). 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

		set size n					
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
WAN	T=1	1.132	1.778	4.412	16.104	67.756	341.758
	T=2	1.127	1.658	3.315	11.025	48.321	230.218
	T=4	1.117	1.553	2.965	8.852	37.847	181.657
	T=8	0.957	1.512	2.626	7.666	34.701	163.82
LAN	T=1	0.412	0.87	2.702	10.751	48.703	251.091
	T=2	0.367	0.615	1.721	6.221	29.812	148.538
	T=4	0.351	0.489	1.256	3.96	21.272	107.298
	T=8	0.325	0.477	1.093	3.582	14.304	71.782
Speedup		**1.31-1.26×**	**1.18-1.95×**	**1.68-2.47×**	**2.10-3.00×**	**1.95-3.40×**	**2.08-3.49×**

Table 7: Scaling of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ with set size ($n_1 = n_2 = n$) and number of threads ($T \in \{1,2,4,8\}$) in WAN/LAN settings.

		set size n					
		2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
WAN	T=1	1.238	2.214	6.233	22.78	102.039	458.731
	T=2	1.368	1.984	4.731	16.346	77.137	347.897
	T=4	1.388	1.79	3.909	13.856	65.319	292.226
	T=8	1.196	1.711	3.504	12.041	59.736	258.244
LAN	T=1	0.434	1.051	3.452	13.382	60.16	279.97
	T=2	0.378	0.764	2.322	7.863	38.434	175.485
	T=4	0.356	0.614	1.685	5.632	25.842	116.678
	T=8	0.408	0.606	1.397	5.204	20.992	96.723
Speedup		**1.04-1.06×**	**1.29-1.73×**	**1.78-2.47×**	**1.89-2.57×**	**1.70-2.87×**	**1.78-2.89×**

Table 8: Scaling of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ with set size ($n_1 = n_2 = n$) and number of threads ($T \in \{1,2,4,8\}$) in WAN/LAN settings.

	LAN						WAN					
n_2 (resp. n_1)	2^8			2^{12}			2^8			2^{12}		
n_1 (resp. n_2)	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}	2^{16}	2^{20}	2^{24}
$\Pi^{\mathsf{R}}_{\mathsf{PSU}}$	0.487	3.17	47.788	0.524	3.648	51.513	1.266	5.101	64.109	1.396	5.341	67.802
$\Pi^{\mathsf{S}}_{\mathsf{PSU}}$	0.511	2.918	44.606	0.581	2.958	48.379	1.042	3.759	51.043	1.406	3.789	57.352

Table 9: Runtime (in seconds) of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ for unbalanced set sizes ($n_1 \in \{2^{16}, 2^{20}, 2^{24}\}, n_2 \in \{2^8, 2^{12}\}$) and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ for unbalanced set sizes ($n_1 \in \{2^8, 2^{12}\}, n_2 \in \{2^{16}, 2^{20}, 2^{24}\}$) with 8 threads in WAN/LAN settings.

threads, and the communication is so much that it will affect the protocol executed in the WAN setting. Therefore, for large sets, we test the runtime and communication of the protocols with Cuckoo hashing.

The cases with large sets. In both $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, PRF values in set I 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ by evaluating it on the large set sizes $n_1 = n_2 = n \in \{2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}\}$ in parallel with $T \in \{1,2,4,8\}$ threads. Table 7 shows the experimental results of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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× in WAN setting and 3.49× in LAN setting. Similarly, we report the results of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ in Table 8. On the whole, as the set sizes and number of threads increase, the runtime of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ changes in the same way as that of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$. However, the speedup of $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ is less than that of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, since $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ performs Permute + Share sub-protocol twice, which is not completely parallelized.

4.4 Design for Unbalanced Datasets

In this section, we show that $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ to obtain the union. We can see that for the sender's set of size 2^{24}, $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ 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., $n_1 \ll n_2$), $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ can obtain a better performance since it replaces OT related to the larger set with Permute + Share just related to the smaller set.

