The Ideal Functionalities for Private Set Union, Revisited

Yanxue Jia
Shanghai Jiao Tong University

Shi-Feng Sun
Shanghai Jiao Tong University

Hong-Sheng Zhou
Virginia Commonwealth University

Dawu Gu
Shanghai Jiao Tong University

June 13, 2022

Abstract

A Private Set Union (PSU) protocol allows parties, each holding an input set, to jointly compute the union of the sets without revealing anything else. In the literature, when we design scalable two-party PSU protocols, we follow the so-called "split-execute-assemble" paradigm, and also use Oblivious Transfer as a building block. Recently, Kolesnikov et al. (ASIACRYPT 2019) pointed out that security issues could be introduced when we design PSU protocols following the "split-execute-assemble" paradigm. Surprisingly, we observe that the typical way of invoking Oblivious Transfer also causes unnecessary leakage.

In this work, to enable a better understanding of the security for PSU, we provide a systematic treatment of the typical PSU protocols, which may shed light on the design of practical and secure PSU protocols in the future. More specifically, we define different versions of PSU functionalities to properly capture the subtle security issues arising from protocols following the "split-execute-assemble" paradigm and using Oblivious Transfer as subroutines. Then, we survey the typical PSU protocols, and categorize these protocols into three design frameworks, and prove what PSU functionality the protocols under each framework can achieve at best, in the semi-honest setting.

Contents

1 Introduction

In a Private Set Union (PSU) protocol, two players, a sender and a receiver, holding input sets X and Y, respectively, can jointly compute the union $X \cup Y$ as output. To ensure the joint computation is *private*, any additional information except the union $X \cup Y$, is not allowed to be learned by the players. Especially, information about the items in the intersection set $X \cap Y$ should not be learned by the players. Often we consider a simplified version of PSU: Instead of having both players to obtain the same output $X \cup Y$, in the simplified version of PSU, the receiver obtains the output $X \cup Y$, while the sender obtains, not the output $X \cup Y$, but a notification[1] indicating that the protocol execution is complete.[2] PSU, as a typical two-party secure computation task, can be used in multiple application scenarios, and efficient PSU protocols have been constructed in the literature [Fri07, DC17, KRTW19, GMR+21, JSZ+22].

"Split-execute-assemble" paradigm in PSU. In the context of developing Private Set Intersection (PSI) protocols, in order to improve the performance for large datasets, many constructions (e.g., [PSZ14, PSSZ15, KKRT16, PSZ18, PRTY19]) essentially follow a paradigm that we call "split-execute-assemble" paradigm: first the two large input datasets are **split** into multiple small input subset pairs; then multiple PSI instances are **executed** based on these *small* subset pairs respectively; finally the output results from these executed instances are **assembled** together, forming the final output of the PSI protocol execution (for the *large* input sets).

Very recently, Kolesnikov et al. [KRTW19] introduced for the first time this paradigm into the design of efficient PSU protocols:

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

The above is an oversimplified description of the "split-execute-assemble" paradigm; please find a complete description in Section 3.1.

Security concerns about PSU protocols under the "split-execute-assemble" paradigm.[3] Applying the above paradigm for designing PSU protocols is definitely a natural and interesting idea. However, even if the underlying subroutine, i.e. the PSU sub-protocol for the small-size input subsets, is UC-secure (i.e., it can UC-realize the standard PSU functionality in Section 4.1), it is unclear if the "assembled" PSU protocol is still UC-secure. We next discuss potential security concerns on the PSU protocols following the "split-execute-assemble" paradigm.

As already indicated in [KRTW19], in the above "split-execute-assemble" paradigm, the receiver can learn if a subset Y_i includes items that are in the intersection $X \cap Y$, which is not allowed in PSU. In order to conceal this leakage information, Kolesnikov et al. introduced a careful padding strategy in [KRTW19]. Subsequently, Jia et al. [JSZ+22] pointed out that their strategy is insufficient to address the leakage issue: Roughly, when observing the output $Z_i := X_i \cup Y_i$ is equal to Y_i for the i-th PSU sub-protocol instance, the receiver will know that the subset Y_i includes the items in $X \cap Y$ with an overwhelming probability (see Section 3.3 for more details). Nevertheless, a formal treatment of this leakage is still left open.

In this work, we revisit the problem pointed out in [JSZ+22]. We start with the definition of the ideal functionality for PSU, and observe that the leakage can be deduced from the *final* output $X \cup Y$; the standard PSU functionality reveals nothing to the receiver but the final output. Intuitively, this means that a UC-secure PSU protocol should allow the receiver to learn the above leakage *only after* obtaining the entire

[1] We remark that this notification is needed since in the real world protocol execution, the environment is indeed aware if the sender completes its execution or not.

[2] In the semi-honest setting, the sender can easily obtain the output from the receiver.

[3] We remark that, the very similar security concerns also occur in many PSI protocols under the "split-execute-assemble" paradigm. In this paper, we focus on the task of PSU only.

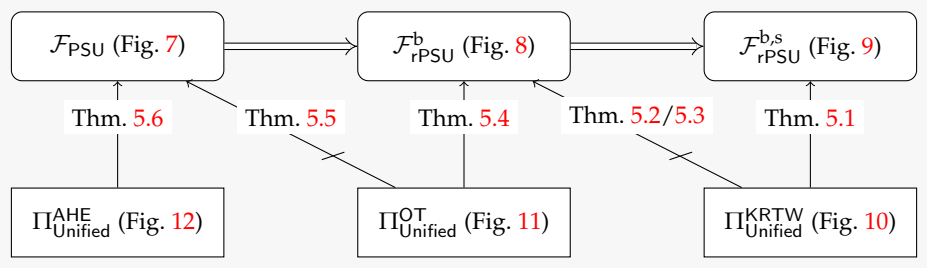

Figure 1: The relationship between the PSU functionalities/protocols. Here, "$\mathcal{F}_A \Longrightarrow \mathcal{F}_B$" means that if a protocol can UC-realize $\mathcal{F}_A$, then it can UC-realize $\mathcal{F}_B$; "$\Pi \longrightarrow \mathcal{F}$" means that the protocols following framework Π can UC-realize $\mathcal{F}$, and "$\Pi \not\longrightarrow \mathcal{F}$" means that the protocols following framework Π cannot UC-realize $\mathcal{F}$.

output $X \cup Y$. However, if the subroutine PSU instances are executed sequentially[4], the receiver obtains $Z_i := X_i \cup Y_i$ one by one, and will be able to learn (part of) the leakage information *before* obtaining all the items in $X \cup Y$. Therefore, to avoid leaking the information in advance, the subroutine PSU instances are supposed to be executed *at the same time*; then the "assembled" PSU protocol could be proven to be UC-secure when the underlying PSU sub-protocol is UC secure.

Security concerns about OT-based PSU protocols. We further observe that the PSU sub-protocol designed in [KRTW19] (as shown in Fig. 11) is not UC-secure. More specifically, for each item in the input set X, the receiver first learns whether it is a member of the set Y, and then obtains the output $Z := X \cup Y$ by invoking Oblivious Transfer (OT) instances with the membership information. In other words, the OT-based PSU sub-protocol in [KRTW19] leaks the membership information *before* the execution of the protocol is completed (see Section 3.3 for more details). This issue also exists in the most recent works [GMR+21, JSZ+22], although the security concerns arising from the "split-execute-assemble" paradigm can be effectively addressed.

AHE-based PSU protocols. In contrast to above works that mainly rely on symmetric-key techniques, Frikken et al. [Fri07] and Davidson et al. [DC17] proposed PSU protocols based on additively homomorphic encryption (AHE). The AHE-based PSU protocols *avoid* using the "split-execute-assemble" paradigm or OT, thus *not* suffering from the security concerns mentioned before, but they are much less efficient than the protocols based on symmetric-key primitives. The latter are promising in practical applications, but due to the above concerns it is unclear what security guarantees they can provide. Therefore, it is important to figure out the functionalities they can realize in practice. In this work, we provide a systematic treatment for understanding the security of the typical PSU protocols, and provably show what functionalities they can achieve in the semi-honest setting.

1.1 Our results

Motivated by the above discussions, we revisit the ideal functionality of PSU and develop several versions of PSU functionalities. Furthermore, we show what functionality the typical PSU protocols can achieve. Our findings are summarized in Fig. 1 and the details are described as below:

Defining ideal functionalities for PSU: Starting with the PSU functionality in [KRTW19], denoted as $\mathcal{F}^*_{\mathsf{PSU}}$, we develop different versions of PSU functionalities. Note that, in $\mathcal{F}^*_{\mathsf{PSU}}$, the interactions between the simulator (ideal adversary) and the functionality are not explicitly specified. In order to address the subtle issues in existing PSU protocols [KRTW19, GMR+21, JSZ+22], we explicitly specify these interactions in the description of PSU functionalities.

- The standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ (as in Fig. 7): Initially, both players provide inputs to the functionality. After receiving the input from each player, the simulator will be notified; to ensure the privacy,

[4]Notice that, it is not always feasible to execute all subroutine instances at the same time in practice, especially when the number of instances is large.

the input is not allowed to reveal to the simulator. When both input sets are ready, the simulator can enable the joint computation by issuing a "command" to the functionality, and the functionality at this point will return the union of the input sets to the receiver and return a notification to the sender. Note that, without explicitly presenting the interactions with the simulator, the PSU functionality is denoted as $\mathcal{F}^{*}_{\mathsf{PSU}}$ in [KRTW19].

- Next we define two relaxed PSU functionalities to capture the OT-based protocols in [GMR+21, JSZ+22] and the protocol in [KRTW19] that additionally leverages the "split-execute-assemble" paradigm, respectively.
  1. PSU functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ (as in Fig. 8) with set membership leakage: Different from the above standard version, the functionality does not return the union to the receiver directly. The simulator first issues a "command" to the functionality to enable a set membership test, which means that the functionality leaks a bit $b_i \in \{0,1\}$ for each $x_i \in X$ to the receiver; $b_i = 1$ if $x_i \in Y$, otherwise, $b_i = 0$. After that, the functionality will receive the simulator's "command" for computing union and return the union to the receiver and return a notification to the sender.
  2. PSU functionality $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ (as in Fig. 9) with set membership and subsets leakage : The functionality leaks more information than the above functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$. More specifically, the functionality first splits the pair of input sets into multiple much smaller pairs of subsets. The simulator will be notified when a subset of each input set is prepared; again, to ensure the privacy, the subset input is not allowed to be revealed to the simulator. When a pair of input subsets are ready, the simulator can enable the *set membership test for the pair of input subsets* by issuing a "command" to the functionality, and the functionality at this point will record the *bits for the pair of input subsets* within the functionality. Then, the simulator will issue another "command" to enable the *joint computation for the pair of input subsets*, and the functionality at this point will record the *union of the pair of input subsets* within the functionality. When all pairs of input subsets have been jointly computed, the functionality will return the union of the input sets to the receiver and return a notification to the sender.

Understanding the security of existing typical constructions: We classify the existing typical PSU protocols into three design frameworks. Then, for each framework, we prove what functionality it can realize.

- Design framework $\Pi^{\mathsf{KRTW}}_{\mathsf{Unified}}$ (as in Fig. 10) using OT and "split-execute-assemble" paradigm: Currently, only the PSU protocol in [KRTW19] follows this design framework. We prove that it can UC-realize the relaxed PSU functionality $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ in the semi-honest setting, as stated in Theorem 5.1. Moreover, we show that no matter whether the sub-protocols are executed simultaneously or sequentially, the PSU protocol in [KRTW19] cannot UC-realize the other relaxed PSU functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ that leaks less information in the semi-honest setting, as stated in Theorem 5.2 and Theorem 5.3.
- Design framework $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ (as in Fig. 11) only using OT: We unify the PSU protocols in [GMR+21, JSZ+22] and the basic PSU protocol in [KRTW19] into this design framework. Then we prove that the protocols following the design framework can UC-realize the relaxed PSU functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ but not the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ in the semi-honest setting, as stated in Theorem 5.4 and Theorem 5.5, respectively.
- Design framework $\Pi^{\mathsf{KRTW}}_{\mathsf{Unified}}$ (as in Fig. 12) using AHE: We observe that the PSU protocols in [Fri07, DC17] can be unified to this design framework. We prove that the protocols following this framework can UC-realize the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ in the semi-honest setting as stated in Theorem 5.6.

2 Preliminaries

In this section, we briefly recall the Universal Composability (UC) framework and the main building blocks used throughout this work, including Oblivious Transfer and the generalized Reversed Private Membership Test. Also, we recall Simple hashing and Cuckoo hashing in Appendix A, which are helpful for constructing scalable PSU protocols.

2.1 Universal Composability Framework

In secure multi-party computation (MPC), a set of parties jointly compute a function of their private inputs while preserving the privacy of each party's input. The security of an MPC protocol can be established by the ideal/real simulation paradigm. More concretely, the security requirements of a cryptographic task is defined through a *trusted party* who locally carries out the computation. In the ideal process, all parties send their private inputs to the trusted party, then obtain the prescribed outputs. A protocol in the real world is said to *securely realize* a task if it can be executed to "emulate" the ideal process for the task. That is, for any adversary attacking a real protocol execution, there is an adversary attacking the ideal process such that the real execution and the ideal process are indistinguishable.

In modern network settings, different protocol instances may be executed at the same time. Therefore, it is not enough to analyze the security of a protocol in the *stand-alone setting*, where a single set of parties run a single protocol instance in isolation. A protocol should remain secure when running with other protocol instances. The Universal Composability (UC) framework, introduced by Canetti in [Can00, Can01], is to analyze the security of cryptographic protocols under *arbitrary composition*. Next, we briefly describe the framework. Please refer to [Can00] for more details.

The real-world model. In the real world, there is a system of interactive Turing Machines (ITMs), including the execution of a protocol Π, an adversary $\mathcal{A}$, and an environment $\mathcal{E}$. More specifically, the protocol Π involves a set of parties (in this work, we focus on two-party PSU protocol, and thus there are two parties), the adversary $\mathcal{A}$ represents all the adversarial activities against the protocol execution, and the environment $\mathcal{E}$ represents all other protocol instances and adversaries. Once receiving inputs from $\mathcal{E}$, the parties execute the protocol Π and then hand outputs to $\mathcal{E}$. Adversary $\mathcal{A}$ can corrupt any party to get its internal state and control its behaviors. Moreover, the environment and the adversary are allowed to interact at any point throughout the course of the protocol execution, which represents the "flow of information" between the protocol instance under consideration and other protocol instances that are running concurrently. Finally, the environment will output one bit and halt. Let $\text{EXEC}_{\Pi,\mathcal{A},\mathcal{E}}$ denote the ensemble of random variables describing the environment $\mathcal{E}$'s output in the real world.

The ideal-world model. An ideal functionality is used to capture the desired functionality of the given task. In the ideal world, there is an environment $\mathcal{E}$, an ideal functionality $\mathcal{F}$, an ideal adversary (i.e., simulator) Sim and a set of dummy parties. Once receiving an input from $\mathcal{E}$, the dummy party directly forwards it to $\mathcal{F}$, and when $\mathcal{F}$ returns the output, the dummy party immediately outputs this value to $\mathcal{E}$. The adversary Sim can interact with $\mathcal{F}$, such that Sim can get the "allowed leakage of information" from $\mathcal{F}$ and have "allowed influence" on the computation of $\mathcal{F}$. In addition, Sim can send corrupt messages to $\mathcal{F}$ to corrupt parties. Once a party is corrupted, Sim will get its input and output. Let $\text{EXEC}_{\mathcal{F},\mathsf{Sim},\mathcal{E}}$ denote the ensemble of random variables describing the environment $\mathcal{E}$'s output in the ideal world.

Securely realizing an ideal functionality. A protocol Π can be determined whether to be UC-secure for a task $\mathcal{F}$ by the following definition:

Definition 2.1. *A protocol Π UC-realizes a task $\mathcal{F}$ if for any real-world adversary $\mathcal{A}$ that interacts with Π there exists an ideal-world simulator Sim that interacts with $\mathcal{F}$, such that for any environment $\mathcal{E}$ it holds that $EXEC_{\mathcal{F},\mathsf{Sim},\mathcal{E}} \overset{c}{\approx} EXEC_{\Pi,\mathcal{A},\mathcal{E}}$.*

The hybrid model. The hybrid model with a functionality $\mathcal{F}$, called $\mathcal{F}$-hybrid model, is similar to the real-world model, except that the parties may invoke an unbounded number of $\mathcal{F}$ subroutines. There are real messages communicated between parties and ideal messages used for oracle access to functionality $\mathcal{F}$.

Semi-honest Adversaries. In this work, we focus on semi-honest adversaries. A semi-honest adversary runs the protocol honestly, but may try to learn as much as possible from the messages received from other parties. Semi-honest adversaries are also considered passive, since they cannot take any actions other than attempting to learn private information by observing a view of a protocol execution. Semi-honest adversaries are also commonly called honest-but-curious.

Static Corruptions. The adversary corrupts parties before the protocol execution begins, and the set of corrupted parties is fixed throughout the course of the protocol execution.

