Efficient Private Multiset ID Protocols [⋆]

Cong Zhang[1,2], Weiran Liu[3], Bolin Ding[3], and Dongdai Lin[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences
[2] School of Cyber Security, University of Chinese Academy of Sciences
`{zhangcong,ddlin}@iie.ac.cn`,
[3] Alibaba Group
`{weiran.lwr,bolin.ding}@alibaba-inc.com`

Abstract. Private-ID (PID) protocol enables two parties, each holding a private set of items, to privately compute a set of random universal identifiers (UID) corresponding to the records in the union of their sets, where each party additionally learns which UIDs correspond to which items in its set but not if they belong to the intersection or not. PID is very useful in the privacy computation of databases query, e.g. inner join and join for compute. Known PID protocols all assume the input of both parties is a set. In the case of join, a more common scenario is that one party's primary key (unique) needs to join the other party's foreign key (duplicate). How to construct an efficient Private Multiset ID (PMID) protocol to support the above *key-foreign key join* remains open.

We resolve this problem by constructing efficient PMID protocols from Oblivious PRF, Private Set Union, and a newly introduced primitive called Deterministic-Value Oblivious Programmable PRF (dv-OPPRF). We also propose some PMID applications, including Private Inner Join, Private Full Join, and Private Join for Compute.

We implement our PMID protocols and state-of-the-art PID protocols as performance baselines. The experiments show that the performances of our PMID are almost the same as the state-of-the-art PIDs when we set the multiplicity $U_x = U_y = 1$. Our PMID protocols scale well when either $U_x > 1$ or $U_y > 1$. The performances also correctly reflect excessive data expansion when both $U_x, U_y > 1$ for the more general *cross join* case.

1 Introduction

1.1 Motivation

A large number of services today collect valuable but sensitive data from the same group of users. These services could benefit from pooling their data together and jointly performing analytical tasks (e.g., filtering and aggregation) on the aligned data. For example, consider two parties Alice and Bob: Alice owns users' profile data where each record has four attributes (user_id, user_name, age, sex), and Bob collects users' transaction data with each record as (user_id, prod_id, prod_name, price). The identifier of a user, i.e., user_id, could be the username, e-mail address, or telephone number. The two parties want to, e.g., securely align their records on user_id (concatenating the records with matching user_id, one from each party), filter aligned records on age, and aggregate price, without exposing the identifiers and the values of records. Such alignment is called the *join* of two tables A and B in databases. Depending on whether to include records with unmatched identifiers in the join results, there are *inner joins* (including only those with matched identifiers) and *full joins* (otherwise).

One way to realize the above functionalities is to use Private Set Operation (PSO) protocols. For example, Private Set Intersection (PSI) [18, 10, 31, 24, 29, 6] offers a way to inner join two datasets and learn the intersection membership without revealing anything outside the intersection; Private Set Union (PSU) [23, 11, 7, 26, 22] could be used to compute full join of two datasets privately. To compute on the join result, Private Set Intersection Cardinality/Sum (PSI-CA/PSI-Sum) [20, 19] focus on computing the cardinality or linear functions of the intersection. Nevertheless, one PSO protocol only focuses on one scenario, and different protocol introduces different design ideas, e.g. DDH-based, OT-based, HE-based, which makes it difficult to unify them in a variety of application scenarios. Circuit-PSI/PSU [17, 16, 4], on the other hand, support any function computation on the

[⋆] This is the full version of a paper to be published in ICICS 2023.

intersection/union, since it outputs the secret shared result of intersection/union. Although circuit-PSI/PSU is more powerful than PSI/PSU, it is usually less efficient due to the use of the general MPC technique.

Private-ID (PID) protocol [5, 12] enables two parties, each holding a private set of items, to privately compute a set of random universal identifiers (UID) corresponding to the records in the union of their sets, where each party additionally learns which UIDs correspond to which items in its set but not if they belong to the intersection or not. Private-ID provides a unified method to construct the above PSO protocols. The main use of PID is to realize data alignment, that is, both parties can sort their private data according to these universal identifiers. They can then proceed item-by-item, doing any desired private computation. As a result, we can easily construct the above different PSO protocols from PID.

However, the existing PID protocols [5, 12] require that the inputs of both parties are a set, that is, the elements cannot be duplicated. The reason is that the way they generated UID only supports distinct elements. This requirement restricts the existing PID protocols from being applied to perform a wide range of analytical tasks with joins. In most analytical workloads, such as the decision support benchmark TPC-DS [34], the majority of joins are *key-foreign key joins* which correspond to one-to-many relationship between records from the two tables. In such joins, one party or table's *primary key* (an attribute with unique values in different records) needs to match the other's *foreign key* (with possibly duplicated values).

Recall the above example that Alice owns one table A (user_id, user_name, age, sex) and Bob owns the other table B (user_id, prod_id, prod_name, price). Here, user_id is the primary key of table A as each user's profile corresponds to exactly one record; user_id in table B is a foreign key and may have duplicated values as one user can buy multiple products. The two parties want to privately compute the average price in transactions from users with ages older than 30:

$$\begin{gathered}\textsf{SELECT AVG(B.price) FROM A INNER JOIN B}\\ \textsf{ON A.user\_id} = \textsf{B.user\_id WHERE A.age} > 30\end{gathered}$$

Note that such joint and private analytical tasks cannot be supported by the existing PID protocols [5, 12], because the identifiers on both sides need to be unique. There are some works to support these private queries [2, 3, 33]. However, all these works are implemented on the circuit using the general MPC technique, e.g. Yao's garbled circuit [37], GMW [15] etc, which makes it very inefficient.

For the general many-to-many relationship, the matched values will generate the Cartesian product of the two datasets (also known as *cross join*)[1]. All the above applications require generating multiple UIDs for duplicated values. We have the following questions:

Can we construct an efficient PID protocol in which the inputs of the parties are multiset?

1.2 Our Contribution

In this paper, we answer this question affirmatively in the semi-honest setting. Our contribution can be summarized as follows:

– **Efficient PMID constructions.** We introduce the notion of private multiset ID (PMID) protocol which supports the input of both parties to be multiset. We propose two PMID constructions: the first one is based on sloppy OPRF [12], which has a faster running-time; the second one is based on multi-point OPRF [6], which has a lower communication. The two constructions could be viewed as a trade-off between computation and communication.
– **Deterministic-Value Programmable PRF.** To construct efficient PMIDs, we propose a new variant of Programmable PRF [25] called Deterministic-Value Programmable PRF (dv-PPRF), and its corresponding protocol called Deterministic-Value Oblivious PPRF (dv-OPPRF). The deterministic-value property helps programming (probably duplicate) multiplicity for each element in the multiset(s). With the help of dv-OPPRF, we obtain desired PMIDs by extending the PIDs based on Oblivious PRF (OPRF) and PSU [12].

[1] In real scenarios, most join operations are one-to-many relationship, and the many-to-many relationship is usually considered to be avoided due to excessive data expansion. For completeness, we also consider such a general case in this paper.

– **Implementations.** We implement our PMID protocols and state-of-the-art PID protocols as performance baselines. The experiments show that our PMID performances are almost the same as their underlying PID counterparts [12] when we set the multiplicity $U_x = U_y = 1$. Our PMIDs scale well when either $U_x > 1$ or $U_y > 1$, and the performance results also reflect excessive data expansion when both $U_x, U_y > 1$ for the more general *cross join* case. Our implementations have been open-sourced and freely available under public requests.

1.3 Overview of Our Techniques

We provide the high-level technical overview for our PMID constructions. We assume that party Alice and Bob have multisets X and Y, respectively.

At first glance, one may consider using PIDs in a black-box manner to construct PMID protocols. A natural idea is to let both parties use their de-duplicated sets to execute the PID protocol, then let both parties tell each other the multiplicity of their elements, and finally both parties extend the UID of duplicated elements locally. Unfortunately, such construction introduces additional information leakage. Specifically, since Bob needs to tell Alice the multiplicity of *all* his elements, no matter whether they are in the intersection or not, Alice would know the multiplicity of elements that *even do not belongs to herself*, making this idea insecure.

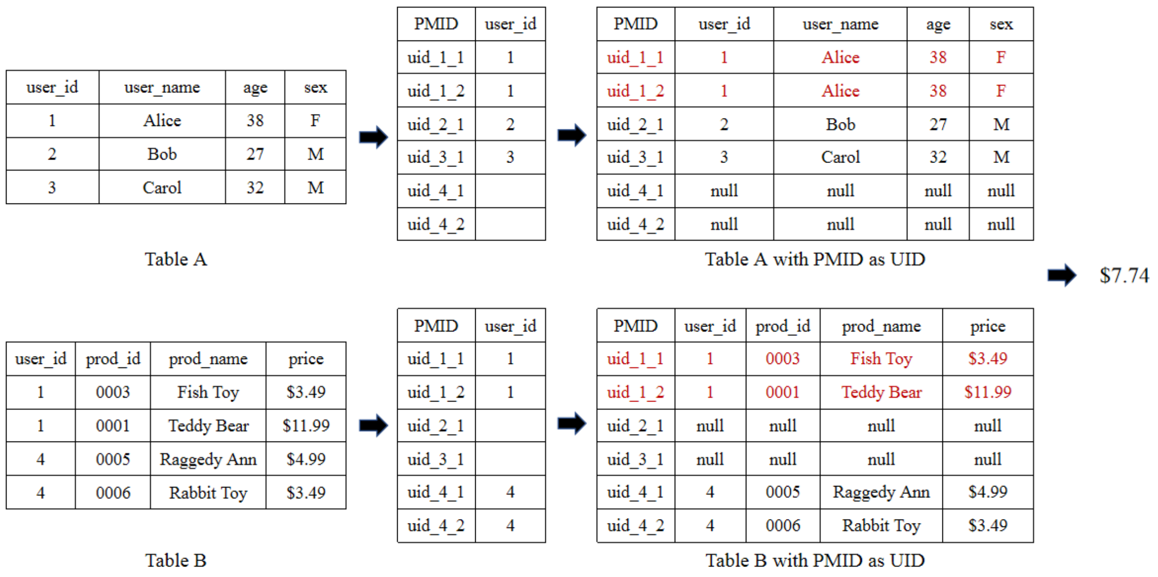
Our starting point is the PID protocols of [12]. Their main idea is as follows, the parties execute two Oblivious Pseudo-Random Function (OPRF) instances symmetrically. In the first instance, Alice learns k_A and Bob learns $F_{k_A}(y_i)$ for each of his items y_i; in the second instance, Bob learns k_B and Alice learns $F_{k_B}(x_i)$ for each of her items x_i. The UIDs is defined as $\mathsf{id}(x) := F_{k_A}(x) \oplus F_{k_B}(x)$. The parties compute the UIDs of the elements in their set and finally they execute a PSU protocol to obtain the whole UID set. For better efficiency, [12] further introduces a "sloppy OPRF" technique to generate UIDs. Roughly speaking, the sender inputs a set X and learns a key k, the receiver inputs a set Y and learns values $\{z_i\}_{i \in [n]}$. For every $y_i \in Y$, if $y_i \in X$, then $z_i = F_k(y_i)$, but such equality does not hold for other z_i. They use efficient batch single-point OPRF [24] to construct sloppy OPRF, see Section 4.1 for more details.