Table 10: The probability of Case_2 for different set sizes

parameters	set size n							
	2^8	2^{10}	2^{12}	2^{14}	2^{16}	2^{18}	2^{20}	2^{22}
α	0.043	0.055	0.05	0.053	0.058	0.052	0.06	0.051
$\Pr(\times 10^{-11})$	7.946	1270	206.1	639.4	3252	444.8	5778	305.1

Here, αn is the number of bins.

5 Leakage Analysis of Protocol in [19]

In this section, we first recall the optimization via bucketing in [19], and then explain in detail why the usage of bucketing technique will leak the intersection information. Please refer to Appendix B.1 for more details of the protocol in [19]. Also, we further explain why the protocol in [19] cannot benefit from Cuckoo hashing in Appendix B.2.

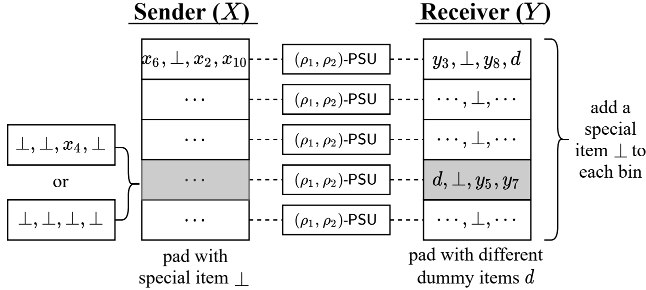


Figure 10: The bucketing technique in [19].

Optimization via bucketing. 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 ρ_1 and ρ_2, respectively. Then they perform the (ρ_1, ρ_2)-PSU sub-protocol on the items of each bin separately. As pointed out by Kolesnikov et al. in [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 (ρ_1, ρ_2)-PSU. By adding the special item $\perp$ to both sides, if the receiver learns that an item from the sender belongs to $\{y_3, \perp, y_8, d\}$, it seems that the receiver cannot know whether the item is a real item (namely, in X) or the special item $\perp$. Unfortunately, we observe that this strategy is insufficient to avoid the leakage incurred by the bucketing technique. A detailed analysis is given below.

Leakage analysis. For ease of exposition, we take the 4th (ρ_1, ρ_2)-PSU sub-protocol in Figure 10 as an example to explain why the optimization in [19] fails to hide the intersection information. After the execution of the sub-protocol over the 4th bins, if the receiver does not obtain any items from the sender (that is, all items in the sender's 4th bin belong to the subset in the receiver's 4th bin i.e., $\{d, \perp, y_5, y_7\}$), then the receiver could obtain additional information about the intersection. Concretely, one of the following will occur:

- Case_1: all the real items that are mapped to the sender's bin (say x_4 in Figure 10) belong to $\{y_5, y_7\}$;
- Case_2: no real items are mapped to the sender's bin (i.e., all items are special item $\perp$).

We denote the probability that Case_1 and Case_2 occur by $\Pr[\mathsf{Case}_1]$ and $\Pr[\mathsf{Case}_2]$, respectively. Clearly, if the receiver is able to determine that Case_1 occurs with certain (high) probability, she will know that items belonging to $X \cap Y$ are in $\{y_5, y_7\}$ with the same probability. Next we provide our estimation of the probability that Case_1 occurs. Note $\Pr[\mathsf{Case}_1] = 1 - \Pr[\mathsf{Case}_2]$; we now bound $\Pr[\mathsf{Case}_2]$. In general, assuming that there are αn bins and n items and the hash mapping is a random oracle, the probability Case_2 occurs is $P = \Pr[\mathsf{Case}_2] = (1 - \frac{1}{\alpha n})^n \approx e^{-1/\alpha}$.

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.