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

Parameters:

- The functionality interacts with two parties, the sender P_0 and the receiver P_1, and the simulator Sim;
- Let n be the set size for the receiver P_1.

Functionality:

0. Initialize an ideal state $state_{\mathsf{U}} := \emptyset$ for party U where $\mathsf{U} \in \{P_0, P_1\}$; if U is corrupted, the simulator Sim is allowed to access U's state $state_{\mathsf{U}}$;
1. Upon receiving input set $X = \{x_1, \cdots, x_{n_1}\}$ from the sender P_0, update state $state_{p_0} := \langle X \rangle$, and send $\langle \text{REQUEST}, P_0 \rangle$ to the simulator Sim;
2. Upon receiving input set $Y = \{y_1, \cdots, y_n\}$ from the receiver P_1, update state $state_{P_1} := \langle Y \rangle$, and send $\langle \text{REQUEST}, P_1 \rangle$ to the simulator Sim;
3. Upon receiving $\langle \text{RESPONSE}, \text{OK} \rangle$ from Sim, for each $i \in [n_1]$, if $x_i \in Y$, set $b_i = 1$, otherwise, set $b_i := 0$; then add $\langle \text{FINISHED} \rangle$ to the sender's state $state_{P_0}$ and $\langle \{b_1, \cdots, b_{n_1}\} \rangle$ to the receiver's state $state_{P_1}$;
4. Output $\{b_1, \cdots, b_{n_1}\}$ to P_1, and FINISHED to P_0.

Figure 2: The generalized Reversed Private Set Membership Test functionality. Note that $\mathcal{F}_{\text{g-RPMT}}$ is equivalent with $\mathcal{F}_{\text{RPMT}}$ when $n_1 = 1$.

2.2 Building Blocks

Next, we recollect the main building blocks, including Oblivious Transfer and (generalized) Reversed Private Membership Test.

2.2.1 Oblivious Transfer

A 1-out-of-2 oblivious transfer (OT) is a two-party protocol, where the sender P_0 takes as input two strings $\{x_0, x_1\}$, and the receiver P_1 chooses a random bit $b \in \{0, 1\}$. After the protocol, P_1 obtains nothing other than x_b while P_0 learns nothing about b. The first OT protocol was proposed by Rabin in [Rab05]. And due to the lower bound in [IR89], all the OT protocols require expensive public-key operations. To improve the performance, Ishai et al. [IKNP03] introduced the concept of OT extension that enables us to carry out many OTs based on a small number of basic OTs. The functionality $\mathcal{F}_{\text{OT}}$ of OT is shown in Fig. 3.

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

- The functionality interacts with two parties, the sender P_0 and the receiver P_1, and the simulator Sim;
- Let ℓ be the bit-length of each item in the sender's input.

Functionality:

0. Initialize an ideal state $state_{\mathsf{U}} := \emptyset$ for party U where $\mathsf{U} \in \{P_0, P_1\}$; if U is corrupted, the simulator Sim is allowed to access U's state $state_{\mathsf{U}}$;
1. Upon receiving input (x_0, x_1) from the sender P_0 where $x_i \in \{0, 1\}^{\ell}$, update state $state_{p_0} := \langle (x_0, x_1) \rangle$, and send $\langle \text{REQUEST}, P_0 \rangle$ to the simulator Sim;
2. Upon receiving input b from the receiver P_1 where $b \in \{0, 1\}$, update state $state_{P_1} := \langle b \rangle$, and send $\langle \text{REQUEST}, P_1 \rangle$ to the simulator Sim;
3. Upon receiving $\langle \text{RESPONSE}, \text{OK} \rangle$ from Sim, add $\langle \text{FINISHED} \rangle$ to the sender's state $state_{P_0}$ and $\langle x_b \rangle$ to the receiver's state $state_{P_1}$;
4. Output x_b to P_1, and FINISHED to P_0.

Figure 3: 1-out-of-2 Oblivious Transfer functionality.

2.2.2 Generalized Reversed Private Membership Test

Reversed Private Membership Test (RPMT) was first proposed and formalized in [KRTW19]. More concretely, there are two parties, the sender P_0 holding an item x and the receiver P_1 holding a set Y. Then, the receiver P_1 can learn a bit b without obtaining any information else about item x; if $x \in Y$, $b = 1$, otherwise, $b = 0$. Meanwhile, the sender P_0 knows nothing about P_1's set Y. We denote the RPMT functionality as $\mathcal{F}_{\mathsf{RPMT}}$.

Based on the above RPMT, a generalized RPMT was proposed in [JSZ+22] where the sender P_0 inputs a set X, rather than an item x. Likewise, for each item $x_i \in X$, the receiver P_1 can learn a bit b_i without obtaining any information else about the item x_i; if $x_i \in Y$, $b_i = 1$, otherwise, $b_i = 0$. Meanwhile, the sender P_0 knows nothing about P_1's set Y. The functionality $\mathcal{F}_{\mathsf{g\text{-}RPMT}}$ is shown in Fig. 2.

We remark that given a protocol Π_{RPMT} that UC-realizes functionality the $\mathcal{F}_{\mathsf{RPMT}}$, executing Π_{RPMT} repeatedly can naturally achieve a generalized RPMT protocol $\Pi_{\mathsf{g\text{-}RPMT}}$ that UC-realizes functionality $\mathcal{F}_{\mathsf{g\text{-}RPMT}}$ as in [KRTW19]. Note that in each Π_{RPMT} execution, the set Y needs to be processed at least once. Therefore, in this way, the process on the set Y needs to be repeatedly performed, which is unacceptable in practice when both sets X and Y are large. Jia et al. [JSZ+22] and Garimella et al. [GMR+21] designed more efficient generalized RPMT protocols where the set Y only needs to be processed once; please find Fig. 18 and Fig. 19 in Appendix C.4 for more details.

3 KRTW-PSU: Review and Reflection

In this section, we recall the protocol in [KRTW19] where multiple PSU sub-protocol instances will be executed to support large datasets, as well as the PSU functionality defined in [KRTW19]. Moreover, we informally analyze the subtle issues in the protocol of [KRTW19], which motivate us to revisit the ideal functionality of PSU.

3.1 Overview of the protocol in [KRTW19]

In a PSU protocol, there are two players, a sender and a receiver; initially, the sender holds a set X, and the receiver holds a set Y; after a few rounds of communications/computations, eventually the receiver obtains the set $X \cup Y$ as her output. Note that, except for the union $X \cup Y$, the receiver is not allowed to learn anything else, especially any additional information about the items in $X \cap Y$, while the sender is not allowed to learn any output.

Of course, we can construct a PSU protocol based on any generic compiler for secure two-party computation (2PC), since PSU is a special case of 2PC. To achieve high performance, the protocol in [KRTW19], denoted as Π_{KRTW}, relies on a "split-execute-assemble" paradigm[5], as illustrated in Fig. 4:

- "**split**:" First, the pair of input sets X and Y are carefully split into multiple much smaller pairs of subsets, i.e., $\{(X_1, Y_1), (X_2, Y_2), \cdots, (X_\beta, Y_\beta)\}$, where $\beta \in \mathbb{N}$. Then, each pair of subsets (X_i, Y_i), will be "padded" into $(\tilde{X}_i, \tilde{Y}_i)$ by the sender and the receiver, respectively. Note that, here the padded subsets are all with the same predefined size $m \in \mathbb{N}$ (i.e., $|\tilde{X}_i| = |\tilde{Y}_i| = m$). The padding strategies from both players are slightly different: from the sender, each subset X_i will be augmented with multiple special items e up to the (predefined) size m; then the augmented subset will be randomly permuted, resulting in the subset $\tilde{X}_i$. From the receiver, each subset Y_i will be augmented with a *single* special item e along with multiple distinct dummy items $d_1, d_2, \ldots$, up to the (predefined) size m, resulting in the subset $\tilde{Y}_i$.
- "**execute**:" Second, the two parties execute β number of PSU sub-protocol instances: in the i-th instance, the sender and the receiver provide subset $\tilde{X}_i$ and subset $\tilde{Y}_i$ as their inputs respectively, and then the receiver obtains set $\tilde{Z}_i = \tilde{X}_i \cup \tilde{Y}_i$, as output. After discarding the special item e and dummy items from set $\tilde{Z}_i$, the receiver can obtain $Z_i = X_i \cup Y_i$.

[5] In [KRTW19], the paradigm is named as "bucketing" technique.

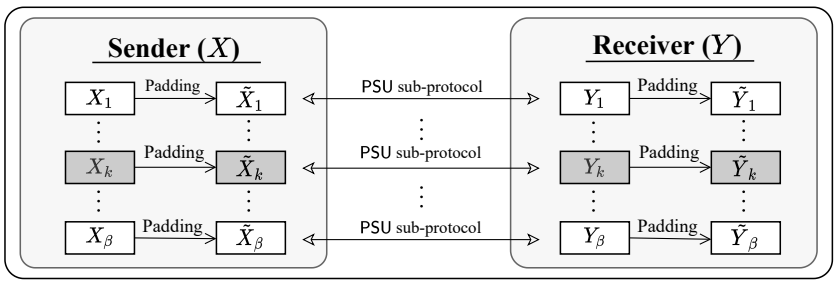

Figure 4: The overview of the protocol in [KRTW19]. The sender's set X and the receiver's set Y are split into multiple disjoint subsets $\{X_1, X_2, \cdots, X_\beta\}$ and $\{Y_1, Y_2, \cdots, Y_\beta\}$, respectively. Padding set X_i with several special items e can obtain set $\tilde{X}_i$. Padding set Y_i with a special item e and some different dummy items can obtain set $\tilde{Y}_i$. The two parties execute a PSU sub-protocol instance on the pair $(\tilde{X}_i, \tilde{Y}_i)$ and the receiver can obtain set $\tilde{Z}_i = \tilde{X}_i \cup \tilde{Y}_i$. After discarding the special item e and dummy items from $\tilde{Z}_i$, the receiver can obtain $Z_i = X_i \cup Y_i$ for each $i \in [\beta]$. Finally, the receiver obtains the output $Z = Z_1 \cup Z_2 \cup \cdots \cup Z_\beta = X \cup Y$. As discussed in Section 3.3, once finding $Z_k = Y_k$, the receiver can learn that Y_k has items belonging to $X \cap Y$ with overwhelming probability when the probability that $X_k \neq \emptyset$ is overwhelming.

- "**assemble**:" Finally, the receiver assembles the output (i.e., set Z_i) of each PSU sub-protocol, and obtains the output $Z := Z_1 \cup Z_2 \cup \cdots \cup Z_\beta$.

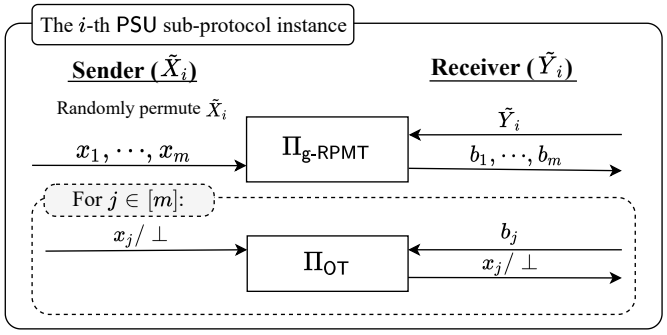

Figure 5: The design framework of PSU sub-protocol in [KRTW19].

Each PSU sub-protocol follows the design framework shown in Fig. 5. More specifically, the two parties first perform the "genaralized Reversed Private Membership Test" sub-protocol, denoted as $\Pi_{\text{g-RPMT}}$, such that for each item $x_j \in \tilde{X}_i$ the receiver will learn a bit b_j; if $x_j \in \tilde{Y}_i$, $b_j = 1$, otherwise, $b_j = 0$. Note that the sender learns nothing about set $\tilde{Y}_i$ and the receiver also obtains no more information about each item in $\tilde{X}_i$ than whether it belongs to $\tilde{Y}_i$. Obviously, if $x_j \notin \tilde{Y}_i$ (i.e., $b_j = 0$), the receiver should obtain the item x_j, otherwise learns nothing about the item x_j. To this end, the two parties perform the OT sub-protocol, denoted as Π_{OT}, for each item $x_j \in \tilde{X}_i$.

3.2 The PSU ideal functionality in [KRTW19]

In Fig. 6, we recall the ideal PSU functionality defined by Kolesnikov et al. in [KRTW19], and denote it as $\mathcal{F}^*_{\text{PSU}}$[6]. Intuitively, in PSU, the two parties, the sender with input set X and the receiver with input set Y can jointly compute the union set $X \cup Y$ as the output. Without loss of generality, in [KRTW19], only the receiver obtains the output $X \cup Y$, while the sender does not.

[6]Note that in the PSU functionality defined in [KRTW19], the sender is not notified when the execution is finished.

We remark that, in the *standard* ideal functionality for PSU, a notification should be returned to the sender once the joint computation has been carried out: the justification is that, in a natural real-world PSU protocol execution, the sender should be aware if the protocol execution has been completed or not. Thus in Fig. 6, we explicitly present the notification, and the functionality returns an output FINISHED to the sender once the joint computation is completed.

Functionality $\mathcal{F}^*_{\mathsf{PSU}}$

Parameters:

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

Functionality:

1. Wait for input $X = \{x_1, \cdots, x_{n_1}\}$ from S, abort if $|X| \neq n_1$;
2. Wait for input $Y = \{y_1, \cdots, y_{n_2}\}$ from R, abort if $|Y| \neq n_2$;
3. Output $Z = X \cup Y$ to R, and FINISHED to S.

Figure 6: The PSU ideal functionality defined in [KRTW19]. Note that, in this formulation, we explicitly describe that the sender obtains a notification FINISHED as output when the PSU evaluation is complete; please see the text in blue.

We further remark that, in the formulation of the PSU functionality in [KRTW19], the interactions between the functionality and the simulator (i.e., ideal world adversary), are not explicitly described. This presentation is consistent with that in [CLOS02], in which the simulator is in charge of the message delivery in the ideal world execution. In this work, to address the subtle issues in existing PSU protocols, we follow Canetti's original formulation [Can00, Can01]; thus, we must explicitly present the interactions between the functionality and the simulator. For example, when the PSU functionality receives an input, the simulator must be notified, and an explicit notification message from the functionality to the simulator will be described; when a player is corrupted, the simulator must be allowed to "see" the corresponding "ideal state". Jumping ahead, in Section 4.1, we provide an equivalent presentation, $\mathcal{F}_{\mathsf{PSU}}$, of the standard PSU functionality by following the original Canetti's formulation.

3.3 Reflection

The "split-execute-assemble" paradigm (also called bucketing technique in [KRTW19]), that has been widely used to support large input sets in PSI protocols, is definitely a promising and interesting idea to design scalable PSU protocols. As pointed out by [KRTW19], however, this paradigm will enable the receiver to learn *if a subset Y_i includes items that belongs to the intersection $X \cap Y$*. More concretely, suppose that the two parties take subsets X_i and Y_i as the inputs of $\Pi_{\text{g-RPMT}}$, then the receiver will learn that Y_i includes some item belonging to $X \cap Y$ after obtaining a bit $b_j = 1$. Notice that, this information is not allowed to be learned by the receiver in PSU.

To address this concern, a careful padding strategy was introduced in [KRTW19]. More specifically, after the input sets X and Y are split into multiple smaller pairs of subsets $\{(X_1, Y_1), \cdots, (X_\beta, Y_\beta)\}$, each subset pair (X_i, Y_i) is padded into $(\tilde{X}_i, \tilde{Y}_i)$; the subset $\tilde{X}_i$ is padded by the sender with special items e, and $\tilde{Y}_i$ is padded by the receiver with a special item e and distinct dummy items. In this way, when the PSU sub-protocol is executed on the k-th subset pair $(\tilde{X}_k, \tilde{Y}_k)$, even if the receiver learns that an item $x^* \in \tilde{X}_k$ belongs to $\tilde{Y}_k$, she cannot determine whether x^* is a real item (i.e., $x^* \in Y_k$) or a special item (i.e., $x^* = e$). Therefore, the receiver cannot learn whether or not Y_k has items belonging to $X \cap Y$.

At first sight, the above strategy adopted by [KRTW19] works. However, Jia et al. [JSZ+22] pointed out the strategy is in fact insufficient to avoid the "leakage" incurred by the "split-execute-assemble" paradigm. In more details, Jia et al. [JSZ+22] observed that when finding $Z_k = Y_k$ from the PSU sub-protocol instance on the k-th pair of subsets $(\tilde{X}_k, \tilde{Y}_k)$, the receiver can learn that Y_k has the items belonging to $X \cap Y$ with an overwhelming probability, as illustrated in Fig. 4. Roughly speaking, there are two cases in which the event $Z_k = Y_k$ happens: the first case Case_1 is that $X_k \neq \emptyset \wedge X_k \subseteq Y_k$, and the second case Case_2 is that $X_k = \emptyset$.

As analyzed by Jia et al. [JSZ+22], according to the parameters used in [KRTW19], the probability of Case_1 happening is overwhelming; this means that the receiver can learn that Y_k has items belonging to $X \cap Y$ (with an overwhelming probability) when observing $Z_k = Y_k$. For completeness, we recall the detailed analysis in Appendix B.