To generate multiple UIDs for duplicated elements, a natural idea is to lengthen the original UID with Pseudo-Random Generator (PRG) and take the output of PRG as the new UID. However, this method meets difficulties in security proof. In the original PID security proof [12], the UID is directly generated from the OPRF output. Since the simulator plays the role of OPRF functionality, the simulator can program the OPRF outputs directly so that the adversary can get the simulated UIDs that are consistent with the real ones. However, if we use PRG to generate the new UIDs, the simulator could only program the seed of PRG instead of PRG outputs, since PRG is one-way. As a result, the simulator cannot get the simulated UIDs output from PRG that are consistent with the ones from the real execution. To solve this problem, our idea is to use a programmable Random Oracle (RO) to institute the PRG, and the simulator could program the output of RO to real UIDs.

We also note that there is an important difference between PMID and PID: for an intersection element x, if its multiplicity in Alice's set is u_1 and its multiplicity in Bob's set is u_2, then x meets the general *cross join* case and both parties will obtain $u_1 u_2$ UIDs. If u_2 (resp. u_1) > 1, Alice (resp. Bob) will know that x is in the intersection and its multiplicity in Bob (resp. Alice)'s set. This leakage is implicit in the PMID definition[2] and cannot be avoided. To conclude, the security of PMID is guaranteed in two aspects: on the one hand, one party cannot distinguish the elements of its own set from the elements which the other party's multiplicity is 1 in the intersection; on the other hand, both parties cannot learn the multiplicity of elements outside their set.

To tell each other the multiplicity of intersection elements, our idea is to let both parties use an Oblivious Key-Value Store (OKVS) [13] to encode the multiplicity of their elements. However, if we use OKVS to encode multiplicity directly, both parties could use this OKVS to test any element's multiplicity, which makes the protocol insecure. Due to the above problem of OKVS, we consider using Oblivious Programmable PRF (OPPRF) [25] to program the multiplicity. The main difference between OKVS and OPPRF is that OPPRF actively enforces the receiver to evaluate the function on a limited number of queries, whereas OKVS is simply a data structure that is sent in the clear to the receiver, thus, no limit on the number of evaluation is set. The main idea of OPPRF is to

[2] The definition of our PMID naturally comes from the rules of join operation.



user_id	user_name	age	sex
1	Alice	38	F
2	Bob	27	M
3	Carol	32	M

Table A

PMID	user_id
uid_1_1	1
uid_1_2	1
uid_2_1	2
uid_3_1	3
uid_4_1	
uid_4_2	

PMID	user_id	user_name	age	sex
uid_1_1	1	Alice	38	F
uid_1_2	1	Alice	38	F
uid_2_1	2	Bob	27	M
uid_3_1	3	Carol	32	M
uid_4_1	null	null	null	null
uid_4_2	null	null	null	null

Table A with PMID as UID

$7.74

user_id	prod_id	prod_name	price
1	0003	Fish Toy	$3.49
1	0001	Teddy Bear	$11.99
4	0005	Raggedy Ann	$4.99
4	0006	Rabbit Toy	$3.49

Table B

PMID	user_id
uid_1_1	1
uid_1_2	1
uid_2_1	
uid_3_1	
uid_4_1	4
uid_4_2	4

PMID	user_id	prod_id	prod_name	price
uid_1_1	1	0003	Fish Toy	$3.49
uid_1_2	1	0001	Teddy Bear	$11.99
uid_2_1	null	null	null	null
uid_3_1	null	null	null	null
uid_4_1	4	0005	Raggedy Ann	$4.99
uid_4_2	4	0006	Rabbit Toy	$3.49

Table B with PMID as UID

Fig. 1: A Toy Demonstration of Computing the Example Query using PMID.

generate some one-time pads by OPRF and let the value encoded by the OKVS be multiplicity plus these one-time pads. However, we find the security of underlining programmable PRF (PPRF) is overkill for our construction because the security requires the values programmed by PPRF are all randomly selected. In our construction, these values are multiplicity of both parties' input, which is deterministic. As a result, we propose a weaker variant of PPRF, which we called deterministic-value programmable PRF (dv-PPRF). Roughly, the dv-PPRF program some deterministic values in PPRF and the adversary will learn these values, what we need is the queries outside of these deterministic values are pseudorandom. Furthermore, we room in the construction of dv-PPRF, and we find this new notion comes from a new property of OKVS, which we called partial obliviousness, see section 2.6 for details. After defining dv-PPRF, we naturally extend this primitive to the protocol called deterministic-value oblivious programmable PRF (dv-OPPRF) as [25]. See section 3 for more technical details. We note that we take the first step to explore the possibility that using PPRF to program non-random values, since the security property [25] requires the values should be randomly selected.

For those single elements in the multiset, i.e. the multiplicity is 1, if we program 1 directly in OPPRF, the parties can distinguish these elements from their own set elements because OPPRF will output a random number in their elements by the randomness of underlining PRF. Thus we also let both parties program random values for those single elements, resulting in they cannot distinguish them.

Putting all the pieces together, we can build PMID protocol from OPRF, dv-OPPRF, and PSU functionality in a modular way. (See Section 4 for the technical details). With the help of PMID, we can compute the example query shown in Section 1.1 as follows. First, both parties run PMID to compute the set of UID corresponding to the records in the union of Alice's set and Bob's *multisets*, where each party learns which UIDs correspond to which items. Then, both parties extend their dataset to have an UID column and sort the dataset by UID. In this way, datasets from both parties are aligned using UID without leaking the intersection. The attributes for UIDs that do not match any records are set as null. Finally, two parties run the desired computation under any general MPC protocol to obtain the query result. See Figure 1 as a toy demonstration.

2 Preliminaries

2.1 Notation

We use κ and λ to denote the computational and statistical security parameters, respectively. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$ and $[m, n]$ to denote the set $\{m, m+1, \ldots, n\}$. We use a set of key-value pairs to represent multiset, e.g. $Y = \{(y_1, u_1), (y_2, u_2)\}$ denotes a multiset in which the multiplicity of element y_1 is u_1 and the multiplicity of element y_2 is u_2. For a bit string v we let v_i denote the i-th bit. We use the abbreviation PPT to denote probabilistic polynomial-time. We denote $a \overset{R}{\leftarrow} A$ that a is randomly selected from the set A, and $a \leftarrow \mathsf{A}(x)$ that a is the output of the randomized algorithm A on input x, and $a := b$ that a is assigned by b.

2.2 Security Model

This work operates in the *semi-honest model*, where adversaries may try to learn as much information as possible from a given protocol execution but are not able to deviate from the protocol steps. We use the standard security definition for two-party computation [14] in this work.

Definition 1. *Let* $\mathsf{view}_A^{\Pi}(X, Y)$ *and* $\mathsf{view}_B^{\Pi}(X, Y)$ *be the views of Alice and Bob in the protocol, and let* $\mathsf{output}(X, Y)$ *be the output of both parties in the protocol. A protocol* Π *is said to securely compute functionality* f *in the semi-honest model if for every PPT adversary* $\mathcal{A}$ *there exists a PPT simulator* $\mathsf{Sim}_{\mathcal{S}}$ *and* $\mathsf{Sim}_{\mathcal{R}}$ *such that for all inputs* X *and* Y*,*

$$\{\mathsf{view}_A^{\Pi}(X, Y), \mathsf{output}(X, Y)\} \approx_c \{\mathsf{Sim}_A(X, f(X, Y)), f(X, Y)\}$$

$$\{\mathsf{view}_B^{\Pi}(X, Y), \mathsf{output}(X, Y)\} \approx_c \{\mathsf{Sim}_B(Y, f(X, Y)), f(X, Y)\}$$

2.3 Oblivious Transfer

Oblivious Transfer (OT) [35] is an important cryptographic primitive used in various multiparty computation protocols. We define the generalized primitive of 1-out-of-2 OT in Figure 2.

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, message length κ.
Functionality:

- Wait for input $b \in \{0, 1\}$ from the receiver $\mathcal{R}$.
- Wait for input (x_0, x_1) from the sender $\mathcal{S}$.
- Give x_b to the receiver $\mathcal{R}$.

Fig. 2: 1-out-of-2 Oblivious Transfer Functionality $\mathcal{F}_{\mathsf{ot}}$

2.4 Oblivious PRF

An OPRF [9] allows the receiver to input x and learn $F_k(x)$, where F is a PRF, and k is known to the sender. In this work, we use two variant of OPRF, namely, batch single-point OPRF [24] and multi-point OPRF [29, 6, 36].

In the batch single-point OPRF, the sender learns a set of PRF keys $\{k_i\}_{i\in[n]}$[3] and the receiver learns PRF values $\{F_{k_i}(x_i)\}_{i\in[n]}$ on its inputs $\{x_i\}_{i\in[n]}$. Note that the receiver learns the output of the PRF on only one value per key, and the sender does not learn which output the receiver learned. In the multi-point OPRF, the sender learns a PRF key k and the receiver learns PRF values $\{F_k(x_i)\}_{i\in[n]}$ on its inputs $\{x_i\}_{i\in[n]}$. The ideal functionality for batch single-point OPRF and multi-point OPRF are shown in Figure 3 and Figure 4.

[3] In fact, the protocol in [24] realizes OPRF instances where the keys k_i are related in some sense. However, the PRF that it instantiates has all the expected security properties, even in the presence of such related keys. For the sake of simplicity, we ignore this issue in our notation. See [24] for more details.

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, a PRF F, set sizes n
Functionality:

- Wait for input $\{x_1, \ldots, x_n\}$ from the receiver $\mathcal{R}$.
- Sample n random PRF keys $\{k_1, \ldots, k_n\}$ and give them to the sender $\mathcal{S}$.
- Give $\{F_{k_1}(x_1), \ldots, F_{k_n}(x_n)\}$ to the receiver $\mathcal{R}$.

Fig. 3: Batch Single-Point OPRF Functionality $\mathcal{F}_{\mathsf{bsp\text{-}oprf}}$

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, a PRF F, set sizes n
Functionality:

- Wait for input $\{x_1, \ldots, x_n\}$ from the receiver $\mathcal{R}$.
- Sample a random PRF key k and give it to the sender $\mathcal{S}$.
- Give $\{F_k(x_1), \ldots, F_k(x_n)\}$ to the receiver $\mathcal{R}$.