On the contrary, in our $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ shown in Figure 8, for an item x_i in $X \cap Y$, the receiver will find a $F(k, a_j) \in I_i$ where a_j is the share of $y_{\pi^{-1}(j)}$, which means that $y_{\pi^{-1}(j)}$ is the item equal to x_i. But from the receiver's point of view, any item in Y may corresponds to $F(k, a_j)$ as she does not know π, and thus any item in Y may be the item in $X \cap Y$. In our $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ shown in Figure 9, the receiver's set Y is inserted into a simple hash table as in [19], but our protocol does not suffer from the leakage analyzed before. This is because we use the shuffling technique to hide which bins contain items in $X \cap Y$. To sum up, in $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, any item in Y may be the item in $X \cap Y$, whereas in the protocol [19], the receiver can know that items

belonging to $X \cap Y$ are in a bin (namely, a subset of Y) with a 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 $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, 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 $\mathcal{F}_{\mathsf{OT}}$ the receiver obtains x_i if $b_i = 0$, and nothing otherwise. To obtain a PSI protocol from $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$, the receiver only needs to set $b'_i = b_i \oplus 1$ and obtains the sender's items through $\mathcal{F}_{\mathsf{OT}}$ 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 $\Pi^{\mathsf{R}}_{\mathsf{PSI}}$. In $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$, 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 $X \cap Y$ by flipping each bit in U, and then the receiver will obtain items in $X \cap Y$. Thus we obtain a PSI protocol, denoted by $\Pi^{\mathsf{S}}_{\mathsf{PSI}}$.

It can be seen that $\Pi^{\mathsf{R}}_{\mathsf{PSI}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSI}}$ 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, $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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 Permute + Share protocol can be designed with linear complexity, the cost of $\Pi^{\mathsf{R}}_{\mathsf{PSU}}$ and $\Pi^{\mathsf{S}}_{\mathsf{PSU}}$ 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

[1] SSL blacklist. https://sslbl.abuse.ch/blacklist/.

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

[3] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In Bimal K. Roy, editor, *ASIACRYPT 2005*, volume 3788 of *LNCS*, pages 236–252. Springer, Heidelberg, December 2005.

[4] Martin Burkhart, Mario Strasser, Dilip Many, and Xenofontas A. Dimitropoulos. SEPIA: Privacy-preserving aggregation of multi-domain network events and statistics. In *USENIX Security 2010*, pages 223–240. USENIX Association, August 2010.

[5] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. Secret-shared shuffle. In *ASIACRYPT 2020*, volume 12493 of *Lecture Notes in Computer Science*, pages 342–372, 2020.

[6] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In *CRYPTO 2020*, volume 12172 of *Lecture Notes in Computer Science*, pages 34–63, 2020.

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

[8] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In Joe Kilian, editor, *TCC 2005*, volume 3378 of *LNCS*, pages 303–324. Springer, Heidelberg, February 2005.

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

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

[11] Carmit Hazay and Yehuda Lindell. Efficient protocols for set intersection and pattern matching with security against malicious and covert adversaries. In Ran Canetti, editor, *TCC 2008*, volume 4948 of *LNCS*, pages 155–175. Springer, Heidelberg, March 2008.

[12] K. Hogan, N. Luther, N. Schear, E. Shen, D. Stott, S. Yakoubov, and A. Yerukhimovich. Secure multiparty computation for cooperative cyber risk assessment. In *2016 IEEE Cybersecurity Development (SecDev)*, pages 75–76, 2016.

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

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

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

[16] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. Cryptology ePrint Archive, Report 2022/157, 2022. https://ia.cr/2022/157.

[17] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 241–257. Springer, Heidelberg, August 2005.

[18] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 818–829. ACM Press, October 2016.

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

[20] Yehuda Lindell. How to simulate it - A tutorial on the simulation proof technique. Cryptology ePrint Archive, Report 2016/046, 2016. http://eprint.iacr.org/2016/046.

[21] Catherine Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134, 1986.

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

[23] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[24] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM STOC*, pages 245–254. ACM Press, May 1999.

[25] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-N OT extension with application to private set intersection. In Helena Handschuh, editor, *CT-RSA 2017*, volume 10159 of *LNCS*, pages 381–396. Springer, Heidelberg, February 2017.

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

[27] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. SpOT-light: Lightweight private set intersection from sparse OT extension. In Alexandra Boldyreva and Daniele Micciancio, editors, *CRYPTO 2019, Part III*, volume 11694 of *LNCS*, pages 401–431. Springer, Heidelberg, August 2019.

[28] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from PaXoS: Fast, malicious private set intersection. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 739–767. Springer, Heidelberg, May 2020.