Although Jia et al. [JSZ+22] pointed out the above "leakage", they did not make it clear how this "leakage" is simulated in the ideal world or what is the gap between the real world and the ideal world. Since the "leakage" is not explicitly captured in the ideal functionality $\mathcal{F}^*_{\mathsf{PSU}}$ in Fig. 6 defined by Kolesnikov et al. [KRTW19], it is reasonable to guess that the "leakage" could be deduced from the output $Z = X \cup Y$. It is indeed this case: to simulate the adversary $\mathcal{A}$'s view, as shown in the proof of Π_{KRTW} in [KRTW19], the simulator for the corrupted receiver must *first obtain the entire output* $Z = X \cup Y$, and then deduce the "leakage" from the output Z.

However, we observe that the "leakage" in the protocol Π_{KRTW} is subtly different from the leakage implied in the functionality $\mathcal{F}^*_{\mathsf{PSU}}$: in $\mathcal{F}^*_{\mathsf{PSU}}$ the receiver can only deduce the leakage after obtaining *all the items in* $X \cup Y$, while in Π_{KRTW} the receiver can learn this leakage *during the execution process* of the protocol. To be more precise, in the real world (as exemplified in Fig. 4) , where we assume that the PSU sub-protocol instances are executed sequentially, once the execution of PSU sub-protocol instance on the k-th subset pair $(\tilde{X}_k, \tilde{Y}_k)$ is completed and $Z_k = Y_k$ happens, the receiver can learn with an overwhelming probability that Y_k has items belonging to $X \cap Y$ according to above analysis. Note that at this point, the execution of protocol Π_{KRTW} has not been finished. In the ideal world, however, to simulate the executions of the first k PSU sub-protocol instances, the simulator for the corrupted receiver has to obtain the output $Z = X \cup Y$ first, which means that the execution of the functionality $\mathcal{F}^*_{\mathsf{PSU}}$ must have been finished. Therefore, the two worlds can actually be distinguished. The formal analysis is given in Section 5.1.

From the above, we can see that Π_{KRTW} cannot UC-realize $\mathcal{F}^*_{\mathsf{PSU}}$, under the assumption that the PSU sub-protocol instances are executed sequentially. Then a natural question is that *if the PSU sub-protocol instances are executed simultaneously, can* Π_{KRTW} *UC-realize* $\mathcal{F}^*_{\mathsf{PSU}}$*?*

At a first glance, it seems that the answer is positive, as the receiver in the real world may obtain $Z_1, \cdots, Z_\beta$ at the same time, say t, and then the execution is finished. In this case, the simulator can simulate the leakage properly as it needs not to obtain the entire output Z before time t. However, there is still a subtle issue: before obtaining $Z_1, \cdots, Z_\beta$, the receiver can learn the output, denoted as $\tilde{B}_i$, of each $\Pi_{\text{g-RPMT}}$ sub-protocol instance on $(\tilde{X}_i, \tilde{Y}_i)$, where the bits in $\tilde{B}_i$ indicate if the items in $\tilde{X}_i$ belong to $\tilde{Y}_i$. To properly simulate $\tilde{B}_1, \cdots, \tilde{B}_\beta$, the simulator still needs to obtain the output Z before time t. Therefore, the real world can still be distinguished from the ideal world, and thus the answer is negative. We give the formal analysis in Section 5.1.4. Recall that, whenever observing that $\tilde{B}_k$ only includes 1, the receiver learns that $Z_k = Y_k$ without needing to execute the Π_{OT} sub-protocol instances. In other words, even if the PSU sub-protocol instances are executed simultaneously, the "leakage" in Π_{KRTW} pointed out by [JSZ+22] still exists during the execution.

It can be seen from the above discussion that due to the Π_{OT} sub-protocol and the "split-execute-assemble" paradigm utilized in the design, the receiver in Π_{KRTW} can learn $\tilde{B}_1, \cdots, \tilde{B}_\beta$ and $Z_1, \cdots, Z_\beta$ before obtaining the entire output Z, thus resulting in the extra "leakage" not allowed by the standard functionality $\mathcal{F}^*_{\mathsf{PSU}}$. To understand what functionality the protocol Π_{KRTW} can achieve, we define a relaxed PSU functionality $\mathcal{F}^{\mathsf{b},s}_{\mathsf{rPSU}}$ in Section 4.2.2. Before showing $\mathcal{F}^{\mathsf{b},s}_{\mathsf{rPSU}}$, we also present the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ to capture the security achieved by the subsequent works [JSZ+22, GMR+21] that avoids the "split-execute-assemble" paradigm, but still rely on the Π_{OT} sub-protocol.

4 Ideal Functionalities for PSU

As discussed in Section 3.2, the PSU functionality $\mathcal{F}^*_{\mathsf{PSU}}$ defined in [KRTW19] (as shown in Fig. 6) cannot reflect the leakage during their protocol execution, since the messages between the dummy parties and the ideal functionality are delivered straightforwardly by the simulator in $\mathcal{F}^*_{\mathsf{PSU}}$. In this section, we revisit the ideal functionality for PSU by following Canetti's original formulation [Can00, Can01]. Specifically, we

explicitly describe the instructions in the functionality, like the interactions between the ideal functionality and the simulator, to address the issues in existing PSU protocols [KRTW19, GMR+21, JSZ+22].

4.1 $\mathcal{F}_{\mathsf{PSU}}$: A Standard PSU Functionality

We provide an equivalent formulation of the standard functionality $\mathcal{F}^{*}_{\mathsf{PSU}}$, and denote it as $\mathcal{F}_{\mathsf{PSU}}$, in Fig. 7. In particular, we follow Canetti's original formulation [Can00, Can01] and define $\mathcal{F}_{\mathsf{PSU}}$ by explicitly describing the interactions between the functionality and the simulator.

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

Parameters:

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

Functionality:

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

Figure 7: An equivalent formulation of the standard PSU functionality

In our formulation, whenever receiving an input, the PSU functionality notifies the simulator by sending to it an explicit notification message. Beyond, an "ideal state" $state_{\mathsf{U}}$ for each party U is explicitly introduced in our formulation for recording the party's initial input and intermediate state as well as final output; when one party is corrupted, the simulator is allowed to obtain the corresponding "ideal state". As in Canetti's original formulation, the simulator here is enabled to decide when the joint computation in the functionality starts, by issuing an "OK" message to the functionality. Once the joint computation is finished, the receiver's ideal state is updated from the initial state $\langle Y \rangle$ to $\langle Y, Z = X \cup Y \rangle$, and the sender's ideal state is updated from $\langle X \rangle$ to $\langle X, \textsc{Finished} \rangle$. Finally, the functionality sends $X \cup Y$ and $\textsc{Finished}$ to the sender and the receiver, respectively.

We remark that, the PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ is essentially equivalent to the functionality $\mathcal{F}^{*}_{\mathsf{PSU}}$ given by [KRTW19] (see Fig. 6). By introducing the ideal states and the interactions (between the functionality and the simulator) into $\mathcal{F}_{\mathsf{PSU}}$, we can capture security requirements in a more fine-grained manner; it benefits us a lot to refine the functionality. To address the issues mentioned before, we further present in Section 4.2 two relaxed variants of $\mathcal{F}_{\mathsf{PSU}}$ by taking into account of the practical leakage in the existing PSU protocols [KRTW19, JSZ+22, GMR+21].

4.2 $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$: New Variants of PSU Functionalities

Next we define two relaxed versions, $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$, of the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$. In particular, $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ can be used to capture the security of the OT-based protocols and $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ is used to capture the security of the protocols additionally using the "split-execute-assemble" paradigm.

4.2.1 $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$: PSU Functionality with Set Membership Leakage.

To reflect the inherent leakage during all OT-based PSU protocols, we present a relaxed variant, $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, of the standard functionality $\mathcal{F}_{\mathsf{PSU}}$, as shown in Fig. 8. Compared to $\mathcal{F}_{\mathsf{PSU}}$, the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ further

leaks (to the receiver) the set membership $b_i \in \{0,1\}$ for each $x_i \in X$, prior to finishing the execution. Note that $b_i = 1$ if $x_i \in Y$, otherwise $b_i = 0$.

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

Parameters:

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

Functionality:

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

Figure 8: A relaxed PSU ideal functionality leaking **set membership** in advance. Compared to the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ shown in Fig. 7, $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ additionally adds $\{b_1, \cdots, b_{n_1}\}$ into the receiver's state $state_{\mathsf{R}}$ as shown in steps 3 - 4.

The functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ is similar to $\mathcal{F}_{\mathsf{PSU}}$, except that it additionally leaks whether each item in X belongs to Y. In more details, after receiving the input sets from the sender S and the receiver R, the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ sets $b_i = 1$ if x_i belongs to Y for each $x_i \in X$, otherwise sets $b_i = 0$. Then instead of directly sending $Z = X \cup Y$ to the receiver as in $\mathcal{F}_{\mathsf{PSU}}$, the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ asks the simulator if the receiver is allowed to obtain $\{b_1, \cdots, b_{|X|}\}$ through the message $\langle \text{REQUEST\_IF} \rangle$. After receiving the response $\langle \text{RESPONSE\_IF}, \text{OK} \rangle$ from the simulator, $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ records $\{b_1, \cdots, b_{|X|}\}$ to the receiver's ideal state, and asks the simulator if the receiver is allowed to get $Z = X \cup Y$ via $\langle \text{REQUEST\_ITEM} \rangle$. When receiving the response $\langle \text{RESPONSE\_ITEM}, \text{OK} \rangle$, the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ records Z and FINISHED to the receiver's and the sender's ideal state, and returns Z and FINISHED to the receiver R and the sender S, respectively.

It is clear that the relaxed functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ properly captures the extra leakage due to OT-based designs. However, it still cannot reflect the issues incurred by the "split-execute-assemble" paradigm. To capture the security of the protocols using both OT and the "split-execute-assemble" paradigm, we give a more relaxed PSU functionality in the following.

4.2.2 $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$: PSU Functionality with Set Membership and Subsets Leakage.

In this part, we present a second variant, $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$, of the standard PSU functionality, as shown in Fig. 9. Compared to the first relaxed PSU functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, the functionality $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ splits the pair of input sets X and Y into multiple much smaller pairs of subsets, i.e., $\{(X_1, Y_1), (X_2, Y_2), \cdots, (X_\beta, Y_\beta)\}$. For each pair (X_i, Y_i), the functionality $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ leaks $B_i = \{b_i\}$ and $Z_i = X_i \cup Y_i$ to the receiver, where b_i indicates whether or not each item $x_i \in X_i$ belongs to Y_i.

More precisely, after receiving the input sets from the sender S and the receiver R, the functionality $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ first updates the sender's ideal state and the receiver's state to $\langle X \rangle$ and $\langle Y \rangle$. Unlike the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, in which X and Y are processed as a whole, the functionality $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ splits them into disjoint subsets $\{X_1, X_2, \cdots, X_\beta\}$ and $\{Y_1, Y_2, \cdots, Y_\beta\}$ and then processes the subset pairs $\{(X_i, Y_i)\}_{i \in [\beta]}$ separately. In particular, for each $i \in [\beta]$, the functionality $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ asks the simulator if the receiver is allowed to learn B_i through the message $\langle \text{REQUEST\_IF}, i \rangle$. After receiving the response "OK" from the simulator, the

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

Parameters:

- The functionality interacts with two parties, the sender S and the receiver R, and the simulator Sim;
- Set size for the sender S is n_1 and the set size for the receiver R is n_2;
- The functionality is parameterized with a function bucket which divides a set X into multiple disjoint subsets $\{X_1, X_2, \cdots, X_\beta\}$; we write it as $\{X_1, X_2, \cdots, X_\beta\} \leftarrow \mathsf{bucket}(X)$.

Functionality:

0. Initialize an ideal state $state_{\mathsf{U}} := \emptyset$ for party U where $\mathsf{U} \in \{\mathsf{S}, \mathsf{R}\}$; if U is corrupted, the simulator Sim is allowed to access $state_{\mathsf{U}}$;
1. Upon receiving input $X = \{x_1, \cdots, x_{n_1}\}$ from the sender S, abort if $|X| \neq n_1$; otherwise, update state $state_{\mathsf{S}} := \langle X \rangle$, and send $\langle \textsc{Request}, \mathsf{S} \rangle$ to the simulator Sim;
2. Upon receiving input $Y = \{y_1, \cdots, y_{n_2}\}$ from the receiver R, abort if $|Y| \neq n_2$; otherwise, update state $state_{\mathsf{R}} := \langle Y \rangle$, and send $\langle \textsc{Request}, \mathsf{R} \rangle$ to the simulator Sim;
3. Upon receiving $\langle \textsc{Response}, \mathsf{OK} \rangle$ from Sim, compute $\{X_1, X_2, \cdots, X_\beta\} \leftarrow \mathsf{bucket}(X)$ and $\{Y_1, Y_2, \cdots, Y_\beta\} \leftarrow \mathsf{bucket}(Y)$, then record $\{X_1, X_2, \cdots, X_\beta\}$ and $\{Y_1, Y_2, \cdots, Y_\beta\}$ into $state_{\mathsf{S}}$ and $state_{\mathsf{R}}$, respectively;
4. For each $i \in [\beta]$:
   - Send $\langle \textsc{Request\_If}, i \rangle$ to the simulator Sim;
   - Upon receiving $\langle \textsc{Response\_If}, i, \mathsf{OK} \rangle$ from Sim, for each $j \in [|X_i|]$, set $b_j := 1$ if $X_i[j] \in Y_i$, otherwise set $b_j := 0$, then set $B_i = \{b_1, \cdots, b_{|X_i|}\}$ and add $\langle i, B_i \rangle$ to the receiver's state $state_{\mathsf{R}}$;
   - Send $\langle \textsc{Request\_Bin}, i \rangle$ to the simulator Sim;
   - Upon receiving $\langle \textsc{Response\_Bin}, i, \mathsf{OK} \rangle$ from Sim, compute $Z_i := X_i \cup Y_i$, add $\langle i, \textsc{Finished} \rangle$ to the sender's state $state_{\mathsf{S}}$ and $\langle i, Z_i \rangle$ to the receiver's state $state_{\mathsf{R}}$;
5. Output $Z := Z_1 \cup Z_2 \cup \cdots \cup Z_\beta$ to R, and $\textsc{Finished}$ to S.

Figure 9: A relaxed PSU ideal functionality leaking **set membership and subsets** in advance. Compared to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ in Fig. 8, $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ splits the pair of input sets X and Y into multiple much smaller pairs of subsets, i.e., $\{(X_1, Y_1), (X_2, Y_2), \cdots, (X_\beta, Y_\beta)\}$. For each pair (X_i, Y_i), the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ adds B_i and $Z_i = X_i \cup Y_i$ to the receiver's state $state_{\mathsf{R}}$ as shown in step 4.

functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ records $\langle i, B_i \rangle$ to the receiver's ideal state, then asks the simulator if the receiver can obtain $Z_i = X_i \cup Y_i$ through $\langle \textsc{Request\_Bin}, i \rangle$. When receiving the response "OK" from the simulator, $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ records $\langle i, Z_i \rangle$ and $\langle i, \textsc{Finished} \rangle$ to the sender's ideal state. Finally, the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ adds $Z = Z_1 \cup Z_2 \cup \cdots \cup Z_\beta$ and $\textsc{Finished}$ to the receiver's ideal state and the sender's ideal state, and outputs $Z = Z_1 \cup Z_2 \cup \cdots \cup Z_\beta$ and $\textsc{Finished}$ to the receiver R and the sender S, respectively. Notice that, from the description of $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$, we can see that if the receiver observes that each $b_i \in B_k$ is equal to 1 (i.e., $Z_k = Y_k$), she will learn that there must be some items in Y_k belonging to $X \cap Y$.

We can see that the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ splits the pair of the input sets to multiple smaller pairs of subsets. For each pair of subsets, $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ performs the procedure as in the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$. Therefore, compared to $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ additionally captures the leakage incurred by the "split-execute-assemble" paradigm.

5 Protocols for PSU

In this section, we formally analyze the security of the existing typical PSU protocols, including [KRTW19, JSZ+22, GMR+21, Fri07, DC17]. In Section 5.1, we prove that the (sequential/simultaneous version of) PSU protocol using the "split-execute-assemble" paradigm in [KRTW19] can only UC-realize the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$. Then in Section 5.2, we show that the basic protocol in [KRTW19] (i.e., without using the "split-execute-assemble" paradigm) and the follow-ups [JSZ+22, GMR+21] designed under the same framework can UC-realize the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, but not the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$. At last, we prove that the protocols [Fri07, DC17] based on the additively homomorphic encryption

(AHE) can UC-realize the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ in Section 5.3.

5.1 KRTW-PSU: Can Realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ but Not $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$

To the best of our knowledge, the PSU protocol in [KRTW19] is the first and the only one that leverages the "split-execute-assemble" paradigm. Following this way, a large number of PSU sub-protocol instances need to be executed, especially for large datasets. According to the parameters set in [KRTW19], when both input sets X and Y are with size 2^{20}, they are split into $0.06 \cdot 2^{20}$ bins β (i.e., the number of PSU sub-protocol instances). We assume that, without loss of generality, all the PSU sub-protocols instances are executed sequentially or simultaneously[7]. Then we formally analyze the security of the protocol in [KRTW19] in this section.