Fig. 4: Multi-Point OPRF Functionality $\mathcal{F}_{\mathsf{mp\text{-}oprf}}$

2.5 Cuckoo Hashing

Cuckoo hashing was introduced by Pagh and Rodler in [28]. In this hashing scheme, there are α hash functions $h_1, \ldots, h_\alpha$ used to map n items into $\rho = \epsilon n$ bins and a stash, and we denote the i-th bin as $\mathcal{B}_i$. The Cuckoo hashing can guarantee that there is only one item in each bin, and the approach to avoid collisions is as follows: An element x is inserted into a bin $\mathcal{B}_{h_1(x)}$. Any prior contents z of $\mathcal{B}_{h_1(x)}$ are evicted to a new bin $\mathcal{B}_{h_i(z)}$, using h_i to determine the new bin location, where $h_i(z) \neq h_1(x)$ for $i \in [\alpha]$. The procedure is repeated until no more evictions are necessary, or until a threshold number of relocations has been performed. In the latter case, the last element is put in a special stash. According to the empirical analysis in [32], we can adjust the values of α and ϵ to reduce the stash size to 0 while achieving a hashing failure probability of $2^{-\lambda}$.

We use the notation $\mathcal{B} \leftarrow \mathsf{Cuckoo}^{\rho}_{h_1,\ldots,h_\alpha}(X)$ to denote hashing the items of X into ρ bins using Cuckoo hashing on hash functions $h_1, \ldots, h_\alpha : \{0,1\}^* \to [\rho]$. Some positions of $\mathcal{B}$ will not matter, corresponding to empty bins.

2.6 Oblivious Key-Value Store

A key-value store [30, 13] is simply a data structure that maps a set of keys to corresponding values. The definition is as follows:

Definition 2 (Key-Value Store). *A key-value store is parameterized by a set $\mathcal{K}$ of keys, a set $\mathcal{V}$ of values, and a set of function H, and consists of two algorithms:*

- $\mathsf{Encode}_H(\{(x_1, y_1), \ldots, (x_n, y_n)\})$[4]*: on input key-value pairs $\{(x_i, y_i)\}_{i\in[n]} \subseteq \mathcal{K} \times \mathcal{V}$, outputs an object D (or, with statistically small probability, an error indicator $\perp$).*
- $\mathsf{Decode}_H(D, x)$ *: on input D and a key x, outputs a value $y \in \mathcal{V}$.*

Correctness. For all $A \subseteq \mathcal{K} \times \mathcal{V}$ with distinct keys:

$$(x, y) \in A \text{ and } \perp \neq D \leftarrow \mathsf{Encode}_H(A) \Longrightarrow \mathsf{Decode}_H(D, x) = y$$

Obliviousness. For all distinct $\{x_1^0, \ldots, x_n^0\}$ and $\{x_1^1, \ldots, x_n^1\}$, if Encode_H does not output $\perp$ for $\{x_1^0, \ldots, x_n^0\}$ or $\{x_1^1, \ldots, x_n^1\}$, the distribution of $\{D | y_i \leftarrow \mathcal{V}, i \in [n], \mathsf{Encode}_H((x_1^0, y_1), \ldots, (x_n^0, y_n))\}$ is computationally indistinguishable to the distribution of $\{D | y_i \leftarrow \mathcal{V}, i \in [n], \mathsf{Encode}_H((x_1^1, y_1), \ldots, (x_n^1, y_n))\}$.

A key-value store is an oblivious key-value store (OKVS) if it satisfies the obliviousness property.

Intuitively, obliviousness means that when value is randomly selected, the distribution of D is independent from key's set. In our application, we instead require OKVS to satisfy the following *partial obliviousness* property since our application will always leak some values.

[4] We sometimes use $\mathsf{Encode}_H(X, Y)$ for $\mathsf{Encode}_H(\{(x_1, y_1), \ldots, (x_n, y_n)\})$ for convenience, where $X = \{x_1, \ldots, x_n\}, Y = \{y_1, \ldots, y_n\}$.

Partial Obliviousness. For $t \in [n]$, and some fixed key-value pairs $\{(x_i, y_i)\}_{i \in [t]}$, for all distinct $\{x^0_{t+1}, \ldots, x^0_n\}$ and all distinct $\{x^1_{t+1}, \ldots, x^1_n\}$, if Encode_H does not output $\perp$, then the following distributions are computationally indistinguishable:

$$\{D | y_i \xleftarrow{\text{R}} \mathcal{V}, i \in [t+1, n], \mathsf{Encode}_H((x_1, y_1), \ldots, (x_t, y_t), (x^0_{t+1}, y_{t+1}), \ldots, (x^0_n, y_n))\}$$
$$\{D | y_i \xleftarrow{\text{R}} \mathcal{V}, i \in [t+1, n], \mathsf{Encode}_H((x_1, y_1), \ldots, (x_t, y_t), (x^1_{t+1}, y_{t+1}), \ldots, (x^1_n, y_n))\}$$

We note that when $t = 0$, this property is equal to the standard Obliviousness, and when $t = n$, the two distributions are identical.

Proof of Partial Obliviousness. Common OKVS candidates include polynomial, Garbled Bloom Filter (GBF) [8] and Garbled Cuckoo Table (GCT) [30, 36, 13] etc, which are linear OKVS schemes. We give the definition of linear OKVS and prove it to satisfy the partial obliviousness in Appendix A.

2.7 Private Set Union

PSU is a special case of secure two-party computation. The ideal functionality for PSU is given in Figure 5.

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, set sizes n_{y} and n_{x}.
Functionality:

- Wait for input $X = \{x_1, \ldots, x_{n_{\mathsf{x}}}\}$ from the receiver $\mathcal{R}$.
- Wait for input $Y = \{y_1, \ldots, y_{n_{\mathsf{y}}}\}$ from the sender $\mathcal{S}$.
- Give output $X \cup Y$ to the receiver $\mathcal{R}$.

Fig. 5: Private Set Union Functionality $\mathcal{F}_{\mathsf{psu}}$

3 Deterministic-Value (Oblivious) Programmable PRF

We review the concepts of Programmable PRF (PPRF) [25] here and also introduce our novel deterministic-value variant of a PPRF.

3.1 Definitions

Programmable PRF (PPRF) [25] is a special PRF with the additional property that on a certain "programmed" set of inputs the function outputs "programmed" values. A programmable PRF consists of the following algorithms:

- $\mathsf{KeyGen}(1^\kappa, \mathcal{P}) \to (k, \mathsf{hint})$: Given a security parameter and set of points $\mathcal{P} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ with distinct x_i-values, generates a PRF key k and (public) auxiliary information hint.
- $F(k, \mathsf{hint}, x) \to y$: Evaluates the PRF on input x, giving output y.

Correctness. A programmable PRF satisfies correctness if $(x, y) \in \mathcal{P}$, and $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(1^\kappa, \mathcal{P})$, then $F(k, \mathsf{hint}, x) = y$.
Security. For security, considering the following experiment:

```
$\mathsf{Exp}^{\mathcal{A}}(X, Q, \kappa)$:
for each $x_i \in X$, choose random $y_i \leftarrow \mathcal{V}$
$(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(1^\kappa, \{(x_i, y_i) | x_i \in X\})$
return $\mathcal{A}(\mathsf{hint}, \{F(k, \mathsf{hint}, q) | q \in Q\})$
```

We say that a PPRF is (n, μ)-secure if for all $|X_0| = |X_1| = n$, all $|Q| = \mu$, and all PPT $\mathcal{A}$:

$$|Pr[\mathsf{Exp}^{\mathcal{A}}(X_0, Q, \kappa) = 1] - Pr[\mathsf{Exp}^{\mathcal{A}}(X_1, Q, \kappa) = 1]| \leq negl(\kappa)$$

The security requires that it is hard to tell what the set of programmed points was, given the hint and μ outputs of the PRF, if the points were programmed to *random* outputs. This implies that unprogrammed PRF outputs (i.e., those not set by the input to KeyGen) are pseudorandom.

However, we find that the above security property is too strong to be used in our construction. What we want is to use the PPRF to "program" the multiplicity of the sender's elements and let the receiver evaluate the function on his own set elements. The multiplicity is uniquely determined by the input set, instead of randomly selected as in the security definition. In fact, the multiplicity of some intersection elements must be leaked to the adversary. Fortunately, we find the security property of PPRF is overkill and the following *deterministic-value pseudorandomness* is enough:

Deterministic-Value Pseudorandomness. For any fixed set of points $\mathcal{P} = \{(x_1, y_1), \ldots, (x_t, y_t)\}$, considering the following experiment:

```
$\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}, X, Q, \kappa)$:
for each $x_i \in X$, choose random $y_i \leftarrow \mathcal{V}$
$(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(1^\kappa, \mathcal{P} \cup \{(x_i, y_i) | x_i \in X\})$
return $\mathcal{A}(\mathcal{P}, \mathsf{hint}, \{F(k, \mathsf{hint}, q) | q \in Q\})$
```

We say that a PPRF satisfying (t, n, μ)-deterministic-value pseudorandomness if for all $|X_0| = |X_1| = n - t$, all $|Q| = \mu$ satisfying $Q \cap \{x_1, \ldots, x_t\} = \emptyset$ and all PPT $\mathcal{A}$:

$$|Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}, X_0, Q, \kappa) = 1] - Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}, X_1, Q, \kappa) = 1]| \leq negl(\kappa)$$

In the above definition, some points, i.e. $\mathcal{P}$, are definitely leaked to the adversary. However, what we need is only the pseudorandomness of PPRF values outside of the leaked set.

Definition 3 (dv-PPRF). *A Deterministic-Value Programmable PRF (dv-PPRF) is the PPRF scheme satisfying correctness and (t, n, μ)-deterministic-value pseudorandomness.*

After defining the dv-PPRF, it is natural to define the functionality of Deterministic-Value Oblivious Programmable PRF (dv-OPPRF) like [25]. We give the formal description of dv-OPPRF functionality in Figure 6. The only difference between standard OPPRF [25] and dv-OPPRF is that the underlying PPRF is replaced by dv-PPRF.

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, a dv-PPRF scheme (KeyGen, F), and upper bound n on the number of points to be programmed, and bound μ on the number of queries.
Functionality:

- Wait for input $\mathcal{P} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ from the sender $\mathcal{S}$.
- Wait for input $\{q_1, \ldots, q_\mu\}$ from the receiver $\mathcal{R}$.
- Run $(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(1^\kappa, \mathcal{P})$ and give (k, hint) to the sender $\mathcal{S}$.
- Give $\{\mathsf{hint}, F(k, \mathsf{hint}, q_1), \ldots, F(k, \mathsf{hint}, q_\mu)\}$ to the receiver $\mathcal{R}$.

Fig. 6: Deterministic-Value Oblivious Programmable PRF $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$

3.2 Construction of dv-PPRF

To construct dv-PPRF, the main idea is to combine the PRF and the OKVS with partial obliviousness property. Let $\widehat{F}$ be a PRF and $(\mathsf{Encode}_H, \mathsf{Decode}_H)$ be an OKVS scheme satisfying partial obliviousness. We define it as follows:

- $\mathsf{KeyGen}(1^\kappa, \{(x_1, y_1), \ldots, (x_n, y_n)\})$: Choose a random key k for $\widehat{F}$. Compute an OKVS $D := \mathsf{Encode}_H((x_1, y_1 \oplus \widehat{F}_k(x_1)), \ldots, (x_n, y_n \oplus \widehat{F}_k(x_n)))$. Let hint be D.