[29] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 515–530. USENIX Association, August 2015.

[30] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure two-party computation is practical. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 250–267. Springer, Heidelberg, December 2009.

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

[32] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 125–157. Springer, Heidelberg, April / May 2018.

[33] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 797–812. USENIX Association, August 2014.

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

[35] Martin Raab and Angelika Steger. "balls into bins" — a simple and tight analysis. In Michael Luby, José D. P. Rolim, and Maria Serna, editors, *Randomization and Approximation Techniques in Computer Science*, pages 159–170, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

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

[37] Sivaramakrishnan Ramanathan, Jelena Mirkovic, and Minlan Yu. BLAG: improving the accuracy of blacklists. In *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020*. The Internet Society, 2020.

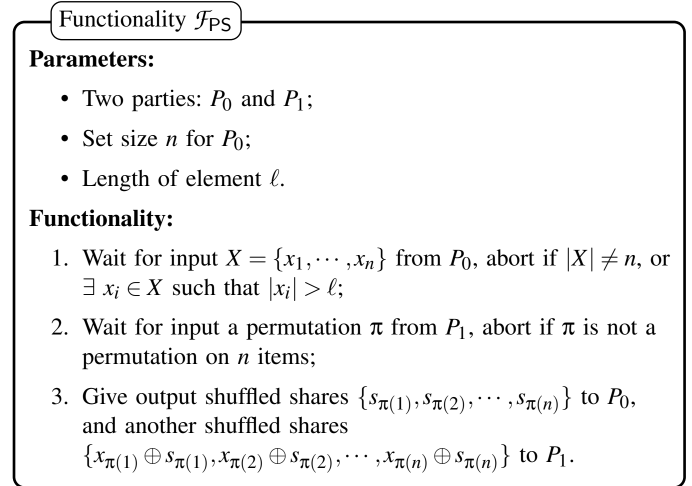
[38] Peter Rindal. libote: an efficient, portable, and easy to use oblivious transfer library. https://github.com/osu-crypto/libOTe.

[39] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017*, pages 1229–1242. ACM Press, October / November 2017.

[40] Peter Rindal and Phillipp Schoppmann. Vole-psi: Fast oprf and circuit-psi from vector-ole. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 901–930, Cham, 2021. Springer International Publishing.

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 $\mathcal{F}_{\mathsf{PS}}$ in Figure 11, multi-point OPRF functionality $\mathcal{F}_{\mathsf{mpOPRF}}$ in Figure 12, and 1-out-of-2 Oblivious Transfer functionality $\mathcal{F}_{\mathsf{OT}}$ in Figure 13.



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

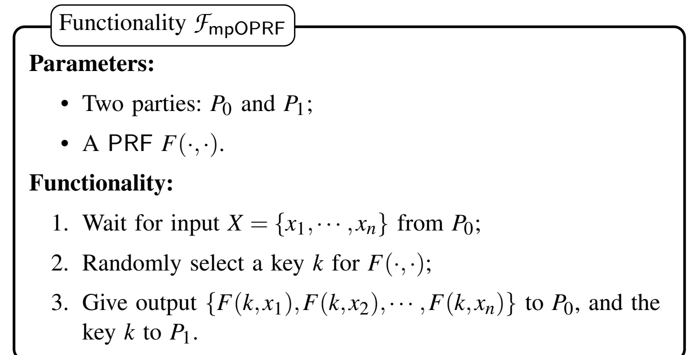
Parameters:

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

Functionality:

1. Wait for input $X = \{x_1, \cdots, x_n\}$ from P_0, abort if $|X| \neq n$, or $\exists\ x_i \in X$ such that $|x_i| > \ell$;
2. Wait for input a permutation π from P_1, abort if π is not a permutation on n items;
3. Give output shuffled shares $\{s_{\pi(1)}, s_{\pi(2)}, \cdots, s_{\pi(n)}\}$ to P_0, and another shuffled shares $\{x_{\pi(1)} \oplus s_{\pi(1)}, x_{\pi(2)} \oplus s_{\pi(2)}, \cdots, x_{\pi(n)} \oplus s_{\pi(n)}\}$ to P_1.