5.1.1 $\Pi_{\mathsf{Unified}}^{\mathsf{KRTW}}$: KRTW-PSU design framework

We give the design framework $\Pi_{\mathsf{Unified}}^{\mathsf{KRTW}}$ of the protocol in [KRTW19] in Fig. 10. More specifically, the sender and the receiver first use simple hashing to split their input sets X and Y into multiple subsets $\{X_1, \cdots, X_\beta\}$ and $\{Y_1, \cdots, Y_\beta\}$, respectively. Then the sender pads each subset X_i with special items e to obtain $\tilde{X}_i$, and the receiver pads Y_i with a special item e and dummy items to obtain $\tilde{Y}_i$. After that, for each subset pair $(\tilde{X}_i, \tilde{Y}_i)$, the two parties perform a PSU sub-protocol $\Pi_{\mathsf{Unified}}^{\mathsf{OT}}$ shown in Fig. 11. The formal security analysis of $\Pi_{\mathsf{Unified}}^{\mathsf{OT}}$ is postponed to Section 5.2. Since the PSU sub-protocol instances can be executed sequentially or simultaneously, we will analyze the security of sequential version and simultaneous version, respectively.

5.1.2 The sequential version of KRTW-PSU can realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$

In this section, we prove that the sequential version of the protocol in [KRTW19] can UC-realize the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$. When the PSU sub-protocol instances are executed sequentially, we can see from the above that the receiver can obtain the subsets Z_i for $i < \beta$ during the execution. Moreover, for each execution of the PSU sub-protocol over $(\tilde{X}_i, \tilde{Y}_i)$, the receiver can obtain a bit b_j for each $x_j \in \tilde{X}_i$; if $x_j \in \tilde{Y}_i$, $b_j = 1$, otherwise $b_j = 0$. Next we show that the protocol $\Pi_{\mathsf{Unified}}^{\mathsf{KRTW}}$ can only securely realize the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ in Fig. 9, which as shown in Section 4.2 is the sole functionality that allows leaking subsets and bits to the receiver *in advance*. The security is formally stated in Theorem 5.1.

Theorem 5.1. *The sequential version of protocol following the framework $\Pi_{\mathsf{Unified}}^{\mathsf{KRTW}}$ in Fig. 10 UC-realizes the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ (as in Fig. 9), against static, semi-honest adversaries.*

Proof. We will show that for any efficient adversary $\mathcal{A}$ who can corrupt the sender or the receiver, we can construct a simulator Sim to simulate the view of adversary $\mathcal{A}$, so that any PPT environment $\mathcal{E}$ cannot distinguish the execution in the ideal world from that in the real world.

Corrupted Sender: Sim first sends the input set X to $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$. For each bin i, Sim simulates $\mathcal{A}$'s view as follows: After receiving $\langle \text{REQUEST\_IF}, i \rangle$ from $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$, Sim can simulate $\mathcal{A}$'s view before invoking Π_{OT} as in [KRTW19]. Then, Sim sends $\langle \text{RESPONSE\_IF}, i, \text{OK} \rangle$ to $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$. Once receiving $\langle \text{REQUEST\_BIN}, i \rangle$ from $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$, Sim can simulate Π_{OT} for this bin as in [KRTW19]. After that, Sim sends $\langle \text{RESPONSE\_BIN}, i, \text{OK} \rangle$ to $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b,s}}$ and obtains $(i, \text{FINISHED})$ from the sender's state $state_{\mathsf{S}}$. Compared to the simulator for corrupted sender constructed in [KRTW19], Sim additionally receives some request messages and additionally sends some response messages for each bin. Moreover, due to the request/response messages, the environment $\mathcal{E}$ will receive the honest receiver's output $X \cup Y$ at the same time in the real and ideal worlds. Therefore, Sim can simulate $\mathcal{A}$'s view such that $\mathcal{E}$ cannot distinguish between the two worlds.

[7] In practice, for large datasets, it is impossible to execute all the PSU sub-protocols at the same time, but it is possible to parallelize the protocol in multiple threads. For example, Kolesnikov et al. [KRTW19] leveraged 32 threads to parallelize the protocol. However, in each thread, the PSU sub-protocols are executed sequentially.

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

Parameters:

- Let n_1 and n_2 denote the size of the sender S's and the receiver R's input set, respectively; Let ℓ be the bit-length of each item in the sender's set or the receiver's set.
- Let β be the number of bins, and m be the maximum bin size;
- Let $H(\cdot)$ be a hash function $H : \{0,1\}^\ell \to [\beta]$;
- Let e denote a special item, where $e \in \{0,1\}^\ell$, and $d_1, d_2, \cdots, d_n \in \{0,1\}^\ell \setminus (X \cup Y)$ be the distinct dummy items; n denotes the maximum set size, i.e., $n = \max(n_1, n_2)$.

Inputs:

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

Protocol:

1. S and R split their input sets X and Y into β bins using hash function H. Let X_i and Y_i denote the set of items in the sender's and receiver's i-th bin, respectively;
2. S pads each X_i with special items e up to the maximum bin size m, then randomly permutes all items in the bin and gets $\tilde{X}_i$;
3. R pads each Y_i with one special item e and different dummy items chosen from $\{d_1, d_2, \cdots, d_n\}$ to the maximum bin size m, then get the padded set $\tilde{Y}_i$;
4. R initializes set $Z := \emptyset$;
5. The sender S and the receiver R execute the PSU sub-protocol instances $\Pi^{\text{OT}}_{\text{Unified}}$ **sequentially / simultaneously** (see Fig. 11 for the sub-protocol description); the i-th sub-protocol instance is based on the input sets $\tilde{X}_i$ and $\tilde{Y}_i$, where $i \in [\beta]$. More concretely,
   - S acts as sender with input set $\tilde{X}_i$;
   - R acts as receiver with input set $\tilde{Y}_i$;
   - R obtains output $\tilde{Z}_i$, then R discards the special item e and dummy items from $\tilde{Z}_i$ and obtains $Z_i = X_i \cup Y_i$; S obtains output FINISHED;
6. R outputs $Z := Z_1 \cup Z_2 \cup \cdots \cup Z_\beta$, and S outputs FINISHED.

Figure 10: The design framework in [KRTW19]. The formal security analysis of $\Pi^{\text{OT}}_{\text{Unified}}$ is given in Section 5.2.

Corrupted Receiver: Likewise, Sim first sends the input set Y to $\mathcal{F}^{\text{b,s}}_{\text{rPSU}}$. For each bin i, Sim simulates $\mathcal{A}$'s view as follows: after receiving $\langle \text{REQUEST\_IF}, i \rangle$ from $\mathcal{F}^{\text{b,s}}_{\text{rPSU}}$, Sim can simulate $\mathcal{A}$'s view before sending s_j for each x_j in this bin as in [KRTW19]. Then, Sim sends $\langle \text{RESPONSE\_IF}, i, \text{OK} \rangle$ to $\mathcal{F}^{\text{b,s}}_{\text{rPSU}}$. After obtaining $\langle i, B_i \rangle$, Sim can simulate s_j. Once receiving $\langle \text{REQUEST\_BIN}, i \rangle$ from $\mathcal{F}^{\text{b,s}}_{\text{rPSU}}$, for each Π_{OT} protocol instance, Sim simulates the steps before sending items. To simulate the last step (i.e., sending items), Sim needs to obtain subset Z_i. Therefore, Sim sends $\langle \text{RESPONSE\_BIN}, i, \text{OK} \rangle$ to $\mathcal{F}^{\text{b,s}}_{\text{rPSU}}$ and obtains (i, Z_i) from the receiver's state $state_{\text{R}}$. Then, Sim can simulate sending items for each Π_{OT} protocol instance. Compared to the simulator for corrupted receiver constructed in [KRTW19], Sim just additionally receives some request messages and additionally sends some response messages for each bin. Likewise, due to the request/response messages, the environment $\mathcal{E}$ will receive the honest sender's output FINISHED at the same time in the real and ideal worlds. Therefore, Sim can simulate $\mathcal{A}$'s view such that $\mathcal{E}$ cannot distinguish between the two worlds. □

At this point, we complete the proof that the sequential version of the protocol in [KRTW19] can UC-realize the relaxed PSU functionality $\mathcal{F}^{\text{b,s}}_{\text{rPSU}}$, which allows leaking subsets and set membership to the receiver in advance. It is natural to ask if the protocol can UC-realize the functionality $\mathcal{F}^{\text{b}}_{\text{rPSU}}$ that leaks less information (say, only bits) than functionality $\mathcal{F}^{\text{b,s}}_{\text{rPSU}}$. A negative answer is given in the next part.

5.1.3 The sequential version of KRTW-PSU cannot realize $\mathcal{F}^{\text{b}}_{\text{rPSU}}$

In this section, we prove that the sequential version of the protocol in [KRTW19] cannot UC-realize the relaxed PSU functionality $\mathcal{F}^{\text{b}}_{\text{rPSU}}$. Now, we divide time into non-overlapping slots $t_1, t_2, \cdots, t_\beta$ and the i-th PSU sub-protocol instance is executed in time slot t_i. At the end of time slot t_k where $1 < k < \beta$, the

environment can learn the unions in the first k bins, i.e., $Z_1, Z_2, \cdots, Z_k$ but does not receive the sender's output FINISHED. In the ideal world, the simulator Sim for the corrupted receiver needs to simulate the unions in the first k bins. To this end, the simulator Sim can have the following two strategies:

One strategy is to send $\langle$RESPONSE_ITEM, OK$\rangle$ to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ to obtain set Z. By using set Z, the simulator Sim can obliviously simulate the unions $Z_1, Z_2, \cdots, Z_k$. However, the strategy will result in that the environment receives the sender's output FINISHED immediately. Note that in the real world, the environment will not receive output FINISHED from the sender at the end of time slot t_k. Then, the environment can distinguish the two worlds.

The other strategy is not to send $\langle$RESPONSE_ITEM, OK$\rangle$ to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ such that in the ideal world, the environment cannot obtain output FINISHED from the sender while the simulator cannot obtain set Z. Therefore, the simulator has to guess the unions $Z_1, Z_2, \cdots, Z_k$. Note that the simulator Sim can obtain the corrupted receiver's input set Y, and thus learns subsets $Y_1, Y_2, \cdots, Y_k$. The simulator Sim actually needs to guess $Z_1 \setminus Y_1, Z_2 \setminus Y_2, \cdots, Z_k \setminus Y_k$ without knowing the sender's input set X. If the items in set X are chosen from a large range, the simulator guess $Z_1 \setminus Y_1, Z_2 \setminus Y_2, \cdots, Z_k \setminus Y_k$ correctly with a very low probability. Whereas, in the real world, the probability that the environment obtains the correct $Z_1, Z_2, \cdots, Z_k$ is 1 except negligible probability. Then, the environment can distinguish the two worlds.

According to the above informal analysis, we can see that for any simulator, we can construct an environment to distinguish the execution in the two worlds. Therefore, we have Theorem 5.2, and we will prove it next.

Theorem 5.2. *The sequential version of protocol following the framework* $\Pi_{\mathsf{Unified}}^{\mathsf{KRTW}}$ *Fig. 10 cannot UC-realize functionality* $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ *(as in Fig. 8), against static, semi-honest adversaries.*

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

Construction of environment $\mathcal{E}$. The environment $\mathcal{E}$ chooses sets X and Y as the inputs of the sender and receiver, respectively. Without loss of generality, let the real world semi-honest adversary $\mathcal{A}$ be a *dummy* adversary who will follow environment $\mathcal{E}$'s instructions, and immediately forward each corrupted player's state to the environment.

Based on function bucket(), the environment $\mathcal{E}$ knows that set X will be split into subsets $X_1, X_2, \cdots, X_\beta$, and set Y will be split into subsets $Y_1, Y_2, \cdots, Y_\beta$. Thus, $\mathcal{E}$ knows sets $Z_1, Z_2, \cdots, Z_\beta$ where $Z_i = X_i \cup Y_i$ for each $i \in [\beta]$.

The environment $\mathcal{E}$, instructs the dummy adversary $\mathcal{A}$, to corrupt the receiver at the beginning of the protocol execution. The environment $\mathcal{E}$ chooses an integer k, where $1 < k < \beta$. Finally, if the tuple $(Z_1, Z_2, \cdots, Z_k)$ has been reported by the dummy adversary $\mathcal{A}$ and message FINISHED has not been reported by the honest sender S, the environment $\mathcal{E}$ outputs 1, otherwise, outputs 0.

The real world execution. In the real world, the PSU sub-protocol instances are executed *sequentially*. We divide time into non-overlapping slots $t_1, t_2, \cdots, t_\beta$. In time slot t_i, the i-th PSU sub-protocol instance is executed based on the i-th pair of subsets, and thus the receiver obtains the output Z_i at the end of the time slot. Therefore, at the end of time slot t_k, the receiver obtains $Z_1, Z_2, \cdots, Z_k$. Note that, the receiver is corrupted and under the control by the semi-honest real world adversary $\mathcal{A}$, the tuple $(Z_1, Z_2, \cdots, Z_k)$ must be reported to the environment at the end of time slot t_k. Note also that, the protocol execution has not been finished, the honest sender S is *not supposed to return* the message FINISHED to the environment $\mathcal{E}$ at the end of time slot t_k.

The ideal world execution. In the ideal world, since the (dummy) receiver is corrupted, the simulator Sim is allowed to access to the ideal state $state_{\mathsf{R}} = \langle Y \rangle$. After receiving from the functionality the message $\langle$REQUEST_IF$\rangle$, to obtain more information, the simulator Sim can send $\langle$RESPONSE_IF, OK$\rangle$ to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$. Then, the simulator Sim can obtain $\{b_1, \cdots, b_{n_1}\}$ from the receiver's state $state_{\mathsf{R}}$. However, the bits $\{b_1, \cdots, b_{n_1}\}$ are not enough for the simulator Sim to simulate $Z_1, Z_2, \cdots, Z_k$. Therefore, the simulator Sim *must* face the following two simulation strategies:

1. *Do send* $\langle$RESPONSE_ITEM, $OK\rangle$ *to the functionality* $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, *immediately.*

   Note that, now the functionality will update the ideal states into $state_{\mathsf{S}} := \langle X, \text{FINISHED}\rangle$ and $state_{\mathsf{R}} := \langle Y, Z\rangle$, and immediately report FINISHED to the environment $\mathcal{E}$.
2. *Do not send* $\langle$RESPONSE_ITEM, $OK\rangle$ *to* $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, *immediately.*

   Note that, now the functionality will **not** update the ideal states into $state_{\mathsf{S}} := \langle X, \text{FINISHED}\rangle$ and $state_{\mathsf{R}} := \langle Y, Z\rangle$; of course, no output FINISHED will be reported to the environment $\mathcal{E}$ immediately. Note again that, the simulator Sim is aware of the ideal state $state_{\mathsf{R}} = \langle Y\rangle$. We must emphasize that, however, Sim cannot obtain Z. At the same time, Sim must simulate the view of $\mathcal{A}$ for the first k executions of sub-protocol instances. Let $(Z_1^{\mathsf{ideal}}, Z_2^{\mathsf{ideal}}, \cdots, Z_k^{\mathsf{ideal}})$ denote the tuple in the view of $\mathcal{A}$, which will be reported to the environment.

Security analysis. Now, we can see, if the simulator follows the first simulation strategy, the environment will tell the difference with probability 1, since in the real world, no output FINISHED will be reported while there is FINISHED in the ideal world.

If the simulator follows the second simulation strategy, the probability that $\mathcal{E}$ outputs 1 in the real world is 1 except negligible probability. However, in the ideal world, assuming that the items in set X are selected from a large range, the probability that $Z_i^{\mathsf{ideal}} = Z_i$ for all $i \in [k]$ is far less than 1. Thus, the probability that $\mathcal{E}$ outputs 1 in the ideal world is far less than 1. Therefore, $\mathcal{E}$ can distinguish the two worlds with non-negligible probability.

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

Currently, we complete the proof that the sequential version of the protocol in [KRTW19] *cannot* UC-realize the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$. We can see from the proof of Theorem 5.2 that if the simulator can obtain the set Z, then it can simulate each subset Z_i properly. To guarantee that the environment cannot distinguish between the two worlds through the output FINISHED from the honest sender, the receiver in the real world should obtain $Z_1, Z_2, \cdots, Z_\beta$ *at the same time*, which means that all PSU sub-protocol instances in [KRTW19] need to be executed simultaneously. At first glance, the simultaneous version of the protocol of [KRTW19] could UC-realize the relaxed PSU functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$. In fact, we show it is not this case in Section 5.1.4.

5.1.4 The simultaneous version of KRTW-PSU cannot realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$

In this section, we prove that even if all PSU sub-protocol instances are executed simultaneously in [KRTW19], the simultaneous version of the protocol in [KRTW19] cannot yet UC-realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$.