– $F(k, \mathsf{hint}, q) = \widehat{F}_k(q) \oplus \mathsf{Decode}_H(\mathsf{hint}, q)$.

Theorem 1. *Assuming the OKVS scheme satisfies partial obliviousness, the above construction is a dv-PPRF.*

Proof. If there is an adversary $\mathcal{A}$ can break the deterministic-value pseudorandomness of dv-PPRF, then we can construct a PPT distinguisher $\mathcal{D}$ to distinguish the two distributions of partial obliviousness in OKVS with non-negligible probability. Let $\mathcal{P} = \{(x_1, y_1), \dots, (x_t, y_t)\}$, $X_0 = \{x^0_{t+1}, \dots, x^0_n\}$ and $X_1 = \{x^1_{t+1}, \dots, x^1_n\}$.

$\mathcal{D}$ works as follows: after receiving D, the distinguisher $\mathcal{D}$ selects a random PRF key k from the key space $\mathcal{K}$ of PRF $\widehat{F}$. Then, $\mathcal{D}$ defines $\mathsf{hint} := D$ and let $F(k, \mathsf{hint}, q) := \widehat{F}_k(q) \oplus \mathsf{Decode}_H(\mathsf{hint}, q)$ for any query $q \in Q$. The distinguisher $\mathcal{D}$ invokes $\mathcal{A}$ with input $(\mathcal{P}, \mathsf{hint}, \{F(k, \mathsf{hint}, q) | q \in Q\})$ and outputs $\mathcal{A}$'s output. For simplicity, we use $(\mathcal{P}, (X_b, Y))$ to denote $((x_1, y_1), \dots, (x_t, y_t), (x^b_{t+1}, y_{t+1}), \dots, (x^b_n, y_n))$ for $b \in \{0, 1\}$, where $y_i \xleftarrow{\text{R}} \mathcal{V}, i \in [t+1, n]$. We have:

$$\begin{aligned}
&|Pr[\mathcal{D}(D)|D = \mathsf{Encode}_H(\mathcal{P}, (X_0, Y))] - Pr[\mathcal{D}(D)|D = \mathsf{Encode}_H(\mathcal{P}, (X_1, Y))]| \\
=&|Pr[\mathcal{A}(\mathcal{P}, \mathsf{hint}, \{F(k, \mathsf{hint}, q)\}_{q \in Q})|k \xleftarrow{\text{R}} \mathcal{K}, \mathsf{hint} = \mathsf{Encode}_H(\mathcal{P}, (X_0, Y))] - \\
&Pr[\mathcal{A}(\mathcal{P}, \mathsf{hint}, \{F(k, \mathsf{hint}, q)\}_{q \in Q})|k \xleftarrow{\text{R}} \mathcal{K}, \mathsf{hint} = \mathsf{Encode}_H(\mathcal{P}, (X_1, Y))]| \\
=&|Pr[\mathcal{A}(\mathcal{P}, \mathsf{hint}, \{F(k, \mathsf{hint}, q)\}_{q \in Q})|(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\mathcal{P}, (X_0, Y))] - \\
&Pr[\mathcal{A}(\mathcal{P}, \mathsf{hint}, \{F(k, \mathsf{hint}, q)\}_{q \in Q})|(k, \mathsf{hint}) \leftarrow \mathsf{KeyGen}(\mathcal{P}, (X_1, Y))]| \\
=&|Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}, X_0, Q, \kappa) = 1] - Pr[\mathsf{Exp}^{\mathcal{A}}(\mathcal{P}, X_1, Q, \kappa) = 1]|
\end{aligned}$$

Thus $\mathcal{D}$ breaks partial obliviousness of OKVS with the same advantages as $\mathcal{A}$.

Now we are ready to give the construction of dv-OPPRF protocol, the formal description is in Figure 7. Simulation is trivial, as the parties' views in the protocol are exactly the dv-OPPRF output.

Parameters:

– Two parties: Sender $\mathcal{S}$ and Receiver $\mathcal{R}$.
– Ideal $\mathcal{F}_{\mathsf{mp\text{-}oprf}}$ primitives specified in Figure 4.
– An OKVS scheme $(\mathsf{Encode}_H, \mathsf{Decode}_H)$.

Input of Sender: $\mathcal{P} = \{(x_1, y_1), \dots, (x_n, y_n)\}$
Input of Receiver: $Q = \{q_1, \dots, q_\mu\}$
Protocol:

1. The parties invoke multi-point OPRF functionality $\mathcal{F}_{\mathsf{mp\text{-}oprf}}$, $\mathcal{S}$ acts as the sender and receives a random PRF key k, $\mathcal{R}$ acts as the receiver with input Q and receives $\{\widehat{F}_k(q) | q \in Q\}$.
2. $\mathcal{S}$ computes an OKVS $D := \mathsf{Encode}_H((x_1, y_1 \oplus \widehat{F}_k(x_1)), \dots, (x_n, y_n \oplus \widehat{F}_k(x_n)))$ and defines $\mathsf{hint} := D$.
3. S sends hint to the receiver $\mathcal{R}$.
4. $\mathcal{R}$ outputs $(\mathsf{hint}, \{\mathsf{Decode}_H(\mathsf{hint}, q) \oplus \widehat{F}_k(q) | q \in Q\})$.

Fig. 7: General construction of dv-OPPRF Protocol $\Pi_{\mathsf{dv\text{-}OPPRF}}$

4 Private Multiset ID

We give the formal definition of Private Multiset ID (PMID) functionality $\mathcal{F}_{\mathsf{PMID}}$ in Figure 8. As we mentioned in Section 1.1, the one-to-many relationship is the main application scenario of PMID. Therefore, we consider two versions of PMID: in the one-sided version, the input of Alice is a set while the input of Bob is a multiset; in the general version, the input of both parties is multiset. We also explicitly define the leaked set from in the functionality, though it could be inferred from the PMID outputs. We highlight the redundant part, i.e. X_{leak}, in the definition of one-sided version, which could be easily verified $X_{\mathsf{leak}} = \emptyset$ when $u^{\mathsf{x}}_1 = \cdots = u^{\mathsf{x}}_m = 1$.

Parameters: Two parties Alice and Bob. Number of items m, n for the Alice and Bob; length of identifiers l; the upper bound of duplicate item in X and Y, U_x and U_y; the ID mapping $\mathsf{id} : \{0,1\}^* \to \{0,1\}^l$.
Functionality:

- Wait for input $X = \{(x_1, u_1^{\mathsf{x}}), \ldots, (x_m, u_m^{\mathsf{x}})\} \subset \{0,1\}^* \times [U_x]$ from Alice. In the one-sided version, $u_1^{\mathsf{x}} = \cdots = u_m^{\mathsf{x}} = 1$, i.e. the input of Alice is a set.
- Wait for input $Y = \{(y_1, u_1^{\mathsf{y}}), \ldots, (y_n, u_n^{\mathsf{y}})\} \subset \{0,1\}^* \times [U_y]$ from Bob.
- Let $X' := \{x_1 \ldots, x_m\}$ and $Y' := \{y_1 \ldots, y_n\}$ be the sets without duplication items corresponding to X and Y.
- For every $x_i \in X' \setminus Y'$, choose u_i^{x} random identifier $\mathsf{id}(x_i^{(t)}) \in \{0,1\}^l, t \in [u_i^{\mathsf{x}}]$; for every $y_i \in Y' \setminus X'$, choose u_i^{y} random identifier $\mathsf{id}(y_i^{(t)}) \in \{0,1\}^l, t \in [u_i^{\mathsf{y}}]$; for every $z_i \in X' \cap Y'$, assuming $(z_i, u_i^{\mathsf{x}}) \in X, (z_i, u_i^{\mathsf{y}}) \in Y$, choose $u_i^{\mathsf{x}} u_i^{\mathsf{y}}$ random identifier $\mathsf{id}(z_i^{(t)}) \in \{0,1\}^l, t \in [u_i^{\mathsf{x}} u_i^{\mathsf{y}}]$.
- Define $R^* := \{\mathsf{id}(x_i^{(t)}) | x_i \in X' \setminus Y', t \in [u_i^{\mathsf{x}}]\} \cup \{\mathsf{id}(y_i^{(t)}) | y_i \in Y' \setminus X', t \in [u_i^{\mathsf{y}}]\} \cup \{\mathsf{id}(z_i^{(t)}) | z_i \in X' \cap Y', t \in [u_i^{\mathsf{x}} u_i^{\mathsf{y}}]\}$.
- Define $ID_X := \{\mathsf{id}(x_i^{(t)}) | x_i \in X' \setminus Y', t \in [u_i^{\mathsf{x}}]\} \cup \{\mathsf{id}(z_i^{(t)}) | z_i \in X' \cap Y', t \in [u_i^{\mathsf{x}} u_i^{\mathsf{y}}]\}$. Define $ID_Y := \{\mathsf{id}(y_i^{(t)}) | y_i \in Y' \setminus X', t \in [u_i^{\mathsf{y}}]\} \cup \{\mathsf{id}(z_i^{(t)}) | z_i \in X' \cap Y', t \in [u_i^{\mathsf{x}} u_i^{\mathsf{y}}]\}$.
- Define $X_{\mathsf{leak}} := \{(x_i, u_i^{\mathsf{x}}) | (x_i, u_i^{\mathsf{x}}) \in X, u_i^{\mathsf{x}} > 1\}$ and $Y_{\mathsf{leak}} := \{(y_i, u_i^{\mathsf{y}}) | (y_i, u_i^{\mathsf{y}}) \in Y, u_i^{\mathsf{y}} > 1\}$.
- Give output $(R^*, ID_X, Y_{\mathsf{leak}})$[a] to Alice and give output $(R^*, ID_Y, X_{\mathsf{leak}})$ to Bob.

[a] We note that the ID_X also includes the mapping relationship between x_i and $\mathsf{id}(x_i^{(t)}), t \in [u_i^{\mathsf{x}}]$ (similarly for y's) while the R^* does not contain this relationship.

Fig. 8: Private Multiset ID Functionality $\mathcal{F}_{\mathsf{PMID}}$. The highlighted parts could omit for one-sided version.

4.1 PMID from Sloppy OPRF

Now we describe our PMID protocol. As we mentioned in Section 1.3, the parties run sloppy OPRF twice to generate the UIDs of the de-duplicated set, then both parties program the multiplicity of their elements by dv-OPPRF, the elements with multiplicity 1 are programmed by a random value. After execution of dv-OPPRF, the parties compute the multiplicity of all their elements and query random oracle to obtain the UIDs. Finally, the parties run a PSU protocol to obtain the whole UIDs' set. Now, we give our PMID protocol in Figure 9. The redundant parts for one-sided version are highlighted.