Figure 11: Permute+Share functionality.



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

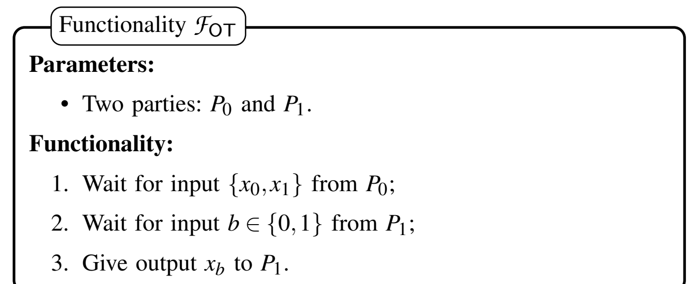
Parameters:

- Two parties: P_0 and P_1;
- A PRF $F(\cdot,\cdot)$.

Functionality:

1. Wait for input $X = \{x_1, \cdots, x_n\}$ from P_0;
2. Randomly select a key k for $F(\cdot,\cdot)$;
3. Give output $\{F(k,x_1), F(k,x_2), \cdots, F(k,x_n)\}$ to P_0, and the key k to P_1.

Figure 12: Multi-Point OPRF functionality.



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

Parameters:

- Two parties: P_0 and P_1.

Functionality:

1. Wait for input $\{x_0, x_1\}$ from P_0;
2. Wait for input $b \in \{0,1\}$ from P_1;
3. Give output x_b to P_1.

Figure 13: 1-out-of-2 Oblivious Transfer functionality.

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 $Y = \{y_1, \cdots, y_n\}$ in Figure 14. 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 $X = \{x_1, \cdots, x_{n_1}\}$ and $Y = \{y_1, \cdots, y_{n_2}\}$, respectively. Note that for each sender's item, both the secret value s 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 ℓ;
- Ideal functionalities $\mathcal{F}_{\mathsf{OT}}$ and $\mathcal{F}_{\mathsf{OPRF}}$ ($F(k,x) \in \{0,1\}^\sigma$);
- A collision-resistant hash function $h(x) : \{0,1\}^\ell \to \{0,1\}^\sigma$;

Inputs:

- Sender $\mathcal{S}$: $x^* \in \{0,1\}^\ell$;
- Receiver $\mathcal{R}$: set $Y = \{y_1, \cdots, y_n\}, y_i \in \{0,1\}^\ell$;

Protocol:

1. $\mathcal{S}$ acts as $\mathcal{F}_{\mathsf{OPRF}}$ receiver, sends x^* to $\mathcal{F}_{\mathsf{OPRF}}$, and $\mathcal{S}$ receives $q^* = F(k,x^*)$ and $\mathcal{R}$ receives k;
2. $\mathcal{R}$ randomly picks $s \xleftarrow{\$} \{0,1\}^\sigma$, and interpolates a polynomial $P(y)$ over points $\{(h(y_i), s \oplus q_i)\}$ where $q_i = F(k,y_i), \forall i \in [n]$. Here $s \oplus q_i$ is computed as operation on σ-bit strings.
3. $\mathcal{R}$ sends the coefficients of $P(y)$ to $\mathcal{S}$;
4. $\mathcal{S}$ computes $s^* = P(h(x^*)) \oplus q^*$ and sends it to $\mathcal{R}$;
5. $\mathcal{S}$ and $\mathcal{R}$ invoke $\mathcal{F}_{\mathsf{OT}}$:
   - $\mathcal{R}$ acts as receiver with input 1 if $s^* = s$ and input 0 otherwise;
   - $\mathcal{S}$ acts as sender with input $(x^*, \perp)$;
6. If $s^* = s$, then $\mathcal{R}$ 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 ℓ and $n = \max(n_1,n_2)$;
- Number of bins $\beta = \beta(n)$, hash function $H : \{0,1\}^\ell \to [\beta]$, and max bin size m;
- A special item $\perp \in \{0,1\}^*$;

Inputs:

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

Protocol:

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

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