In the simultaneous version, although the output FINISHED cannot be leveraged to distinguish between the two worlds, we can still construct an environment to distinguish the real world from the ideal world for any simulator. Roughly speaking, the main reason is that the simulator only obtaining $\{b_1, \cdots, b_{n_1}\}$ from the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ does not know the associated item with b_i, and thus knows nothing about which bin b_i should be in. More specifically, even if the PSU sub-protocol instances are executed simultaneously, for each instance over (X_i, Y_i), the receiver can obtain a bit b_j for each item $x_j \in X_i$ in advance. We denote the bit set for (X_i, Y_i) as B_i. Therefore, in the real world, there is a time t when the receiver has obtained all the bit sets $\tilde{B}_1, \tilde{B}_2, \cdots, \tilde{B}_\beta$ but has not received any subset Z_i, where $\tilde{B}_i$ is obtained by padding B_i with 1 up to the maximum bin size. Moreover, the environment is not supposed to receive FINISHED from the sender since the protocol execution has not been finished. In the ideal world, the simulator for the corrupted receiver have the following two strategies to simulate the $\mathcal{A}$'s view *before time* t:

One strategy is to send $\langle$RESPONSE_ITEM, OK$\rangle$ to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ to obtain set Z. By using set Z, the simulator Sim can obliviously simulate the bit sets $\tilde{B}_1, \tilde{B}_2, \cdots, \tilde{B}_\beta$. However, the strategy will result in that the environment receives the sender's output FINISHED immediately. Note that in the real world, the environment will not receive output FINISHED from the sender until the execution ends. Then, the environment can distinguish the two worlds.

The other strategy is not to send $\langle$RESPONSE_ITEM, OK$\rangle$ to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, so that in the ideal world, the environment cannot obtain output FINISHED from the sender while the simulator cannot obtain set Z. Therefore, the simulator has to guess the bit sets $\tilde{B}_1, \tilde{B}_2, \cdots, \tilde{B}_\beta$. Note that the simulator can obtain $\{b_1, \cdots, b_{n_1}\}$ by sending $\langle$RESPONSE_IF, OK$\rangle$ to $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, where $b_i \in \{0, 1\}$. In other words, the simulator can learn the size of $X \setminus Y$, i.e., the number of 1's in $\{b_1, \cdots, b_{n_1}\}$. However, the simulator cannot know the size of $X_i \setminus Y_i$, i.e., the number of 1's in $\tilde{B}_i$. More concretely, for an item x_k, which bin x_k is put into depends on the value $h(x_k)$ where h is the hash function used in simple hashing. Assuming that $b_k = 1$, the simulator does not know the associated item x_k, and thus cannot know $h(x_k)$. This means that the simulator does not know which $\tilde{B}_i$ should include $b_k = 1$. Thus, the simulator guess $\tilde{B}_1, \tilde{B}_2, \cdots, \tilde{B}_\beta$ correctly with a very low probability. Whereas, in the real world, the probability that the environment obtains the correct $\tilde{B}_1, \tilde{B}_2, \cdots, \tilde{B}_\beta$ is 1 except negligible probability. Then, the environment can distinguish between the two worlds.

According to the above informal analysis, we can see that for any simulator, we can construct an environment to distinguish between the execution in the real world from that in the ideal world. The formal statement is given in Theorem 5.3.

Theorem 5.3. *The simultaneous version of protocol following the framework $\Pi_{\mathsf{Unified}}^{\mathsf{KRTW}}$ cannot UC-realize the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ (as in Fig. 8), against static, semi-honest adversaries.*

Proof. Similar to the proof of Theorem 5.2, we first construct an environment $\mathcal{E}$. Then we will show that for any simulator Sim, the constructed $\mathcal{E}$ can tell the difference between the execution in the real world and that in the ideal world, with at least non-negligible probability.

Construction of environment $\mathcal{E}$. The environment $\mathcal{E}$ chooses sets X and Y as the inputs of the sender and receiver, respectively. Without loss of generality, let the real world semi-honest adversary $\mathcal{A}$ be a *dummy* adversary who will follow $\mathcal{E}$'s instructions, and immediately forward each corrupted player's state to $\mathcal{E}$.

Based on the function bucket(), $\mathcal{E}$ knows that the set X will be split into subsets $X_1, X_2, \cdots, X_\beta$, and Y will be split into subsets $Y_1, Y_2, \cdots, Y_\beta$. Therefore, $\mathcal{E}$ knows $X_1 \setminus Y_1, X_2 \setminus Y_2, \cdots, X_\beta \setminus Y_\beta$, and thus the number of 0 in each $\tilde{B}_i$.

The environment $\mathcal{E}$ instructs the dummy adversary $\mathcal{A}$ to corrupt the receiver at the beginning of the protocol execution, and then chooses a time t. Finally, if the tuple $(\tilde{B}_1, \tilde{B}_2, \cdots, \tilde{B}_\beta)$ has been reported by the dummy adversary $\mathcal{A}$ and the message FINISHED has not been reported by the honest sender S at that time, the environment $\mathcal{E}$ outputs 1, otherwise, outputs 0.

The real world execution. In the real world, the Π_{OT} sub-protocol instances will not be executed until the execution of $\Pi_{\mathsf{g\text{-}RPMT}}$ sub-protocol instance is finished. Therefore, there is a time t when the receiver obtains $(\tilde{B}_1, \cdots, \tilde{B}_\beta)$ but not the items in subset Z_i. Note that, the receiver is corrupted and under the control of the semi-honest real world adversary $\mathcal{A}$, the tuple $(\tilde{B}_1, \cdots, \tilde{B}_\beta)$ must be reported to the environment at the time t. Note also that, the protocol execution has not been finished, the honest sender S is *not supposed to return* the message FINISHED to $\mathcal{E}$ at the time t.

The ideal world execution. In the ideal world, since the (dummy) receiver is corrupted, the simulator Sim is allowed to access to the ideal state $state_{\mathsf{R}} = \langle Y \rangle$. After receiving from the functionality the message $\langle$REQUEST_IF$\rangle$, to obtain more information, the simulator Sim can send $\langle$RESPONSE_IF, OK$\rangle$ to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$. Then, the simulator Sim can obtain $\{b_1, \cdots, b_{n_1}\}$ from the receiver's state $state_{\mathsf{R}}$. However, the bits $\{b_1, \cdots, b_{n_1}\}$ are not enough for the simulator Sim to simulate $(\tilde{B}_1, \cdots, \tilde{B}_\beta)$. Therefore, the simulator Sim *must* face the following two simulation strategies:

1. *Do send $\langle$RESPONSE_ITEM, OK$\rangle$ to the functionality $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, immediately.*

   Note that, now the functionality will update the ideal states into $state_{\mathsf{S}} := \langle X,$ FINISHED$\rangle$ and $state_{\mathsf{R}} := \langle Y, Z \rangle$, and immediately report FINISHED to the environment $\mathcal{E}$.
2. *Do not send $\langle$RESPONSE_ITEM, OK$\rangle$ to $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, immediately.*

   Note that, now the functionality will **not** update the ideal states into $state_{\mathsf{S}} := \langle X,$ FINISHED$\rangle$ and $state_{\mathsf{R}} := \langle Y, Z \rangle$; of course, no output FINISHED will be reported to the environment $\mathcal{E}$ immediately.

Note again that, the simulator Sim is aware of the ideal state $state_{\mathsf{R}} = \langle Y \rangle$. We must emphasize that, however, Sim cannot obtain Z. At the same time, Sim must simulate the view of $\mathcal{A}$ before the execution of Π_{OT} sub-protocol instances. Let $(\tilde{B}_1^{\mathsf{ideal}}, \tilde{B}_2^{\mathsf{ideal}}, \cdots, \tilde{B}_k^{\mathsf{ideal}})$ denote the tuple in the view of $\mathcal{A}$, which will be reported to the environment.

Security analysis. Now, we can see, if the simulator follows the first simulation strategy, the environment will tell the difference with probability 1, since in the real world, no output FINISHED will be reported while FINISHED is reported in the ideal world.

If the simulator follows the second simulation strategy, the probability that $\mathcal{E}$ outputs 1 in the real world is 1 except negligible probability. In the ideal world, however, the probability that $\tilde{B}_i^{\mathsf{ideal}} = \tilde{B}_i$ for all $i \in [k]$ is far less than 1 as the simulator does not know $|X_i \setminus Y_i|$. Thus, the probability that $\mathcal{E}$ outputs 1 in the ideal world is far less than 1. Therefore, $\mathcal{E}$ can distinguish the two worlds with non-negligible probability.

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

5.2 OT-based PSU: Can Realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$ but Not $\mathcal{F}_{\mathsf{PSU}}$

In the previous section, we show that the protocol in [KRTW19] cannot UC-realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, even if all PSU sub-protocol instances in [KRTW19] are executed simultaneously. As analyzed before, the main reason is that the receiver's input set Y is split into multiple subsets following the "split-execute-assemble" paradigm. In [KRTW19], a basic scheme without using the "split-execute-assemble" paradigm has also been proposed, in which the input set Y is processed as a whole (i.e., the number of bins $\beta = 1$). However, the basic scheme is not efficient enough especially for large datasets[8]. Subsequently, two practical PSU protocols [JSZ+22, GMR+21] were put forward without relying on the "split-execute-assemble" paradigm. We observe that these two protocols [JSZ+22, GMR+21] together with the basic protocol in [KRTW19] follow the same design framework as shown in Fig. 11. In this section, we prove that all the protocols following this framework can UC-realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$, but cannot realize the functionality $\mathcal{F}_{\mathsf{PSU}}$.

5.2.1 $\Pi_{\mathsf{Unified}}^{\mathsf{OT}}$: OT-based PSU design framework

We unify the three protocols [JSZ+22, GMR+21, KRTW19] into the same framework $\Pi_{\mathsf{Unified}}^{\mathsf{OT}}$ as shown in Fig. 11. More specifically, the two parties first perform a "generalized Reversed Private Membership Test" sub-protocol $\Pi_{\mathsf{g\text{-}RPMT}}$[9] with input sets X and Y, respectively. After $\Pi_{\mathsf{g\text{-}RPMT}}$, the receiver can obtain a bit b_i for each item in X; if $X[i] \in Y$, $b_i := 1$, otherwise $b_i := 0$. At last, for each item in X, the receiver can obtain $X[i]$ via Π_{OT} if $b_i = 0$, otherwise obtain $\perp$. Note that, in practice, the n_1 number of Π_{OT} instances can be implemented by OT extension [IKNP03], so that the receiver can obtain all the items in $X \setminus Y$ at the same time.

We remark that the only difference between the three protocols is the way of designing the sub-protocol $\Pi_{\mathsf{g\text{-}RPMT}}$. (Note that, the sub-protocol $\Pi_{\mathsf{g\text{-}RPMT}}$ of [KRTW19, JSZ+22, GMR+21] can be found in Appendix C.4.)

5.2.2 OT-based PSU can realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$

Next we prove that all the protocols following the framework $\Pi_{\mathsf{Unified}}^{\mathsf{OT}}$ in Fig. 11 can UC-realize $\mathcal{F}_{\mathsf{rPSU}}^{\mathsf{b}}$. Recall that in $\Pi_{\mathsf{Unified}}^{\mathsf{OT}}$, the receiver does not split her input set Y into multiple subsets; that is, all the

[8] For each item $x_i \in X$, the receiver in the basic scheme needs to perform a n_2-degree polynomial interpolation, where n_2 is the size of set Y. The computational cost of n_2-degree polynomial interpolation is $O(n_2 \log^2 n_2)$, and thus the overall computational cost will be $O(n_1 n_2 \log^2 n_2)$, which is not acceptable for large n_1 and n_2 in practice.

[9] The sub-protocol $\Pi_{\mathsf{g\text{-}RPMT}}$ is almost equivalent to the "permuted characteristic" sub-protocol Π_{pc} in [GMR+21] except that Π_{pc} includes the permutation on input set X. Here, the sender randomly permutes the set X before invoking $\Pi_{\mathsf{g\text{-}RPMT}}$.

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

Parameters:

- Let n_1 and n_2 denote the set size for the sender S's input set, and for the receiver R's input set, respectively; Let ℓ be the bit-length of each item in the sender's set or the receiver's set;

Inputs:

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

Protocol:

1. The sender S randomly permutes the set X into the set X^*;
2. The two players S and R invoke the "generalized Reversed Private Membership Test" sub-protocol $\Pi_{\text{g-RPMT}}$ shown in Fig. 18:
   - S acts as sender with input set X^*;
   - R acts as receiver with input set Y;
   - R obtains output b_i for each $i \in [n_1]$;
3. R initializes set $Z := Y$;
4. The two players S and R **simultaneously** execute n_1 number of Π_{OT} instances. In the i-th instance, where $i \in [n_1]$,
   - R acts as receiver with input b_i;
   - S acts as sender with input $(X^*[i], \perp)$;
   - If R obtains $X^*[i]$, R sets $Z := Z \cup \{X^*[i]\}$;
5. R outputs Z, and S outputs FINISHED.

Figure 11: The design framework unifying PSU protocols in [KRTW19, JSZ+22, GMR+21]. Here, the protocol in [KRTW19] refers to the basic scheme without using "split-execute-assemble" paradigm.

items in Y are processed as a whole. Then after obtaining the bits $\{b_1, b_2, \cdots, b_{n_1}\}$ that indicate if each item $X[i] \in Y$, the receiver invokes the Π_{OT} instances with $\{b_1, b_2, \cdots, b_{n_1}\}$ as the inputs, and gets the items $X \setminus Y$. Intuitively, $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ can UC-realize the relaxed functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, which leaks only the set membership information and can be seen a special case of $\mathcal{F}^{\mathsf{b,s}}_{\mathsf{rPSU}}$ (i.e., the number of bins β is 1). Formally, the security of $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ is stated in Theorem 5.4.

Theorem 5.4. *The protocol following the framework $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ in Fig. 11 UC-realizes the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ (as in Fig. 8), against static, semi-honest adversaries.*

Proof. To prove this theorem, we will show that for any efficient adversary $\mathcal{A}$, we can construct a simulator Sim to properly simulate the view of the corrupted sender and the corrupted receiver, such that any PPT environment $\mathcal{E}$ cannot distinguish between the execution in the ideal world from that in the real world. In particular, according to the modular design of $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ from the sub-protocols $\Pi_{\text{g-RPMT}}$ and Π_{OT}, the simulator Sim can be constructed by invoking the simulator Sim′ in [KRTW19], [JSZ+22] or [GMR+21].

Corrupted Sender: Simulator Sim first sends the input set X to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$. After receiving $\langle$REQUEST_IF$\rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim first invokes Sim′ to simulate the execution of the sub-protocol $\Pi_{\text{g-RPMT}}$. Then, Sim sends $\langle$RESPONSE_IF, OK$\rangle$ to the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$. Once receiving $\langle$REQUEST_ITEM$\rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim simulates the execution of sub-protocol Π_{OT} by invoking Sim′. When $\mathcal{A}$ sends items in all Π_{OT} instances, Sim sends $\langle$RESPONSE_ITEM, OK$\rangle$ to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and then obtains $\langle$FINISHED$\rangle$ from the sender's state $state_{\mathsf{S}}$. Compared to the simulator for corrupted sender in [KRTW19], [JSZ+22] or [GMR+21], Sim just additionally receives some request messages and additionally sends some response messages. Moreover, due to the request/response messages, the environment $\mathcal{E}$ will receive the honest receiver's output $X \cup Y$ at the same time in the real and ideal worlds. Therefore, Sim can simulate $\mathcal{A}$'s view such that $\mathcal{E}$ cannot distinguish the two worlds.

Corrupted Receiver: Likewise, simulator Sim first sends the input set Y to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$. After receiving $\langle$REQUEST_IF$\rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim first invokes Sim′ to simulate the execution of the sub-protocol $\Pi_{\text{g-RPMT}}$ except for the

last steps (i.e., computing and sending s_i for each $x_i \in X$ in [KRTW19], computing and sending I_i for each $x_i \in X$ in [JSZ+22], computing a'_i for each $x_i \in X$ and invoking $\mathcal{F}_{\mathsf{bEQ}}$ in [GMR+21]). To simulate the last steps of $\Pi_{\text{g-RPMT}}$, Sim sends $\langle \text{RESPONSE\_IF}, \text{OK} \rangle$ to the functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and then obtains $\{b_1, b_2, \cdots, b_{n_1}\}$. Given $\{b_1, b_2, \cdots, b_{n_1}\}$, Sim can invoke Sim′ to simulate the last steps of $\Pi_{\text{g-RPMT}}$. Once receiving $\langle \text{REQUEST\_ITEM} \rangle$ from $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, Sim invokes Sim′ to simulate the execution of the sub-protocol Π_{OT} except for the last step (i.e., sending items). Then, Sim sends $\langle \text{RESPONSE\_ITEM}, \text{OK} \rangle$ to $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$ and then obtains set Z. By using the items in set $X \setminus Y$, Sim can simulate the last step of each Π_{OT} instance. Compared to the simulator for corrupted receiver in [KRTW19], [JSZ+22] or [GMR+21], Sim just additionally receives some request messages and additionally sends some response messages. Likewise, due to the request/response messages, the environment $\mathcal{E}$ will receive the honest sender's output FINISHED at the same time in the real and ideal worlds. Therefore, Sim can simulate $\mathcal{A}$'s view such that $\mathcal{E}$ cannot distinguish the two worlds. □

Now we complete the proof that the protocol following the framework $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ can UC-realize the relaxed PSU functionality $\mathcal{F}^{\mathsf{b}}_{\mathsf{rPSU}}$, which allows the set membership to the receiver in advance. It is natural to ask whether it can UC-realize the functionality $\mathcal{F}_{\mathsf{PSU}}$. Next we give a negative answer in the following part.