Correctness. For $x_i \in X' \cap Y'$, suppose x_i is placed to bin $h_v(x_i)$ by Alice, then Alice computes $r^A(x_i) = \mathsf{Decode}_H(P^B, x_i || v) \oplus f^A_{h_v(x_i)} \oplus F_{s^A}(x_i)$. Since $x_i \in Y'$, the OKVS P^B satisfies $\mathsf{Decode}_H(P^B, x_i || v) = F_{s^B}(x_i) \oplus F_{k^B_{h_v(x_i)}}(x_i || v)$. Thus we have that $r^A(x_i) = F_{s^B}(x_i) \oplus F_{s^A}(x_i)$. Similarly, for $y_j \in X \cap Y'$, we also have $r^B(y_j) = F_{s^A}(y_j) \oplus F_{s^B}(y_j)$, which means $r^A(x_i) = r^B(y_j)$ for $x_i = y_j \in X' \cap Y'$. In the case of $u_j^{\mathsf{y}} > 1$, we have $d_i^B = F(k_B, \mathsf{hint}_B, x_i) = F(k_B, \mathsf{hint}_B, y_j) = c_j^{\mathsf{y}} = u_j^{\mathsf{y}}$. Thus $\bar{u}_i^{\mathsf{x}} = u_i^{\mathsf{x}} \cdot d_i^B = u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}$, and $\mathsf{id}(x_i^{(t)}) = \mathsf{id}(y_j^{(t)})$ for $t \in [u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}]$. In the case of $u_j^{\mathsf{y}} = 1$, we have $d_i^B = F(k_B, \mathsf{hint}_B, x_i) = F(k_B, \mathsf{hint}_B, y_j) = c_j^{\mathsf{y}}$. Since c_j is randomly picked from $\{0,1\}^\sigma$, by setting $\sigma = \lambda + \log nU_y$, a union bound shows probability of $c_j \le U_y$ is negligible $2^{-\lambda}$. Thus $\bar{u}_i^{\mathsf{x}} = u_i^{\mathsf{x}} \cdot 1 = u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}$ with overwhelming probability and $\mathsf{id}(x_i^{(t)}) = \mathsf{id}(y_j^{(t)})$ for $t \in [u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}]$. If $x_i \in X' \setminus Y'$, by the deterministic-value pseudorandomness of dv-OPPRF, d_i^B is indistinguishable from random distribution over $\{0,1\}^\sigma$. By setting $\sigma = \lambda + \log mU_x$ [5], the union bound guarantees $d_i^B > U_x$ with overwhelming probability, which infers $\bar{u^{\mathsf{x}}}_i = u_i^{\mathsf{x}}$ with overwhelming probability.

Theorem 2. *The protocol in Figure 9 securely computes $\mathcal{F}_{\mathsf{PMID}}$ against semi-honest adversaries in the $(\mathcal{F}_{\mathsf{bsp\text{-}oprf}}, \mathcal{F}_{\mathsf{psu}})$-hybrid model.*

Proof. Since the protocol is symmetric, we only exhibit the simulator for simulating corrupt Alice.

For the one-sided version, we note that the messages that need to be simulated are strictly less than the general version (there is no need to invoke the functionality $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$ in step 12), which means the simulator of the general version infers the simulator of one-sided version.

[5] Thus we set $\sigma = \max\{\lambda + \log nU_y, \lambda + \log mU_x\}$.

Corrupt Alice: The simulator for corrupt Alice receives ideal output $(R^*, ID_X, Y_{\mathsf{leak}})$. The simulator simulates Alice's view as follows:

1. In step 1, the simulator executes honestly to obtain $\mathcal{A}$.
2. In step 2, the simulator selects uniform random $\{f_u^A\}_{u\in[\rho_1]}$, then invokes OPRF simulator with input $(\mathcal{A}, \{f_u^A\}_{u\in[\rho_1]})$ and appends the output to the view.
3. In step 3, the simulator selects a random OKVS P^B and appends it to the view. Here the simulator also computes $r^A(x)$ for $x \in X'$ as the honest Alice does.
4. In step 6, the simulator selects uniform random PRF keys $\{k_u^A\}_{u\in[\rho_2]}$, then it invokes OPRF simulator with input $\{k_u^A\}_{u\in[\rho_2]}$ and appends the output to the view. The simulator also selects a random key s^A for Alice.
5. In step 9, the simulator defines c_i^{x} as the honest Alice does.
6. Let $|Y_{\mathsf{leak}}| = \mu$. For $x_i \in Y_{\mathsf{leak}}$, let d_i^B denote the multiplicity of x_i. For $x_i \notin Y_{\mathsf{leak}}$, the simulator selects $d_i^B \xleftarrow{\text{R}} \{0,1\}^\sigma$ and computes $(k_B, \mathsf{hint}_B) \leftarrow \mathsf{KeyGen}(\{(x_i, d_i^B)\}_{i\in[m]})$. In step 10, The simulator invokes dv-OPPRF simulator with input $(X', \mathsf{hint}_B, \{d_i^B\}_{i\in[m]})$ and appends the output to the view.
7. In step 11, the simulator defines $\bar{u}_i^{\mathsf{x}}$ as the honest Alice does.
8. In step 12, the simulator computes the key of dv-PPRF as $(k_A, \mathsf{hint}_A) \leftarrow \mathsf{KeyGen}(\{(x_i, c_i^{\mathsf{x}})\}_{i\in[m]})$. Then the simulator invokes dv-OPPRF simulator with input $(\{(x_i, c_i^{\mathsf{x}})\}_{i\in[m]},\ k_A, \mathsf{hint}_A)$ and appends the output to the view.
9. In step 14, for $i \in [m], t \in [\bar{u}_i^{\mathsf{x}}]$: let $\mathsf{id}(x_i^{(t)})$ as the response of RO query $r^A(x_i)||t$.
10. In step 16, the simulator invokes PSU simulator with input ID_X and appends the output and R^* to the view.

We show the correctness of this simulation via a sequence of hybrids:

- Hybrid$_0$. The first hybrid is the real interaction described in Figure 9. Here, the honest Bob uses input Y, and honestly interacts with the corrupt Alice. Let T_0 denote the real view of Alice.
- Hybrid$_1$. Let T_1 be the same as T_0, except that all terms of the form $F_k(z)$ are replaced by random values. This hybrid is computationally indistinguishable from T_0 by the security of $\mathcal{F}_{\mathsf{bsp\text{-}oprf}}$ and the pseudorandomness of PRF.
- Hybrid$_2$. Let T_2 be the same as T_1, except that the OKVS P^B in step 3 is randomly selected. Since all $\{F_{s^B}(y)\}_{y\in Y'}$ are substituted by random values, by obliviousness of OKVS, T_1 and T_2 are statistically indistinguishable.
- Hybrid$_3$. Let T_3 be the same as T_2, except that the inputs of the Bob are replaced by random items except those with multiplicity greater than one in the intersection (i.e. the items in Y_{leak}). Note that for those $y_j \notin Y_{\mathsf{leak}}$, the value d_i^B is uniform and independent from Alice's view. By the deterministic-value pseudorandomness property of dv-PPRF, T_1 and T_2 are computationally indistinguishable.
- Hybrid$_4$. Let T_4 be the same as T_3, except that the PSU execution is replaced by underlining PSU simulator. The security of $\mathcal{F}_{\mathsf{psu}}$ functionality guarantees that T_4 and T_3 are computationally indistinguishable.
- Hybrid$_5$. Let T_5 be the same as T_4, except the way of computing $\mathsf{id}(x)$. Instead of computing $\mathsf{id}(x_i^{(t)})$ as in step 12, where $\bar{H}(r^A(x_i)||t)$ is a uniform random value, we instead compute $\mathsf{id}(x_i^{(t)})$ randomly and then program $\bar{H}(r^A(x_i)||t)$ to be the correct value, i.e., $\bar{H}(r^A(x_i)||t) := \mathsf{id}(x_i^{(t)})$. This change has no effect on Alice's view distribution. This hybrid is exactly the view output by the simulator.

4.2 PMID from Standard OPRF

Though sloppy OPRF-based PMID is usually more efficient, we find that the standard multi-point OPRF-based PMID has lower communication. For completeness, we also present the PMID protocol based on the standard multi-point OPRF here. The protocol is described in Figure 10. The redundant parts of the one-sided version are highlighted.

Correctness. For all $i \in [m]$, if $x_i \in Y'$, there is a $y_j \in Y', j \in [n]$ s.t. $y_j = x_i$. In the case of $u_j^{\mathsf{y}} > 1$, we have $d_i^B = F(k_B, \mathsf{hint}_B, x_i) = F(k_B, \mathsf{hint}_B, y_j) = c_j^{\mathsf{y}} = u_j^{\mathsf{y}}$. Thus $\bar{u}_i^{\mathsf{x}} = u_i^{\mathsf{x}} \cdot d_i^B = u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}$, and $\mathsf{id}(x_i^{(t)}) =$

$\mathsf{id}(y_j^{(t)})$ for $t \in [u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}]$. In the case of $u_j^{\mathsf{y}} = 1$, we have $d_i^B = F(k_B, \mathsf{hint}_B, x_i) = F(k_B, \mathsf{hint}_B, y_j) = c_j^{\mathsf{y}}$. Since c_j is randomly picked from $\{0,1\}^\sigma$, by setting $\sigma = \lambda + \log nU_y$, a union bound shows probability of $c_j \leq U_y$ is negligible $2^{-\lambda}$. Thus $\bar{u}_i^{\mathsf{x}} = u_i^{\mathsf{x}} \cdot 1 = u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}$ with overwhelming probability and $\mathsf{id}(x_i^{(t)}) = \mathsf{id}(y_j^{(t)})$ for $t \in [u_i^{\mathsf{x}} \cdot u_j^{\mathsf{y}}]$. If $x_i \in X' \setminus Y'$, by the deterministic-value pseudorandomness of dv-OPPRF, d_i^B is indistinguishable from random distribution over $\{0,1\}^\sigma$. By setting $\sigma = \lambda + \log mU_x$ [6], the union bound guarantees $d_i^B > U_x$ with overwhelming probability, which infers $\bar{u^{\mathsf{x}}}_i = u_i^{\mathsf{x}}$ with overwhelming probability.

We now state the security properties of our PMID.

Theorem 3. *The protocol in Figure 10 securely computes $\mathcal{F}_{\mathsf{PMID}}$ against semi-honest adversaries in the $(\mathcal{F}_{\mathsf{mp\text{-}oprf}}, \mathcal{F}_{\mathsf{psu}})$-hybrid model.*

Proof. Since the protocol is symmetric, we only exhibit the simulator for simulating corrupt Alice.

For the one-sided version, we note that the messages that need to be simulated are strictly less than the general version (there is no need to invoke the functionality $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$ in step 6), which means the simulator of the general version infers the simulator of one-sided version.