5.2.3 OT-based PSU cannot realize $\mathcal{F}_{\mathsf{PSU}}$

In this section, we show that the protocols unified in Fig. 11 cannot UC-realize the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$. We can see from the proof of Theorem 5.4 that to simulate the corrupted receiver's view, the simulator has to obtain $\{b_1, b_2, \cdots, b_{n_1}\}$ *in advance*. Therefore, these protocol cannot UC-realize the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$, as $\mathcal{F}_{\mathsf{PSU}}$ leaks nothing to the receiver before sending the final output Z. Formally, the security is stated in Theorem 5.5.

Theorem 5.5. *The protocol following the framework $\Pi^{\mathsf{OT}}_{\mathsf{Unified}}$ in Fig. 11 cannot UC-realize functionality $\mathcal{F}_{\mathsf{PSU}}$ (as in Fig. 7), against static, semi-honest adversaries.*

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

Construction of environment $\mathcal{E}$. The environment $\mathcal{E}$ chooses sets X and Y as the inputs of the sender and the receiver, respectively. Without loss of generality, let the real world semi-honest adversary $\mathcal{A}$ be a *dummy* adversary who will follow $\mathcal{E}$'s instructions, and immediately forward each corrupted player's state to the environment. Since the environment $\mathcal{E}$ knows both X and Y, $\mathcal{E}$ of course knows the size of $X \cap Y$, denoted as m. In other words, the environment $\mathcal{E}$ knows the number of 1's in $\{b_1, \cdots, b_{n_1}\}$ is m.

The environment $\mathcal{E}$ instructs the dummy adversary $\mathcal{A}$ to corrupt the receiver at the beginning of the protocol execution, and then chooses a time t. Finally, if the number of 1's in $\{b_1, \cdots, b_{n_1}\}$ reported by the dummy adversary $\mathcal{A}$ is m and the message FINISHED has not been reported by the honest sender S at the time t, the environment $\mathcal{E}$ outputs 1, otherwise, outputs 0.

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

The ideal world execution. In the ideal world, since the (dummy) receiver is corrupted, the simulator Sim is allowed to access the ideal state $state_{\mathsf{R}} = \langle Y \rangle$. After receiving from the functionality $\mathcal{F}_{\mathsf{PSU}}$ the message $\langle \text{REQUEST}, \mathsf{R} \rangle$, the simulator Sim *must* face the following two simulation strategies:

1. *Do send* $\langle$RESPONSE, $OK\rangle$ *to the functionality* $\mathcal{F}_{\mathsf{PSU}}$*, immediately.*

   Note that, now the functionality will update the ideal states into $state_{\mathsf{S}} := \langle X, \text{FINISHED}\rangle$ and $state_{\mathsf{R}} := \langle Y, Z\rangle$, and immediately report FINISHED to the environment $\mathcal{E}$.
2. *Do not send* $\langle$RESPONSE, $OK\rangle$ *to the functionality* $\mathcal{F}_{\mathsf{PSU}}$*, immediately.*

   Note that, now the functionality will **not** update the ideal states into $state_{\mathsf{S}} := \langle X, \text{FINISHED}\rangle$ and $state_{\mathsf{R}} := \langle Y, Z\rangle$; of course, no output FINISHED will be reported to $\mathcal{E}$ immediately.
   Note again that, the simulator Sim knows nothing about Z in this case, although it is aware of the ideal state $state_{\mathsf{R}} = \langle Y\rangle$. At the same time, Sim has to simulate the view of $\mathcal{A}$ before the executions of Π_{OT} sub-protocol instances. Let $\{b_1^{\mathsf{ideal}}, \cdots, b_{n_1}^{\mathsf{ideal}}\}$ denote the bit set in the view of $\mathcal{A}$, which will be reported to the environment.

Security analysis. Now, we can see, if the simulator follows the first simulation strategy, the environment will tell the difference with probability 1, since in the real world, no output FINISHED will be reported while there is FINISHED in the ideal world.

If the simulator follows the second simulation strategy, the probability that $\mathcal{E}$ outputs 1 in the real world is 1 except negligible probability. However, in the ideal world, the simulator Sim does not know m. The probability that there are m 1's in $\{b_1^{\mathsf{ideal}}, \cdots, b_{n_1}^{\mathsf{ideal}}\}$ is $1/n_1$, which is far less than 1 when n_1 is large enough. Thus, the probability that $\mathcal{E}$ outputs 1 in the ideal world is far less than 1. Therefore, $\mathcal{E}$ can distinguish between the two worlds with non-negligible probability.

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

5.3 AHE-based PSU: Can Realize $\mathcal{F}_{\mathsf{PSU}}$

As analyzed in previous sections, the existing PSU protocols relying only on symmetric-key operations cannot UC-realize the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ due to the leverage of the Oblivious Transfer. More specifically, in the OT-based protocols (see Fig. 11), the receiver *before* receiving the final output $Z = X \cup Y$, obtains a bit b_i for each item $x_i \in X$; if $x_i \in Y$, $b_i := 1$, otherwise, $b_i := 0$. Obviously, each b_i reveals the set membership of $x_i \in X$, indicating whether x_i belongs to the set Y. However, the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$ is *not allowed to leak anything* to the receiver prior to sending the final output $Z := X \cup Y$. Therefore, to realize the functionality $\mathcal{F}_{\mathsf{PSU}}$, it seems necessary for a protocol to enable the receiver to obtain the output $Z := X \cup Y$ *directly*, rather than by first getting the set membership leakage $\{b_i\}$ and then obtaining $X \setminus Y$ obliviously.

We observe that the typical PSU protocols in [Fri07, DC17], both of which are based on additively homomorphic encryption (AHE), allow the receiver to obtain the output $Z := X \cup Y$ directly, and they are the only two protocols that satisfy the property. In this section, we first unify them into the same framework (as given in Fig. 12) and then show the protocols under this framework can UC-realize the standard PSU functionality $\mathcal{F}_{\mathsf{PSU}}$.

5.3.1 $\Pi^{\mathsf{AHE}}_{\mathsf{Unified}}$: AHE-based PSU design framework

In this part, we present a design framework $\Pi^{\mathsf{AHE}}_{\mathsf{Unified}}$ in Fig. 12 based on the additively homomorphic encryption (AHE), by unifying the protocols in [Fri07, DC17]. More specifically, the receiver first generates a representation of her input set Y; in [Fri07] Y is represented by a polynomial $P(x) = \prod_{i=1}^{n_2}(x - y_i)$ s.t. $P(x^*) = 0$ if $x^* \in Y$, and in [DC17] it is represented by an inverted Bloom Filter [10] B that inserts Y by using hash functions $h_1, \cdots, h_\gamma$. Since the schemes in [Fri07] and [DC17] share the similar idea, we use $f_Y(\cdot)$ to denote the representation of Y. For an item $x \in Y$, $f_Y(x) = 0$, otherwise $f_Y(x) \neq 0$. Therefore, $f_Y(\cdot)$ can be used to check if $x \in Y$. Then the receiver generates a key pair (pk, sk) of AHE and shares

[10] The "inverted" means each entry value of the Bloom Filter containing Y is flipped, s.t. $\sum_{i=1}^{\gamma} B[h_i(x^*)] = 0$ if $x^* \in Y$.

the public key pk with the sender. By using pk, the receiver encrypts $f_Y(\cdot)$ into $\mathsf{Enc}(f_Y)$ [11] and sends the ciphertext to the sender. Due to the additive homomorphic property, the sender holding pk can compute $c_i = (\mathsf{Enc}_{pk}(r_i f_Y(x_i)), \mathsf{Enc}_{pk}(r_i x_i f_Y(x_i)))$ for each $x_i \in X$, where r_i is a random value chosen by the sender. Upon receiving c_i, the receiver can decrypt it into (d_i^1, d_i^2). If $d_i^1 \neq 0$, the receiver learns $x_i \notin Y$ (i.e., $b_i = 0$), otherwise $x_i \in Y$ (i.e., $b_i = 1$). At last, when $d_i^1 \neq 0$, the receiver can obtain $x_i = d_i^2/d_i^1$, otherwise learn nothing about x_i from $(d_i^1, d_i^2) = (0, 0)$.

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

Parameters:

- Let n_1 and n_2 denote the set size for the sender S's and the receiver R's input set, respectively; Let ℓ be the bit-length of each item in the sender's set and the receiver's set;
- An AHE scheme includes an encryption algorithm $\mathsf{Enc}_{pk}(\cdot)$ and a decryption algorithm $\mathsf{Dec}_{sk}(\cdot)$.

Inputs:

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

Protocol:

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

Figure 12: The design framework unifying PSU protocols in [Fri07, DC17]. Note that $f_Y(\cdot)$ in [Fri07] is a polynomial $P(x) = \prod_{i=1}^{n_2}(x - y_i)$ such that $P(x^*) = 0$ if $x^* \in Y$, and $f_Y(\cdot)$ in [DC17] is an inverted Bloom Filter B where Y is inserted by using hash functions $h_1, \cdots, h_\gamma$ such that $\sum_{i=1}^{\gamma} B[h_i(x^*)] = 0$ if $x^* \in Y$ (the "inverted" means that each bit value of the Bloom Filter containing Y is flipped).

5.3.2 AHE-based PSU can realize $\mathcal{F}_{\mathsf{PSU}}$

As in the protocols [GMR+21, JSZ+22], the AHE-based protocols also process the receiver's set Y at the same time, and thus can avoid leaking subsets. Moreover, the receiver in the AHE-based protocols can obtain the items in $X \setminus Y$ directly from the corresponding ciphertexts without interacting with the sender anymore. Thus, the receiver does not obtain b_i in advance. Intuitively, the AHE-based protocols can UC-realize $\mathcal{F}_{\mathsf{PSU}}$. Next, we give a formal analysis.

Theorem 5.6. *Given an IND-CPA secure AHE scheme, the protocol following the framework $\Pi^{\mathsf{AHE}}_{\mathsf{Unified}}$ in Fig. 12 UC-realizes the functionality $\mathcal{F}_{\mathsf{PSU}}$ (as in Fig. 7), against static, semi-honest adversaries.*

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

Corrupted Sender: The simulator Sim for the corrupted sender first sends the input set X to $\mathcal{F}_{\mathsf{PSU}}$. After receiving $\langle \text{REQUEST}, \mathsf{S} \rangle$ from $\mathcal{F}_{\mathsf{PSU}}$, Sim generates a key pair (pk, sk) and sends pk to $\mathcal{A}$. To simulate the ciphertext c from R, Sim randomly generates a $f_Y(\cdot)$ according to the set size n_2 of Y, then encrypts it to c by using pk and sends c to $\mathcal{A}$. After $\mathcal{A}$ sends back $\{c_1, \cdots, c_{n_1}\}$ to R, Sim sends $\langle \text{RESPONSE}, \text{OK} \rangle$ to $\mathcal{F}_{\mathsf{PSU}}$. We can see that the only difference between the ideal world and the real world is that $f_Y(\cdot)$ is randomly

[11] In [Fri07], $\mathsf{Enc}(f_Y)$ refers to encrypting each coefficient of the polynomial $P(\cdot)$. In [DC17], $\mathsf{Enc}(f_Y)$ refers to encrypting each bit value of the inverted Bloom Filter B.

generated in the ideal world while $f_Y(\cdot)$ is generated based on Y in the real world. The IND-CPA security of AHE scheme guarantees that any PPT environment $\mathcal{E}$ cannot distinguish between the real world from the ideal world.

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

6 Conclusion

In this work, we provide a systematic treatment for understanding the security of the typical PSU protocols. More concretely, we define different PSU functionalities in a more fine-grained manner, in order to reflect the practical leakage due to the "split-execute-assemble" paradigm and Oblivious Transfer. Moreover, we unify the typical PSU protocols into KRTW, OT-based and AHE-based PSU frameworks, and prove what functionality each PSU framework can achieve. It is shown that only the protocols following the AHE-based framework can UC-realize the standard PSU functionality. However, the AHE-based protocols are not efficient enough in practice, especially for large datasets. Therefore, how to design a scalable PSU protocols that can UC-realize the standard PSU functionality is still an open problem.

References

[Can00] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. Cryptology ePrint Archive, Report 2000/067, 2000. https://eprint.iacr.org/2000/067.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[CLOS02] Ran Canetti, Yehuda Lindell, Rafail Ostrovsky, and Amit Sahai. Universally composable two-party and multi-party secure computation. In *34th ACM STOC*, pages 494–503. ACM Press, May 2002.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part III*, volume 12172 of *LNCS*, pages 34–63. Springer, Heidelberg, August 2020.

[DC17] 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.

[Fri07] 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.

[GMR+21] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In Juan Garay, editor, *PKC 2021, Part II*, volume 12711 of *LNCS*, pages 591–617. Springer, Heidelberg, May 2021.

[IKNP03] 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.

[IR89] 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.

[JSZ+22] 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.

[KKRT16] 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.

[KRTW19] 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.

[MR95] Rajeev Motwani and Prabhakar Raghavan. *Randomized algorithms*. Cambridge university press, 1995.

[MS13] 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.

[PR01] 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.

[PRTY19] 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.

[PSSZ15] 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.

[PSZ14] 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.

[PSZ18] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. *ACM Trans. Priv. Secur.*, 21(2):7:1–7:35, 2018.

[Rab05] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. https://eprint.iacr.org/2005/187.

A Simple Hashing and Cuckoo Hashing

Simple Hashing. In the simple hashing scheme, there are γ hash functions $h_i : \{0,1\}^* \to [b]$, where $i \in [\gamma]$, used to map n items into b bins $B_1, \cdots, B_b$. An item x will be added into $B_{h_1(x)}, B_{h_2(x)}, \cdots, B_{h_\gamma(x)}$, regardless of whether these bins are empty. According to the following inequality [MR95], 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 [PR01]. In this hashing scheme, there are γ hash functions $h_1, \cdots, h_\gamma$ used to map n items into $b = \epsilon n$ 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 [PSZ18], 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}.

B Leakage analysis for the KRTW-PSU protocol

To be self-contained, we recall the leakage analysis in [JSZ+22] here.

In order to improve the performance, Kolesnikov et al. [KRTW19] proposed to optimize their protocol by using the "split-execute-assemble" paradigm, as shown in Fig. 13. More specifically, the sender and receiver in [KRTW19] 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 both m. Then they perform a PSU sub-protocol on the items of each bin separately. As pointed out by Kolesnikov et al. in [KRTW19], however, *the "split-execute-assemble" paradigm will leak the information "which bins contain items in $X \cap Y$" to the receiver.* To avoid this leakage, in [KRTW19] the receiver is required to put a special item e into each bin, and to pad the bins with different dummy items $d_1, d_2, \cdots$, while the sender pads his bins with the special item e. For example, in Fig. 13, 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 e, 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 PSU sub-protocol. By adding the special item e to both sides, if the receiver learns that an item from the sender belongs to $\{y_3, e, y_8, d\}$, it seems that the receiver cannot learn whether the item is a real item (namely, in X) or the special item e. Unfortunately, Jia et al. [JSZ+22] pointed out that this strategy is insufficient to avoid the leakage incurred by the "split-execute-assemble" paradigm, and the detailed analysis is given below.

For ease of exposition, we take the 4th PSU sub-protocol instance in Fig. 13 as an example to explain why the optimization in [KRTW19] 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., $\{d, e, y_5, y_7\}$), 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 Fig. 13) 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 e).

The probabilities that Case_1 and Case_2 occur are denoted as $\Pr[\mathsf{Case}_1]$ and $\Pr[\mathsf{Case}_2]$, respectively. Clearly, if the receiver is able to determine that Case_1 occurs with certain (high) probability, she will learn that items

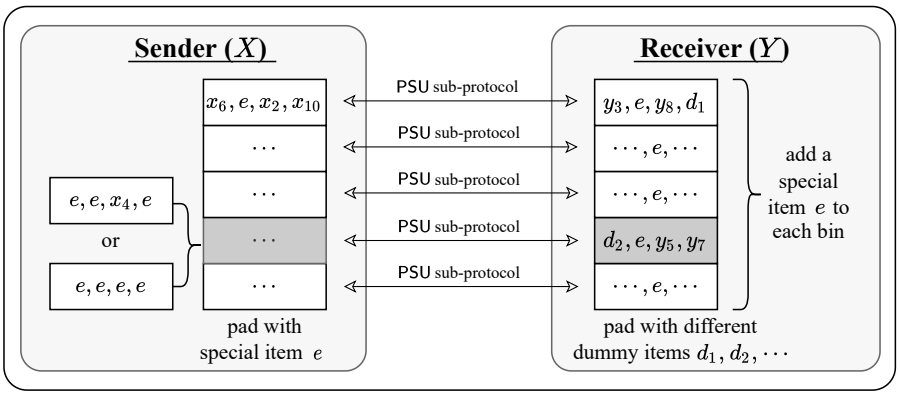

Figure 13: The "split-execute-assemble" paradigm in [KRTW19].

belonging to $X \cap Y$ are in $\{y_5, y_7\}$ with the same probability. According to the parameters in [KRTW19], Jia et al. [JSZ+22] estimated $\Pr[\mathsf{Case}_2]$ in Table 1. Note that $\Pr[\mathsf{Case}_1] = 1 - \Pr[\mathsf{Case}_2]$. From the results, we can see that the probability $\Pr[\mathsf{Case}_2]$ is very small. For example, when the set size is $n = 2^{20}$, $\Pr[\mathsf{Case}_2] = 5.778 \times 10^{-8}$. This means that when the receiver finds that all items in a bin belong to the intersection, she can learn that this bin has at least one real item with probability $1 - 5.778 \times 10^{-8}$, and that her corresponding bin contains at least an item in $X \cap Y$ with the same probability. Hence, their approach is insufficient to avoid the leakage incurred by the "split-execute-assemble" paradigm.

Table 1: 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.

C Details of Sub-protocols

We describe here the "batched related-key OPRF" sub-protocol $\Pi_{\mathsf{BaRK\text{-}OPRF}}$, the "Permute + Share" sub-protocol Π_{PS}, the "multi-point OPRF" sub-protocol Π_{mpOPRF} and the "generalized Reversed Private Membership Test" sub-protocol $\Pi_{\mathsf{g\text{-}RPMT}}$.

C.1 Sub-protocol $\Pi_{\mathsf{BaRK\text{-}OPRF}}$

Kolesnikov et al. [KRTW19] used the batched related-key OPRF protocol designed in Kolesnikov et al. [KKRT16] to implement $\Pi_{\mathsf{BaRK\text{-}OPRF}}$, and the details are shown in Fig. 14. This protocol is suitable for batch generation of a large number of OPRF instances.

Roughly speaking, Kolesnikov et al. [KKRT16] viewed the OT extension in the paradigm of IKNP [IKNP03] from a new angle; the secret key corresponding to a selection string x can be seen as the PRF value of the selection string x. In the j-th OPRF instance, P_0 can obtain the key $k_j = ((C, \boldsymbol{s}), (j, \boldsymbol{q}_j))$ and P_1 can obtain the PRF value of his value x_j, i.e., $F(k_j, x_j) = H(j||\boldsymbol{t}_j)$. Note that $F(k_j, x_j) = H(j||\boldsymbol{t}_j) = H(j||(\boldsymbol{q}_j \oplus (C(x_j) \cdot \boldsymbol{s})))$. Then, P_0 holding k_j can obtain the PRF value of any item y, i.e., $F(k_j, y) = H(j||(\boldsymbol{q}_j \oplus (C(y) \cdot \boldsymbol{s})))$. As mentioned in [KRTW19], the protocol in [KKRT16] actually achieves a slightly weaker variant of OPRF, since that (1) the keys are related as $(C, \boldsymbol{s})$ is the same in all keys, and (2) the protocol additionally reveals $\boldsymbol{t}_j$ to P_1 besides the PRF output $H(j||\boldsymbol{t}_j)$. However, Kolesnikov et al. [KRTW19] stressed that the protocol in [KKRT16] is sufficient to be used in their scheme.

Sub-Protocol $\Pi_{\mathsf{BaRK\text{-}OPRF}}$

Parameters:

- A (κ, ϵ) pseudorandom code family $\mathcal{C}$ with output length $k = k(\kappa)$ where κ is the computational security parameter and a (κ, ϵ) pseudorandom code guarantees that the hamming distance of two codewords is less or equal to κ with probability at most $2^{-\epsilon}$;
- A κ-Hamming correlation-robust $H : [m] \times \{0,1\}^k \to \{0,1\}^v$;
- Ideal functionality $\mathcal{F}_{\mathsf{OT}}$.

Inputs:

- P_0: no input;
- P_1: set $X = \{x_1, x_2, \cdots, x_m\}$, where $x_i \in \{0,1\}^*$.

Protocol:

1. P_0 chooses a random $C \leftarrow \mathcal{C}$ and sends it to P_1;
2. P_0 chooses $\boldsymbol{s} \leftarrow \{0,1\}^k$ at random. Let s_i denote the i-th bit of $\boldsymbol{s}$;
3. P_1 forms $m \times k$ matrices T_0, T_1 in the following way:
   - For $j \in [m]$, choose $\boldsymbol{t}_{0,j} \leftarrow \{0,1\}^k$ and set $\boldsymbol{t}_{1,j} = C(x_j) \oplus \boldsymbol{t}_{0,j}$. Let $\boldsymbol{t}_0^i, \boldsymbol{t}_1^i$ denote the i-th column of matrices T_0, T_1 respectively.
4. For $i \in [k]$:
   - P_0 and P_1 invoke $\mathcal{F}_{\mathsf{OT}}$:
     - P_0 acts as receiver with input s_i;
     - P_1 acts as sender with input $(\boldsymbol{t}_0^i, \boldsymbol{t}_1^i)$;
     - P_0 receives output $\boldsymbol{q}^i$;
5. P_0 forms $m \times k$ matrix Q such that the i-th column of Q is the vector $\boldsymbol{q}^i$. Note that $\boldsymbol{q}^i = \boldsymbol{t}_{s_i}^i$. Let $\boldsymbol{q}_j$ denote the j-th row of Q. Note that $\boldsymbol{q}_j = ((\boldsymbol{t}_{0,j} \oplus \boldsymbol{t}_{1,j}) \cdot \boldsymbol{s}) \oplus \boldsymbol{t}_{0,j}$, i.e., $\boldsymbol{q}_j = (C(x_j) \cdot \boldsymbol{s}) \oplus \boldsymbol{t}_{0,j}$.
6. For $j \in [m]$, P_0 outputs the PRF key $((C, \boldsymbol{s}), (j, \boldsymbol{q}_j))$, P_1 outputs relaxed PRF output $(C, j, \boldsymbol{t}_{0,j})$ and PRF output $H(j||\boldsymbol{t}_{0,j})$.

Figure 14: The batched related-key OPRF Protocol designed in [KKRT16].

C.2 Sub-protocol Π_{PS}

Jia et al. [JSZ+22] used the protocol in [MS13] to implement Π_{PS}, and the details are shown in Fig. 15. Generally speaking, the protocol leverages a switching network to realize the permutation, and the random labels of wires are used to form secret shares. A switching network consists of q switches and $2q+n$ wires where n is the number of items to be permuted. The party P_0 who inputs the permutation π will transfer π into a selection bit set S, in which each item is used to control a switch. The other party P_1 randomly chooses the label of each wire, and uses the labels of input wires to mask the input set $X = \{x_1, \cdots, x_n\}$. The masked values are then sent to P_0 and taken as the switching network's input. Then, in topological order, the two parties jointly compute an atomic swap on each switch. And each atomic swap is implemented by using oblivious transfer according to the corresponding selection bit in the set S. After this, set X is permuted to $\pi(X)$, and each share is re-randomized by the labels in the path. At last, P_0 obtains the blinded values for all the output wires as the share set, and P_1 uses the labels of output wires as the other share set.

Sub-Protocol Π_{PS}

Parameters:

- A switching network with q switches and n inputs/outputs, and the $2q+n$ wires are denoted as $w_1, \cdots, w_{2q+n}$.

Inputs:

- P_0: a permutation π;
- P_1: set $X = \{x_1, \cdots, x_n\}$, where $x_i \in \{0,1\}^\ell$.

Protocol:

1. P_0 transfers the permutation π to the selection bit set $S = \{(s_0(1), s_1(1)), \cdots, (s_0(q), s_1(q))\}$ for the switching network;
2. For each wire, P_1 randomly chooses $r_i \leftarrow \{0,1\}^\ell$ as the wire's label;
3. For each switch u with input wires w_i and w_j, and output wires w_k and w_l:
   - P_1 computes $T_0 = T_0^0 || T_0^1 = (r_i \oplus r_k) || (r_i \oplus r_l)$, $T_1 = T_1^0 || T_1^1 = (r_i \oplus r_k) || (r_j \oplus r_l)$, $T_2 = T_2^0 || T_2^1 = (r_j \oplus r_k) || (r_i \oplus r_l)$ and $T_3 = T_3^0 || T_3^1 = (r_j \oplus r_k) || (r_j \oplus r_l)$;
   - P_0 and P_1 engage in a 1-out-of-4 oblivious transfer where P_0's (the receiver) input is $s(u) = 2s_1(u) + s_0(u)$ and P_1's (the sender) input is (T_0, T_1, T_2, T_3), and P_0 obtains $(T^0_{s(u)} || T^0_{s(u)})$;
4. For each input wire w_i, P_1 sends $x_i \oplus r_i$ to P_0;
5. In topological order, for each switch u with input wires w_i and w_j, and output wires w_k and w_l, P_0 does the following:
   - If $s_0(u) = 0$, then $y_k = y_i \oplus T^0_{s(u)}$, else, $y_k = y_j \oplus T^0_{s(u)}$;
   - If $s_1(u) = 0$, then $y_l = y_i \oplus T^1_{s(u)}$, else, $y_l = y_j \oplus T^1_{s(u)}$;
6. P_0 outputs the blinded values for all the output wires of the switching network as the share set, and P_1 outputs the output wires' labels as the other share set.

Figure 15: The Permute + Share Protocol designed in [MS13].

C.3 Sub-protocol Π_{mpOPRF}

Chase and Miao in [CM20] proposed a new construction for multi-point OPRF, and they leveraged the new construction to realize a lightweight private set intersection (PSI) protocol. In essence, the construction in [CM20] is an extension of BaRK-OPRF in [KKRT16], and Chase and Miao have pointed it out in their paper. To be self-contained, we rewrite their construction in Fig. 16. The key of the multi-point OPRF is $(\boldsymbol{C}, \hat{k})$ and the pseudorandom function is as follows:

$$\boldsymbol{v} = \hat{F}(\hat{k}, H_1(x_i))$$
$$F((\boldsymbol{C}, \hat{k}), x_i) = H_2(\boldsymbol{C}^1[\boldsymbol{v}[1]] || \cdots || \boldsymbol{C}^w[\boldsymbol{v}[w]])$$

Generally speaking, the two parties both know a pseudorandom function $\hat{F}$ with key $\hat{k}$ that maps a ℓ_1-bit item into a vector $\boldsymbol{v} \in [m]^w$. Firstly, P_1 prepares two $m \times w$ binary matrices $\boldsymbol{A}$ and $\boldsymbol{B}$. More concretely, all the items in X will be mapped to some positions in a $m \times w$ matrix (i.e., $\boldsymbol{D_X}$ in Fig. 16) by $\hat{F}$. $\boldsymbol{A}$ is randomly

chosen, and in $\boldsymbol{B}$, the 1-bit elements that are located in these position are equal to the corresponding values in $\boldsymbol{A}$ while the other elements will be different. P_0 picks a random string $s \in \{0,1\}^w$, then P_0 and P_1 perform w number of OTs. For the i-th OT, P_0 takes $s[i]$ as input, and P_1 takes $\boldsymbol{A}^i$ and $\boldsymbol{B}^i$ as input, then P_0 will obtain output $\boldsymbol{A}^i$ or $\boldsymbol{B}^i$ according to $s[i]$. After OTs, P_0 will obtain w column vectors, which will form matrix $\boldsymbol{C}$. P_1 obtains PRFs of her items using $\boldsymbol{A}$ and $\hat{k}$. Note that for $\forall\ x \in X$, $F((\boldsymbol{C}, \hat{k}), x) = F((\boldsymbol{A}, \hat{k}), x)$. The protocol needs $O(n)$ communication and computation cost, and only involves cheap symmetric-key and bitwise operations.

Sub-Protocol Π_{mpOPRF}

Parameters:

- Two hash functions $H_1 : \{0,1\}^* \to \{0,1\}^{\ell_1}$ and $H_2 : \{0,1\}^w \to \{0,1\}^{\ell_2}$;
- Pseudorandom function $\hat{F} : \{0,1\}^\lambda \times \{0,1\}^{\ell_1} \to [m]^w$;

Inputs:

- P_0: no input;
- P_1: set $X = \{x_1, \cdots, x_n\}, x_i \in \{0,1\}^*$;

Protocol:

1. P_0 samples a random string $s \leftarrow \{0,1\}^w$;
2. P_1 does the following:
   - Initialize an $m \times w$ binary matrix $\boldsymbol{D}$ to all 1's. Denote its column vectors by $\boldsymbol{D}^1, \cdots, \boldsymbol{D}^w$. Then $\boldsymbol{D}^1 = \cdots = \boldsymbol{D}^w = \boldsymbol{1}^m$;
   - Sample a uniformly random PRF key $\hat{k} \leftarrow \{0,1\}^\kappa$, and send it to P_0;
   - For each $x \in X$, compute $\boldsymbol{v} = \hat{F}(\hat{k}, H_1(x))$ where $\boldsymbol{v} = (\boldsymbol{v}[1], \boldsymbol{v}[2], \cdots, \boldsymbol{v}[w])$ with the i-th coordinate $\boldsymbol{v}[i] \in [m]$, and set $\boldsymbol{D}^i[\boldsymbol{v}[i]] = 0$ for all $i \in [w]$, then denote the new matrix as $\boldsymbol{D}_{\boldsymbol{X}}$;
   - Randomly sample an $m \times w$ binary matrix $\boldsymbol{A}$, and compute matrix $\boldsymbol{B} = \boldsymbol{A} \oplus \boldsymbol{D}_{\boldsymbol{X}}$;
3. P_0 and P_1 run w oblivious transfer where P_1 is the sender with inputs $\{\boldsymbol{A}^i, \boldsymbol{B}^i\}_{i \in [w]}$ and P_0 is the receiver with inputs $s[1], \cdots, s[w]$. As a result P_0 obtains w number of m-bit strings as the column vectors of matrix $\boldsymbol{C}$ (with dimension $m \times w$). So far, P_0 obtains the key $(\boldsymbol{C}, \hat{k})$ of the multi-point OPRF.
4. For each $x \in X$, P_1 computes $\boldsymbol{v} = \hat{F}(\hat{k}, H_1(x))$ and obtains its OPRF value $H_2(\boldsymbol{A}^1[\boldsymbol{v}[1]] || \cdots || \boldsymbol{A}^w[\boldsymbol{v}[w]])$.

Figure 16: The Multi-Point OPRF Protocol designed in [CM20].

C.4 Sub-protocol $\Pi_{\text{g-RPMT}}$

C.4.1 Sub-protocol $\Pi_{\text{g-RPMT}}$ in [KRTW19]

Firstly, the two parties invoke the "batched related-key OPRF" sub-protocol $\Pi_{\mathsf{BaRK\text{-}OPRF}}$ for a pseudorandom function F; the sender takes the set $\tilde{X}_i$ as input and obtains $F(k_j, x_j)$ for each $x_j \in \tilde{X}_i$ (without knowing k_j), and the receiver obtains the PRF keys $\{k_1, k_2, \cdots, k_m\}$. Then for each sender's item x_j, the receiver picks a random value s and interpolates a polynomial P over points $\{(y, s \oplus F(k_j, y))\}_{y \in \tilde{Y}_i}$, and sends P to the sender. Once receiving the polynomial P, the sender calculates $s_j = P(x_j) \oplus F(k_j, x_j)$ and returns s_j to the receiver. The receiver then checks if $s_j = s$; if not, meaning that $x_j \notin \tilde{Y}_i$, the receiver sets $b_j = 0$, otherwise $b_j = 1$. To achieve "generalized" RPMT, the two parties repeat the above process for all the items in the sender's set.