Corrupt Alice: The simulator for corrupt Alice receives ideal output $(R^*, ID_X, Y_{\mathsf{leak}})$. The simulator simulates Alice's view as follows:

1. In step 1, the simulator selects a random PRF key k_A, then invokes OPRF simulator with input k_A and appends the output to the view.
2. In step 2, the simulator selects random values for $F_{k_B}(x_i)$, then invokes OPRF simulator with input $(X', \{F_{k_B}(x) | x \in X'\})$ and appends the output to the view.
3. Let $|Y_{\mathsf{leak}}| = \mu$. For $x_i \in Y_{\mathsf{leak}}$, let d_i^B denote the multiplicity of x_i. For $x_i \notin Y_{\mathsf{leak}}$, the simulator selects $d_i^B \xleftarrow{\mathrm{R}} \{0,1\}^\sigma$ and computes $(k_B, \mathsf{hint}_B) \leftarrow \mathsf{KeyGen}(\{(x_i, d_i^B)\}_{i \in [m]})$. In step 4, The simulator invokes dv-OPPRF simulator with input $(X', \mathsf{hint}_B, \{d_i^B\}_{i \in [m]})$ and appends the output to the view.
4. In step 5, the simulator defines $\bar{u}_i^{\mathsf{x}}$ as the honest Alice does.
5. In step 6, the simulator computes the key of dv-PPRF as $(k_A, \mathsf{hint}_A) \leftarrow \mathsf{KeyGen}(\{(x_i, c_i^{\mathsf{x}})\}_{i \in [m]})$. Then the simulator invokes dv-OPPRF simulator with input $(\{(x_i, c_i^{\mathsf{x}})\}_{i \in [m]}, k_A, \mathsf{hint}_A)$ and appends the output to the view.
6. In step 8, for $i \in [m], t \in [\bar{u}_i^{\mathsf{x}}]$: let $\mathsf{id}(x_i^{(t)})$ as the response of RO query $F_{k_B}(x_i) \oplus F_{k_A}(x_i) || t$.
7. In step 9, the simulator invokes PSU simulator with input ID_X and appends the output and R^* to the view.

We show the correctness of this simulation via a sequence of hybrids:

- Hybrid$_0$. The first hybrid is the real interaction described in Figure 10. Here, the honest Bob uses input Y, and honestly interacts with the corrupt Alice. Let T_0 denote the real view of Alice.
- Hybrid$_1$. Let T_1 be the same as T_0, except that all terms of the form $F_k(z)$ are replaced by random values. This hybrid is computationally indistinguishable from T_0 by the security of $\mathcal{F}_{\mathsf{mp\text{-}oprf}}$ and the pseudorandomness of PRF.
- Hybrid$_3$. Let T_3 be the same as T_2, except that the inputs of the Bob are replaced by random items except those with multiplicity greater than one in the intersection (i.e. the items in Y_{leak}). Note that for those $y_j \notin Y_{\mathsf{leak}}$, the value d_i^B is uniform and independent from Alice's view. By the deterministic-value pseudorandomness property of dv-PPRF, T_1 and T_2 are computationally indistinguishable.
- Hybrid$_3$. Let T_3 be the same as T_2, except that the PSU execution is replaced by underlining PSU simulator. The security of $\mathcal{F}_{\mathsf{psu}}$ functionality guarantees that T_3 and T_2 are computationally indistinguishable.
- Hybrid$_4$. Let T_4 be the same as T_3, except the way of computing $\mathsf{id}(x)$. Instead of computing $\mathsf{id}(x_i^{(t)})$ as in step 8, where $\bar{H}(F_{k_A}(x_i) \oplus F_{k_B}(x_i) || t)$ is a uniform random value, we instead compute $\mathsf{id}(x_i^{(t)})$ randomly and then program $\bar{H}(F_{k_A}(x_i) \oplus F_{k_B}(x_i) || t)$ to be the correct value, that is, $\bar{H}(F_{k_A}(x_i) \oplus F_{k_B}(x_i) || t) := \mathsf{id}(x_i^{(t)})$. This change has no effect on Alice's view distribution. This hybrid is exactly the view output by the simulator.

[6] Thus we set $\sigma = \max\{\lambda + \log nU_y, \lambda + \log mU_x\}$.

5 Applications

Private Inner Join. The most direct application of PMID is private inner join. In this scenario, two parties with different datasets/tables want to align their record on some identifiers, e.g. user_id. The parties first perform a PMID protocol with the input of their identifiers (which may contain duplicated elements), then let the parties send their own UID set to the other. The parties match the UID of their own set and the other parties' set, and output the matched elements. The security is guaranteed by the fact that the UID of the element outside a party's set is random to him, and no additional information is leaked from these UIDs.

Private Full Join. Unlike inner join, full join returns all records regardless of whether their identifiers are matched. Assuming Alice obtains the output, we should let Alice obliviously retrieve the elements outside her UID set. Note that PMID protocol can be used for data alignment, that is, after execution of PMID, the parties could sort the UIDs in R^*, e.g. let $R^* = \{r_1, \ldots, r_t\}$ be the sorted set, and define an indication bit string ($a, b \in \{0,1\}^t$ for Alice and Bob separately) as a_i(or b_i) $= 1$ if and only if $r_i \in ID_X$(or ID_Y). In this way, both parties get an aligned indication bit string, i.e. the same bit a_i and b_i indicate the same element whether belongs to their set. Note that if $a_i = 0$, we must have $b_i = 1$ and vice versa. We can use this property to compute full join privately. What we want is letting Alice learn the element correspond to $a_i = 0$, we can let both parties invoke t OTs, and let Bob input $(y_i, \perp)$ for $b_i = 1$ and $(\perp, \perp)$ for $b_i = 0$. In this way, Alice will obtain all the elements corresponding to the whole UIDs set R^*, which is exactly the output of full join.

Private Join for Compute. In this scenario, Alice and Bob want to get a secret sharing of the join result for further complicated computations. The main idea is also to use PMID for data alignment. The parties first compute the indication bit string $a, b \in \{0,1\}^t$ as before. Then the parties share their string to the other, i.e. Alice selects random $a' \overset{\text{R}}{\leftarrow} \{0,1\}^t$, computes $a'' := a \oplus a'$ and sends a'' to Bob, Bob selects random $b' \overset{\text{R}}{\leftarrow} \{0,1\}^t$, computes $b'' := b \oplus b'$ and sends b' to Alice. Then Alice and Bob invoke the AND functionality $\mathcal{F}_{\mathsf{and}}$ with input (a', b') and (a'', b'') respectively. As a result, Alice outputs p and Bob outputs q where $p \oplus q = (a' \oplus a'') \wedge (b' \oplus b'') = a \wedge b$. Note that the AND functionality $\mathcal{F}_{\mathsf{and}}$ could be efficiently implemented from OT. The parties could feed the p and q to any MPC circuit to compute any function they want to compute.

6 Theoretical Analysis of Communication

Recall that we have presented two variants of our protocol. In this section, we will refer to them as:

- Sloppy-PMID: PMID protocol from sloppy OPRF specified in Figure 9.
- Std-PMID: PMID protocol from standard OPRF specified in Figure 10.

We use the state-of-the-art 3H-GCT [13] as our OKVS instantiation. The PSU protocol is instantiated with [12].

In Table 1, we show the theoretical communication complexity of our protocol compared with the BKM+ protocol [5] and the GMRSS protocol [12] in the semi-honest setting. This measures how much communication the protocols require on an idealized network where we don't care about protocol metadata, realistic encodings, byte alignment, etc. In practice, data is split up into multiples of bytes (or CPU words), and different data is encoded with headers, etc. Empirical measurements of such real-world costs are given later in Table 2 and Table 3.

7 Implementation and Performance

In this section, we discuss details of our PMID implementations and report our performances. We also implement state-of-the-art PID protocols [5, 12] under the same experiment setting and report their performances as baselines. Since PMID reduces to PID when $U_x = U_y = 1$, such comparisons would show the additional costs from PID to more general PMID functionalities.

Table 1: Theoretical communication costs of P(M)ID protocols (in bits), calculated using computational security κ= 128 and statistical security λ= 40. Ignore costs of base OTs (in our protocol and GMRSS) which are independent of input size. t is the size of the intersection ($t = n/2$ is used here). ρ is the width of OT extension matrix (depends on n and protocol). σ is the length of items. U_x and U_y are the upper bound of multiplicity of X and Y respectively. m and n are the input sizes of the Alice and Bob respectively.

Protocol	Communication	m=n		
		2^{12}	2^{16}	2^{20}
[5]	$(4m + 4n - 3t)\|G\|$	$1664n$	$1664n$	$1664n$
[12]	$(1.27\rho + 3\sigma)(m + n) + 1.27n\rho$ $+3m\sigma + (1.27n \log n + n)(\kappa + \sigma)$	$5340n$	$6588n$	$7918n$
Std-PMID	$(4.8\kappa + 1.2\sigma)(m + n) + 1.27nU\rho$ $+3mU\sigma + (1.27nU \log nU + nU)(\kappa + \sigma)$	$192U_xn + 3667U_yn$ $+1382n + 244U_y \log U_yn$	$216U_xn + 4823U_yn$ $+1402n + 254U_y \log U_yn$	$240U_xn + 6060U_yn$ $+1421n + 264U_y \log U_yn$
Sloppy-PMID	$(1.27\rho + 4.8\sigma)(m + n) + 1.27nU\rho$ $+3mU\sigma + (1.27nU \log nU + nU)(\kappa + \sigma)$	$192U_xn + 3667U_yn$ $+1712n + 244U_y \log U_yn$	$216U_xn + 4823U_yn$ $+1809n + 254U_y \log U_yn$	$240U_xn + 6060U_yn$ $+1906n + 264U_y \log U_yn$

7.1 Experimental Setup

We ran all our experiments on a single Intel Core i9-9900K with 3.6GHz and 128GB RAM. We execute the protocol on two progresses operated by separated terminals with the network connection built via the local network. We emulate two network connections, namely LAN/WAN configurations, using Linux `tc` command. The LAN setting has a latency 0.02ms and bandwidth of 10Gbps, while the WAN setting has a latency 80ms and bandwidth of 100Mbps. All experiments are done with 128-bit inputs, in which half of the inputs from two parties are in the intersection. In PMID, we let the input of Alice have the form $X = \{(x_1, u^{\mathsf{x}}_1), \dots, (x_n, u^{\mathsf{x}}_n)\} = \{(x_1, U_x), \dots, (x_n, U_x)\}$, and the input of Bob have the form $Y = \{(y_1, u^{\mathsf{y}}_1), \dots, (y_n, u^{\mathsf{y}}_n)\} = \{(y_1, U_y), \dots, (y_n, U_y)\}$. In the one-sided PMID, the input of Alice is simply set as $X = \{(x_1, 1), \dots, (x_n, 1)\}$. In this way, we can have consistent total computation/communication costs under single-thread and multi-thread settings with the same inputs. We used the same methodology and environment to report all performances.

We use an asynchronous event-driven network application framework Netty to maintain the network connection, and use the well-known tool Protocol Buffers for data serialization and deserialization. This meets the compatibility and robustness requirements for industry-designed libraries, so that the reported performance results would reflect the actual costs when deploying protocols in real situations.

7.2 Implementation Details

Existing PID implementations are under different experimental settings. For example, [5] implemented their protocol in Rust programming language with specific libraries that support more efficient Curve25519 elliptic curve cryptography (ECC) operations. On the other hand, [12] implemented their PID protocol in C++ that only supports inputs represented as a 64-bit string (i.e., unsigned long) and 64-bit PID outputs. Note that to achieve the statistical security parameter $\lambda = 40$, the bit length of PID should be set as $\lambda + \log m + \log n$, which would beyond 64 even when m and n are relatively small, i.e., $m, n > 2^{12}$. Such different experimental settings make it hard to have unified comparisons.