C.4.2 Sub-protocol $\Pi_{\text{g-RPMT}}$ in [JSZ+22]

The sub-protocol $\Pi_{\text{g-RPMT}}$ uses a simple way to check whether an item x_i is in the receiver's set Y: Each item $y_j \in Y$ are shared into shares s_j^1 and s_j^2 such that $y_j = s_j^1 \oplus s_j^2$. Then, the sender obtains one share set $S_1 = \{s_1^1, s_2^1, \cdots, s_{n_2}^1\}$, and the receiver holds the other one $S_2 = \{s_1^2, s_2^2, \cdots, s_{n_2}^2\}$. The sender can obtain $I_i = \{s_1^1 \oplus x_i, s_2^1 \oplus x_i, \cdots, s_{n_2}^1 \oplus x_i\}$. Obviously, if $x_i \in Y$, $I_i \cap S_2 \neq \emptyset$. Therefore, the sender can send I_i to the receiver and the receiver checks if $I_i \cap S_2 \neq \emptyset$. If so, the receiver obtains the bit $b_i = 1$, otherwise,

Sub-protocol $\Pi_{\text{g-RPMT}}$ in [KRTW19]

Parameters:

- Let m_1 and m_2 denote the set size for the sender $\tilde{\mathsf{S}}$'s and the receiver $\tilde{\mathsf{R}}$'s input set, respectively; let ℓ be the bit-length of each item in the sender's set or the receiver's set.
- Let $h(\cdot)$ be a hash function $h : \{0,1\}^{\ell} \to \{0,1\}^{\sigma}$;

Inputs:

- Sender $\tilde{\mathsf{S}}$: set $X' = \{x_1, \cdots, x_{m_1}\}$, where $x_i \in \{0,1\}^{\ell}$;
- Receiver $\tilde{\mathsf{R}}$: set $Y' = \{y_1, \cdots, y_{m_2}\}$, where $y_i \in \{0,1\}^{\ell}$;

Protocol:

1. The sender $\tilde{\mathsf{S}}$ with input set set X' and the receiver $\tilde{\mathsf{R}}$ invoke the "batched related-key OPRF" sub-protocol $\Pi_{\text{BaRK-OPRF}}$, where $F : \{0,1\}^{*} \times \{0,1\}^{\ell} \to \{0,1\}^{\sigma}$ is the underlying pseudorandom function, please refer to Fig. 14 for the description of $\Pi_{\text{BaRK-OPRF}}$. After $\Pi_{\text{BaRK-OPRF}}$, for each $i \in [m_1]$, $\tilde{\mathsf{S}}$ receives $q_i = F(k_i, x_i)$ and $\tilde{\mathsf{R}}$ receives k_i;
2. For each $x_i \in X'$:
   - $\tilde{\mathsf{R}}$ randomly picks $s \xleftarrow{\$} \{0,1\}^{\sigma}$, and interpolates a polynomial $P(y)$ over points $\{(h(y_j), s \oplus q_j)\}_{j \in [m_2]}$; here $s \oplus q_j$ is computed as the XOR operation on σ-bit strings.
   - $\tilde{\mathsf{R}}$ sends the coefficients of $P(y)$ to $\tilde{\mathsf{S}}$;
   - $\tilde{\mathsf{S}}$ computes $s_i \coloneqq P(h(x_i)) \oplus q_i$ and sends it to $\tilde{\mathsf{R}}$;
   - If $s_i = s$, then $\tilde{\mathsf{R}}$ sets $b_i \coloneqq 1$, otherwise, $b_i \coloneqq 0$;
3. $\tilde{\mathsf{R}}$ outputs $\{b_1, \cdots, b_{m_1}\}$.

Figure 17: Sub-protocol $\Pi_{\text{g-RPMT}}$ in [KRTW19].

$b_i = 0$. However, if $I_i \cap S_2 = s_k^2$, the receiver learns that the item y_k in Y corresponding to s_k^2 belongs to intersection, which is not allowed in PSU. To solve the problem, the sender and receiver in [JSZ+22] invoke the "Permute + Share" sub-protocol Π_{PS}. In sub-protocol Π_{PS}, the receiver takes set Y as an input, and the sender selects a random permutation π as the other input. After Π_{PS}, the sender and receiver obtain the shuffled share sets $\hat{S}_1 = \{a'_1, a'_2, \cdots, a'_{n_2}\}$ and $\hat{S}_2 = \{a_1, a_2, \cdots, a_{n_2}\}$ respectively, where $a'_i \oplus a_i = y_{\pi(i)}$, and the receiver does not learn the permutation π. Please refer to Appendix C.2 for more details of sub-protocol Π_{PS}. In this way, the receiver cannot learn which item in Y corresponds to a_k. In addition, to hide x_i, the two parties invoke the "multi-point OPRF" sub-protocol Π_{mpOPRF} where the receiver takes $\hat{S}_2 = \{a_1, a_2, \cdots, a_{n_2}\}$ as an input. After Π_{mpOPRF}, the receiver obtains $\hat{S}_2^* = \{F(k, a_1), F(k, a_2), \cdots, F(k, a_{n_2})\}$ and the sender obtains PRF key k. Please refer to Appendix C.3 for more details of sub-protocol Π_{mpOPRF}. Then, the sender sends $I_i = \{F(k, a'_1 \oplus x_i), F(k, a'_2 \oplus x_i), \cdots, F(k, a'_{n_2} \oplus x_i)\}$ to the receiver. Likewise, if $I_i \cap \hat{S}_2^* \neq \emptyset$, the receiver obtains $b_i = 1$, otherwise, $b_i = 0$. For each $x_i \in X$, the sender can use the same $\hat{S}_1$ and k to generate the corresponding I_i. To improve the efficiency, [JSZ+22] leveraged Cuckoo hashing to reduce the size of each I_i to a constant γ, which is much smaller than n_2. Note that in protocol of [JSZ+22], although the set Y is inserted to a Cuckoo hashing, the set Y is still processed as a whole, rather than being split into multiple subsets as in [KRTW19].

C.4.3 Sub-protocol $\Pi_{\text{g-RPMT}}$ in [GMR+21]

The core idea to construct the sub-protocol $\Pi_{\text{g-RPMT}}$ in [GMR+21] is similar to that in [KRTW19]. However, the sub-protocol $\Pi_{\text{g-RPMT}}$ in [GMR+21] avoids the repetitive high-degree polynomial interpolations in the sub-protocol $\Pi_{\text{g-RPMT}}$ constructed in [KRTW19] by using the shuffling technique.

More specifically, the sender inserts his input set X to a Cuckoo hash table with b bins by using three hash functions h_1, h_2, h_3, and the receiver inserts her input set Y to a simple hash table with b bins by using the same hash functions. We denote the filled Cuckoo hash table as X_C. After performing the "batched related-key OPRF" sub-protocol, for the jth bin, the sender obtains a PRF value f_j of the item $x \in X$ assigned to this bin and the receiver obtains the corresponding PRF key k_j. Then, for each bin, the receiver randomly chooses a value s_j where $j \in [b]$. The receiver interpolates a polynomial P such that $P(y||i) = s_{h_i(y)} \oplus \mathsf{PRF}(k_{h_i(y)}, y||i)$ where $y \in Y$ and $i \in \{1,2,3\}$, and sends P to the sender. Given the polynomial P, the

Sub-protocol $\Pi_{\text{g-RPMT}}$ in [JSZ+22]

Parameters:

- Let m_1 and m_2 denote the set size for the sender $\tilde{\mathsf{S}}$'s input set, and for the receiver $\tilde{\mathsf{R}}$'s input set, respectively; Let ℓ_1 be the bit-length of each item in the sender's set or the receiver's set;
- Let $h_1(\cdot), \cdots, h_\gamma(\cdot)$ be hash functions $h_i : \{0,1\}^{\ell_1} \to [b]$, where $i \in [\gamma]$;
- A Cuckoo hash table without stash is based on $h_1, \cdots, h_\gamma$ and has $b = \epsilon \cdot m_2$ bins;

Inputs:

- Sender $\tilde{\mathsf{S}}$: set $X = \{x_1, \cdots, x_{m_1}\}$, where $x_i \in \{0,1\}^{\ell_1}$;
- Receiver $\tilde{\mathsf{R}}$: set $Y = \{y_1, \cdots, y_{m_2}\}$, where $y_i \in \{0,1\}^{\ell_1}$;

Protocol:

1. $\tilde{\mathsf{R}}$ inserts set Y into the Cuckoo hash table based on $h_1, \cdots, h_\gamma$, 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. $\tilde{\mathsf{S}}$ and $\tilde{\mathsf{R}}$ invoke the "Permute + Share" sub-protocol Π_{PS} shown in Fig. 15:
   - $\tilde{\mathsf{R}}$ acts as P_1 with input set Y_C, and $\tilde{\mathsf{S}}$ acts as P_0 with a permutation π;
   - $\tilde{\mathsf{R}}$ obtains the shuffled share set $\hat{S}_2 = \{a_1, a_2, \cdots, a_b\}$, and $\tilde{\mathsf{S}}$ obtains the other shuffled share set $\hat{S}_1 = \{a'_1, a'_2, \cdots, a'_b\}$ where $Y_C[\pi(i)] = a'_i \oplus a_i$;
3. $\tilde{\mathsf{S}}$ and $\tilde{\mathsf{R}}$ invoke the "multi-point OPRF" sub-protocol Π_{mpOPRF} shown in Fig. 16; let $F(\cdot,\cdot)$ be the underlying pseudorandom function $F : \{0,1\}^* \times \{0,1\}^{\ell_1} \to \{0,1\}^{\ell_2}$:
   - $\tilde{\mathsf{R}}$ acts as P_1 with her shuffled share set $\hat{S}_2 = \{a_1, a_2, \cdots, a_b\}$, and obtains the output $\hat{S}_2^* = \{F(k, a_1), F(k, a_2), \cdots, F(k, a_b)\}$;
   - $\tilde{\mathsf{S}}$ acts as P_0 and obtains the key k;
4. For $i \in [m_1]$:
   - $\tilde{\mathsf{S}}$ initializes sets $Q_i := \emptyset$ and $I_i := \emptyset$;
   - For $j \in [\gamma]$:
     - $\tilde{\mathsf{S}}$ 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, $r \leftarrow \{0,1\}^{\ell_2}$ and $I_i := I_i \cup \{r\}$;
5. $\tilde{\mathsf{S}}$ sends $\{I_1, I_2, \cdots, I_{m_1}\}$ to $\tilde{\mathsf{R}}$, then outputs FINISHED;
6. For each $i \in [m_1]$, $\tilde{\mathsf{R}}$ checks if $\hat{S}_2^* \cap I_i \neq \emptyset$; if so, $\tilde{\mathsf{R}}$ sets $b_i := 1$, otherwise, sets $b_i := 0$; then $\tilde{\mathsf{R}}$ outputs $\{b_1, \cdots, b_{m_1}\}$.

Figure 18: Sub-protocol $\Pi_{\text{g-RPMT}}$ in [JSZ+22].

sender can compute $t_j = P(X_C[j]) \oplus f_j$ where $j \in [b]$. We can see that if the item $X_C[j]$ is in the subset Y_j of set Y mapped to the jth bin of the simple hash table, $t_j = s_j$. However, if the sender sends t_j to the receiver directly, the receiver can learn that the subset Y_j has an item that is in the intersection $X \cap Y$, which is not allowed in PSU. Therefore, the two parties perform the "Permute + Share" sub-protocol Π_{PS} such that $\{s_1, \cdots, s_b\}$ are shared and permuted into $\{s_1^1, \cdots, s_b^1\}$ and $\{s_1^2, \cdots, s_b^2\}$ obtained by the sender and receiver, respectively, where $s_i^1 \oplus s_i^2 = s_{\pi(i)}$ and the permutation π is not known by the receiver. Obviously, if $s_{\pi(i)} = t_{\pi(i)}$, then $t_{\pi(i)} \oplus s_i^1 = s_i^2$. Therefore, the sender computes $\{a'_1, \cdots, a'_b\}$ where $a'_i = t_{\pi(i)} \oplus s_i^1$. Finally, through the "batched equality testing" ideal functionality $\mathcal{F}_{\mathsf{bEQ}}$ as shown in Fig. 20, the receiver can obtain $\{b_1, \cdots, b_b\}$; if $a'_i = s_i^2$, then $b_i = 1$, otherwise, $b_i = 0$.

Sub-protocol $\Pi_{\text{g-RPMT}}$ in [GMR+21]

Parameters:

- Let m_1 and m_2 denote the set size for the sender $\tilde{\mathsf{S}}$'s input set, and for the receiver $\tilde{\mathsf{R}}$'s input set, respectively; Let ℓ_1 be the bit-length of each item in the sender's set or the receiver's set;
- Let $h_1(\cdot), \cdots, h_3(\cdot)$ be hash functions $h_i : \{0,1\}^{\ell_1} \to [b]$, where $i \in [3]$;
- A Cuckoo hash table without stash is based on $h_1, \cdots, h_3$ and has $b = \epsilon \cdot m_2$ bins;
- Let $\mathcal{F}_{\text{bEQ}}$ denote the "batch equality testing" ideal functionality;

Inputs:

- Sender $\tilde{\mathsf{S}}$: set $X = \{x_1, \cdots, x_{m_1}\}$, where $x_i \in \{0,1\}^{\ell_1}$;
- Receiver $\tilde{\mathsf{R}}$: set $Y = \{y_1, \cdots, y_{m_2}\}$, where $y_i \in \{0,1\}^{\ell_1}$;

Protocol:

1. The sender $\tilde{\mathsf{S}}$ inserts set X into the Cuckoo hash table based on $h_1, \cdots, h_3$, and adds a dummy item d in each empty bin, then denotes the filled Cuckoo hash table as X_C and the item in i-th bin as $X_C[i]$;
2. The parties invoke the "batched related-key OPRF" sub-protocol $\Pi_{\text{BaRK-OPRF}}$ as shown in Fig. 14:
   - $\tilde{\mathsf{S}}$ acts as P_1 with input set X_C and $\tilde{\mathsf{R}}$ acts as P_0;
   - $\tilde{\mathsf{R}}$ receives output $(k_1, \cdots, k_b)$ and $\tilde{\mathsf{S}}$ receives output $(f_1, \cdots, f_b)$ such that, for each $x \in X$ assigned to jth bin by hash function h_i, we have $f_j = \mathsf{PRF}(k_j, x||i)$;
3. For each $j \in [b]$, $\tilde{\mathsf{R}}$ choose a random s_j;
4. $\tilde{\mathsf{R}}$ interpolates a polynomial P of degree $< 3m_2$ such that for every $y \in Y$ and $i \in \{1,2,3\}$, we have $P(y||i) = s_{h_i(y)} \oplus \mathsf{PRF}(k_{h_i(y)}, y||i)$. He sends P to $\tilde{\mathsf{S}}$;
5. $\tilde{\mathsf{S}}$ computes $\{t_1, \cdots, t_b\}$ where $t_j = P(X_C[j]) \oplus f_j$;
6. Recall that $\tilde{\mathsf{S}}$ places her m_1 items into m bins, with each item placed exactly once. Alice chooses a random permutation $\pi : [b] \to [b]$;
7. The parties invoke the "$\mathsf{Permute} + \mathsf{Share}$" sub-protocol Π_{PS} as shown in Fig. 15:
   - $\tilde{\mathsf{S}}$ acts as P with input π and $\tilde{\mathsf{R}}$ acts as sender with input $\{s_1, \cdots, s_b\}$;
   - $\tilde{\mathsf{S}}$ receives output $\{s_1^1, \cdots, s_b^1\}$ and $\tilde{\mathsf{R}}$ receives output $\{s_1^2, \cdots, s_b^2\}$, where $s_i^1 \oplus s_i^2 = s_{\pi(i)}$;
8. $\tilde{\mathsf{S}}$ locally computes $\{a'_1, \cdots, a'_b\}$ where $a'_i = s_i^1 \oplus t_{\pi(i)}$, so that a'_i and s_i^2 are secret shares of $s_{\pi(i)} \oplus t_{\pi(i)}$, i.e., $a'_i = s_i^2$ whenever $s_{\pi(i)} = t_{\pi(i)}$;
9. The parties invoke $\mathcal{F}_{\text{bEQ}}$, where $\tilde{\mathsf{S}}$ is sender with input $\{a'_1, \cdots, a'_b\}$ and $\tilde{\mathsf{R}}$ is receiver with input $\{s_1^2, \cdots, s_b^2\}$, then $\tilde{\mathsf{R}}$ receives output $\{b_1, \cdots, b_b\}$;
10. $\tilde{\mathsf{R}}$ outputs $\{b_1, \cdots, b_b\}$.

Figure 19: Sub-protocol $\Pi_{\text{g-RPMT}}$ in [GMR+21].

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

- The functionality interacts with two parties, the sender P_0 and the receiver P_1, and the simulator Sim;

Functionality:

0. Initialize an ideal state $state_{\mathsf{U}} := \emptyset$ for party U where $\mathsf{U} \in \{P_0, P_1\}$; if U is corrupted, the simulator Sim is allowed to access to U's state $state_{\mathsf{U}}$;
1. Upon receiving input set $X = \{x_1, \cdots, x_n\}$ from the sender P_0 where $x_i \in \{0,1\}^{\ell}$, update state $state_{P_0} := \langle X \rangle$, and send $\langle \text{REQUEST}, P_0 \rangle$ to the simulator Sim;
2. Upon receiving input set $Y = \{y_1, \cdots, y_n\}$ from the receiver P_1 where $y_i \in \{0,1\}^{\ell}$, update state $state_{P_1} := \langle Y \rangle$, and send $\langle \text{REQUEST}, P_1 \rangle$ to the simulator Sim;
3. Upon receiving $\langle \text{RESPONSE}, \text{OK} \rangle$ from Sim, for each $i \in [n]$, if $x_i = y_i$, set $b_i := 1$, otherwise, set $b_i := 0$; then add $\langle \text{FINISHED} \rangle$ to the sender's sate $state_{P_0}$ and $\langle \{b_1, \cdots, b_n\} \rangle$ to the receiver's state $state_{P_1}$;
4. Output $\{b_1, \cdots, b_n\}$ to P_1, and FINISHED to P_0.

Figure 20: The batched equality testing functionality.