We fully re-implemented state-of-the-art PID protocols [5, 12] and their underlying basic protocols using Java, including the base OT construction of [27], the OT extension construction of [21] with the optimization of [1], the batch single-point OPRF of [24] for private equality tests, and the PSU construction of [12] with the multi-thread optimization of [22] for uniting PID/PMID. We did subtle optimizations for our implementations to make our performance results close to or even beyond the ones reported in the original works.

Note that the efficiency of [5] highly depends on the ECC operation efficiency, and base OT also invokes ECC operations. In our experiments, we introduced C/C++ MCL[7] library in our implementations to perform efficient ECC operations and use Java Native Interface (JNI) technique to invoke MCL from Java. We use the curve 'secp256k1', a NIST elliptic curve with 256-bit group elements. For the hash-to-point operation, we use SHA-256 applied to the input, and re-applied until the resulting output lies on the elliptic curve. Such setting has been used in Google's PSI-Sum [19].

[7] `https://github.com/herumi/mcl`

For [12], we did not only re-implement the PID scheme based on "Sloppy OPRF" (Sloppy-[12]) but also implemented the PID scheme based on "standard OPRF" (Std-[12]) by using the lightweight OPRF schemes introduce by [6] as the underlying OPRF. We used the OKVS introduced by [13] in our Std-PMID and Sloppy-PMID. Our PMID implementation covers the general version and the one-sided version. Besides, we also consider the case when both parties' inputs are sets and PMID reduces to its PID counterpart, where all steps for programming multiplicity can be omitted. We leveraged the Fork-Join concurrency technique to support multi-thread computations. We fixed the thread pool size to manually limit the maximal number of threads invoked during our multi-thread experiments. Our complete implementation is available on GitHub[8].

7.3 Performance Analysis

PID Comparisons. The running times and communication costs for existing PID schemes [5, 12] and our PMID schemes when $U_x = U_y = 1$ are shown in Table 2. Observe that the running time and the communication cost of [12] reported in Table 2 are higher than they reported in the original work. This is because [12] supports UID with maximal 64-bit input length, which is not long enough to prevent UID collision under the statistical parameter $\lambda = 40$ when $m, n \in \{2^{14}, 2^{16}, 2^{18}, 2^{20}\}$. The longer UID leads to more costs in PSU and "Sloppy OPRF". We also note that the performance of [5] in our table is slightly better than the original work. This is mainly because we leverage the more efficient ECC library MCL, which introduces assembly language for speeding up the 'secp256k1' ECC operation performances. Since [5] is public-key based, it has the lowest communication of all schemes. Thus it has a better performance in the WAN setting.

The communication cost and the running time of our PMID are identical to that of [12] (both for the standard version and the sloppy version) when $U_x = U_y = 1$. This reflects the fact the PMID reduces to its PID counterpart when both multiplicities are 1.

Scalability and Parallelizability. We demonstrate the scalability and parallelizability of our PMID protocols by evaluating them on set sizes $n = m \in \{2^{14}, 2^{16}, 2^{18}, 2^{20}\}$ with multiplicity $U = 3$ for either party and for both parties. We run each party in parallel with $T \in \{1, 8\}$ threads. We report the performance in Table 3, showing running times in both LAN/WAN settings.

Our PMID protocol scales well when either party has multiplicity 3. When $U_x = 1, U_y = 3$, the running time of our PMID increases by about 2×. When T increases from 1 to 8, we find that our protocol improves by $2.3 - 3.1\times$ in the LAN setting. In the WAN setting, it only speedup about $1.2 - 1.7\times$, which is mainly due to the bandwidth limit.

When both parties have multiplicity $U_x = U_y = 3$, the efficiency of PMID decreases quadratically, which correctly follows the excessive data expansion property for *cross join*. The Java Virtual Machine complains running out of memory when $m = n = 2^{20}$. When $n = m \in \{2^{14}, 2^{16}, 2^{18}\}$, the running time of our PMID increases by about 4× both in the LAN setting and the WAN setting. The result is consistent with the best practice for analytical tasks: except for special cases, avoiding *cross join* because it can blow up the amount of data coming out of the task.

Table 2: Communication (in MB) and run time (in seconds) of the private-ID protocol for input set sizes $n = 2^{14}, 2^{16}, 2^{18}, 2^{20}$ executed over a single thread for LAN and WAN configurations.

Protocols	**LAN(s)**				**WAN(s)**				**Comm(MB)**			
	2^{14}	2^{16}	2^{18}	2^{20}	2^{14}	2^{16}	2^{18}	2^{20}	2^{14}	2^{16}	2^{18}	2^{20}
[5]	4.33	17.4	69.67	277.56	5.07	19.42	75.56	298.05	3.35	13.41	53.63	214.5
Std-[12]	1.86	9.03	4.77	217.51	4.85	17.43	76.96	327.49	16.45	70.51	302.3	1284.47
Sloppy-[12]	1.75	7.82	35.49	162.71	6.02	17.87	73.79	306.53	20.89	87.9	384.28	1602.82
Std-PMID	2.05	9.54	47.56	221.43	5.64	18.41	78.05	326.63	16.45	70.51	302.3	1284.47
Sloppy-PMID	1.75	7.76	35.97	163.73	5.83	18.75	77.88	315.6	20.89	87.9	384.28	1602.82

[8] https://github.com/alibaba-edu/mpc4j

Table 3: Running time (in seconds) of Sloppy-PMID and Std-PMID with set size ($n = m$), number of threads ($T \in \{1, 8\}$) and number of multiplicity ($U \in \{1, 3\}$) in WAN/LAN settings. Cells with ”-” denote setting that program out of memory.

n	Protocol	Multi-plicity		Comm.(MB)			Running time (s)			
							LAN		WAN	
		U_x	U_y	Alice	Bob	Total	T=1	T=8	T=1	T=8
2^{14}	Sloppy-PMID	1	1	9.31	11.58	20.89	1.75	0.7	5.83	4.35
		1	3	15.82	22.73	38.55	3.47	1.53	9.13	7.35
		3	3	43.1	56.09	99.19	7.88	3.21	19.81	16.24
	Std-PMID	1	1	7.09	9.36	16.46	2.05	0.68	5.64	3.95
		1	3	13.6	20.51	34.11	3.82	1.48	9.23	6.84
		3	3	40.88	53.87	94.75	8.42	3.35	20	15.41
2^{16}	Sloppy-PMID	1	1	39.49	48.41	87.9	7.76	3.02	18.75	14.85
		1	3	68.36	95.44	163.8	15.58	6.66	35.04	26.32
		3	3	187.23	237.51	424.74	37.35	16.26	82.3	63.93
	Std-PMID	1	1	30.8	39.71	70.51	9.54	3.24	18.41	13.44
		1	3	59.67	86.75	146.42	17.73	7.03	34.8	24.04
		3	3	178.54	228.82	407.36	38.38	16.3	82.24	60.5
2^{18}	Sloppy-PMID	1	1	174.82	209.46	384.28	35.97	14.94	77.88	56.76
		1	3	299.02	405.66	704.68	72.33	32.88	144	107.13
		3	3	813.55	1010.59	1824.13	181.58	89.62	345.54	268.1
	Std-PMID	1	1	133.83	168.47	302.3	47.56	15.46	78.05	49.78
		1	3	258.03	364.67	622.7	84.51	32.96	147.63	101.62
		3	3	772.56	969.6	1742.15	195.43	92.1	350.43	261.19
2^{20}	Sloppy-PMID	1	1	733.61	869.21	1602.82	163.73	75.93	315.6	230.64
		1	3	1271.21	1690.33	2961.54	347.49	173.61	608.68	449.01
		3	3	-	-	-	-	-	-	-
	Std-PMID	1	1	574.44	710.03	1284.47	221.43	77.49	326.63	203.64
		1	3	1112.04	1531.16	2643.19	405.15	177.51	628.13	422.77
		3	3	-	-	-	-	-	-	-

Acknowledgement

We are grateful for the helpful comments from the anonymous reviewers. Weiran Liu is supported by the Major Programs of the National Social Science Foundation of China (Grant No. 22&ZD147). Cong Zhang and Dongdai Lin are supported by the National Key Research and Development Program of China (No. 2020YFB1805402) and the National Natural Science Foundation of China (Grants No. 61872359 and No. 61936008).

References

[1] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS 2013*, 2013.

[2] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel N. Kho, and Jennie Rogers. SMCQL: secure query processing for private data networks. *Proc. VLDB Endow.*, 10(6):673–684, 2017.

[3] Johes Bater, Xi He, William Ehrich, Ashwin Machanavajjhala, and Jennie Rogers. Shrinkwrap: Efficient SQL query processing in differentially private data federations. *Proc. VLDB Endow.*, 12(3):307–320, 2018.

[4] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In *ASIACCS 2012*, 2012.

[5] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, 2020. https://ia.cr/2020/599.

[6] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In *CRYPTO 2020*, 2020.

[7] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In *ACISP 2017*, 2017.

[8] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *CCS 2013*, 2013.

[9] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC 2005*, 2005.

[10] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT 2004*, 2004.

[11] Keith B. Frikken. Privacy-preserving set union. In *ACNS 2007*, 2007.

[12] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In *PKC 2021*, 2021.

[13] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Oblivious key-value stores and amplification for private set intersection. In *CRYPTO 2021*, 2021.

[14] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[15] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *STOC 1987*, 1987.

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

[17] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security*, 2011.

[18] Bernardo A. Huberman, Matthew K. Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Electronic Commerce (EC-99)*, 1999.

[19] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. On deploying secure computing: Private intersection-sum-with-cardinality. In *EuroS&P 2020*, 2020.

[20] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. *IACR Cryptol. ePrint Archive 2017/738*, 2017.

[21] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, 2003.

[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.

[23] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *CRYPTO 2005*, 2005.

[24] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *CCS 2016*, 2016.

[25] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In *CCS 2017*, 2017.

[26] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In *ASIACRYPT*, 2019.

[27] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms*, 2001.

[28] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings*, pages 121–133, 2001.

[29] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *CRYPTO 2019*, 2019.

[30] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from paxos: Fast, malicious private set intersection. In *EUROCRYPT 2020*, 2020.

[31] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *USENIX Security*, 2014.

[32] 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.

[33] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. Senate: A maliciously-secure MPC platform for collaborative analytics. In *USENIX Security 2021*, 2021.

[34] Meikel Pöss, Bryan Smith, Lubor Kollár, and Per-Åke Larson. Tpc-ds, taking decision support benchmarking to the next level. In *Proceedings of the 2002 ACM SIGMOD International Conference on*

Management of Data, pages 582–587, 2002.

[35] Michael O. Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptol. ePrint Arch.*, 2005:187, 2005.

[36] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: fast OPRF and circuit-psi from vector-ole. In *EUROCRYPT 2021*, 2021.

[37] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

A Proof of Partial Obliviousness

We first give the formal definition of linear OKVS as follows:

Definition 4 (Linear OKVS). *An OKVS is linear (over a field $\mathbb{F}$) if $\mathcal{V} = \mathbb{F}$ ("values" are elements of $\mathbb{F}$), the output of* Encode *is a vector D in $\mathbb{F}^m$, and the* Decode *function is defined as:*

$$\mathsf{Decode}_H(D, x) = \langle \mathsf{row}(x), D \rangle := \sum_{j=1}^{m} \mathsf{row}(x)_j D_j$$

for some function $\mathsf{row} : \mathcal{K} \to \mathbb{F}^m$. Hence Decode *is a linear map from $\mathbb{F}^m$ to $\mathbb{F}$.*

The mapping $\mathsf{row} : \mathcal{K} \to \mathbb{F}^m$ are typically defined by the hash function H.

For a linear OKVS, one can view the Encode function as generating a solution to the linear system of equations:

$$\begin{bmatrix} -\mathsf{row}(x_1)- \\ -\mathsf{row}(x_2)- \\ \vdots \\ -\mathsf{row}(x_n)- \end{bmatrix} D^T = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix} \tag{1}$$

Theorem 4. *When* Encode_H *chooses uniformly from the set of solutions to the linear system, the linear OKVS satisfies the partial obliviousness property.*

Proof. Now we prove the two distribution of D are statistically indistinguishable. We use $RD^T = Y$ to represent equation 1 and we decompose the matrix as

$$\begin{bmatrix} R_1 \\ R_2 \end{bmatrix} D^T = \begin{bmatrix} Y_1 \\ Y_2 \end{bmatrix}$$

where R_1 and Y_1 correspond to the first t rows of the matrix, and R_2 and Y_2 correspond to the last $n - t$ rows. We use $\mathcal{D}_{X,Y}$ to represent all possible outputs of $\mathsf{Encode}_H(X, Y)$. We have $D \leftarrow \mathsf{Encode}_H(X, Y) \iff D \overset{\mathrm{R}}{\leftarrow} \mathcal{D}_{X,Y}$.

We denote the two distributions in the definition of partial obliviousness as W_1 and W_2 respectively. Since there are t fixed key-value pairs $(x_1, y_1), \dots, (x_t, y_t)$, both outputs of W_1 and W_2 must satisfy $R_1 D^T = Y_1$.

For any $D_0 \in \mathbb{F}^m$ constrained on $R_1 D_0^T = Y_1$, we have $Pr[Y_2 \overset{\mathrm{R}}{\leftarrow} \mathbb{F}^{n-t} : R_2 D_0^T = Y_2] = \frac{1}{|\mathbb{F}|^{n-t}}$ and thus $Pr[D \leftarrow \mathsf{Encode}_H(X, Y) : D = D_0 | Y_2 \neq R_2 D_0^T] = 0$. The distribution of W_1 is as follows:

$$\begin{aligned} &Pr[D \leftarrow W_1 : D = D_0] = Pr[Y_2 \overset{\mathrm{R}}{\leftarrow} \mathbb{F}^{n-t}, D \overset{\mathrm{R}}{\leftarrow} \mathcal{D}_{X,Y} : D = D_0] \\ &= \sum_{Y_2' \in \mathbb{F}^{n-t}} Pr[Y_2 \overset{\mathrm{R}}{\leftarrow} \mathbb{F}^{n-t} : Y_2 = Y_2'] \cdot Pr[D \overset{\mathrm{R}}{\leftarrow} \mathcal{D}_{X,Y} : D = D_0 | Y_2 = Y_2'] \\ &= Pr[Y_2 \overset{\mathrm{R}}{\leftarrow} \mathbb{F}^{n-t} : Y_2 = R_2 D_0^T] \cdot Pr[D \overset{\mathrm{R}}{\leftarrow} \mathcal{D}_{X,Y} : D = D_0 | Y_2 = R_2 D_0^T] \\ &= \frac{1}{|\mathbb{F}|^{n-t}} \cdot \frac{1}{|\mathcal{D}_{X,Y}|} \end{aligned}$$

The only difference between W_1 and W_2 is that the constant matrix R_2 is different, which does not affect the probability. Similarly, we obtain $Pr[D \leftarrow W_2 : D = D_0] = \frac{1}{|\mathbb{F}|^{n-t}} \cdot \frac{1}{|\mathcal{D}_{X,Y}|}$.

Parameters:

- Two parties: Alice and Bob.
- An OKVS scheme $(\mathsf{Encode}_H, \mathsf{Decode}_H)$.
- Ideal $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$ primitives specified in Figure 6.
- Ideal $\mathcal{F}_{\mathsf{bsp\text{-}oprf}}$ primitives specified in Figure 3.
- Ideal $\mathcal{F}_{\mathsf{psu}}$ primitives specified in Figure 5.
- A PRF $F : \{0,1\}^* \to \{0,1\}^\sigma$.
- Random oracle $\bar{H} : \{0,1\}^* \to \{0,1\}^l$.
- Random hash functions $h_1, \dots, h_{\alpha_1} : \{0,1\}^* \to [\rho_1]$ and $h'_1, \dots, h'_{\alpha_2} : \{0,1\}^* \to [\rho_2]$.

Input of Alice: $X = \{(x_1, u^{\mathsf{x}}_1), \dots, (x_m, u^{\mathsf{x}}_m)\} \subset \{0,1\}^* \times [U_x]$. In the one-sided version, $u^{\mathsf{x}}_1 = \dots = u^{\mathsf{x}}_m = 1$, i.e. the input of Alice is a set. Let $X' := \{x_1 \dots, x_m\}$ be the set without duplication items corresponding to X.
Input of Bob: $Y = \{(y_1, u^{\mathsf{y}}_1), \dots, (y_n, u^{\mathsf{y}}_n)\} \subset \{0,1\}^* \times [U_y]$. Let $Y' := \{y_1 \dots, y_n\}$ be set without duplication items corresponding to Y.
Protocol:

1. **(Sloppy OPRF Bob $\to$ Alice)** Alice does $\mathcal{A} \leftarrow \mathsf{Cuckoo}^{\rho_1}_{h_1,\dots,h_{\alpha_1}}(X')$.
2. The parties call $\mathcal{F}_{\mathsf{bsp\text{-}oprf}}$, where Alice is the receiver with input $\mathcal{A}$ and Bob is the sender. Bob receives output $(k^B_1, \dots, k^B_{\rho_1})$ and Alice receives output $(f^A_1, \dots, f^A_{\rho_1})$. Alice output is such that, for each $x \in X$, assigned to bin u by hash function h_v, we have $f^A_u = F_{k^B_u}(x||v)$.
3. Bob chooses a random PRF key s^B, he computes an OKVS $P^B := \mathsf{Encode}_H(\{(y||v, F_{s^B}(y) \oplus F_{k^B_{h_v(y)}}(y||v))\}_{y\in Y', v\in[\alpha_1]})$ and sends P^B to Alice.
4. For each item x that Alice assigned to a bin with hash function h_v, Alice defines $r^A(x) := \mathsf{Decode}_H(P^B, x||v) \oplus f^A_{h_v(x)} \oplus F_{s^A}(x)$.
5. **(Sloppy OPRF Alice $\to$ Bob)** Bob does $\mathcal{B} \leftarrow \mathsf{Cuckoo}^{\rho_2}_{h'_1,\dots,h'_{\alpha_2}}(Y')$.
6. The parties call $\mathcal{F}_{\mathsf{bsp\text{-}oprf}}$, where Bob is the receiver with input $\mathcal{B}$ and Alice is the sender. Alice receives output $(k^A_1, \dots, k^A_{\rho_2})$ and Bob receives output $(f^B_1, \dots, f^B_{\rho_2})$. Bob output is such that, for each $y \in Y'$, assigned to bin u by hash function h_v, we have $f^B_u = F_{k^A_u}(y||v)$.
7. Alice chooses a random PRF key s^A, she computes an OKVS $P^A := \mathsf{Encode}_H(\{(x||v, F_{s^A}(x) \oplus F_{k^A_{h_v(x)}}(x||v))\}_{x\in X, v\in[\alpha_2]})$ and sends P^A to Bob.
8. For each item y that Bob assigned to a bin with hash function h_v, Bob defines $r^B(y) := \mathsf{Decode}_H(P^A, y||v) \oplus f^B_{h_v(y)} \oplus F_{s^B}(x)$.
9. **(Program Multiplicity)** For $i \in [m]$, if $u^{\mathsf{x}}_i = 1$, Alice selects a random $c^{\mathsf{x}}_i \xleftarrow{R} \{0,1\}^\sigma$, else defines $c^{\mathsf{x}}_i := u^{\mathsf{x}}_i$. For $j \in [n]$, if $u^{\mathsf{y}}_j = 1$, Bob selects a random $c^{\mathsf{y}}_j \xleftarrow{R} \{0,1\}^\sigma$, else defines $c^{\mathsf{y}}_j := u^{\mathsf{y}}_j$. Note that here we pad u^{x}_i and u^{y}_j with 0 from $\log U_x$ and $\log U_y$ bits to σ bits.
10. The parties call $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$, where Bob is sender with input $\{(y_j, c^{\mathsf{y}}_j)\}_{j\in[n]}$ and receives (k_B, hint_B), and Alice is receiver with input X'. As a result, Alice receives $\mathsf{hint}_B, \{d^B_i := F(k_B, \mathsf{hint}_B, x_i)\}_{i\in[m]}$.
11. For $i \in [m]$, if $1 < d^B_i \le U_y$, Alice defines $\bar{u}^{\mathsf{x}}_i := u^{\mathsf{x}}_i \cdot d^B_i$; else $\bar{u}^{\mathsf{x}}_i := u^{\mathsf{x}}_i$. Alice also defines $Y_{\mathsf{leak}} := \{(x_i, d^B_i) | i \in [m], 1 < d^B_i \le U_y\}$.
12. The parties call $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$, where Alice is sender with input $\{(x_i, c^{\mathsf{x}}_i)\}_{i\in[m]}$ and receives (k_A, hint_A), and Bob is receiver with input Y'. As a result, Bob receives $\mathsf{hint}_A, \{d^A_j := F(k_A, \mathsf{hint}_A, y_j)\}_{j\in[n]}$.
13. For $j \in [n]$, if $1 < d^A_j \le U_x$, Bob defines $\bar{u}^{\mathsf{y}}_j := u^{\mathsf{y}}_j \cdot d^A_j$; else $\bar{u}^{\mathsf{y}}_j := u^{\mathsf{y}}_j$. Bob also defines the leaked set $X_{\mathsf{leak}} := \{(y_j, d^A_j) | j \in [n], 1 < d^A_j \le U_x\}$. In the one-sided version, Bob defines $\bar{u}^{\mathsf{y}}_j := u^{\mathsf{y}}_j$ for $j \in [n]$.
14. **(ID computation)** For $i \in [m]$, $t \in [\bar{u}^{\mathsf{x}}_i]$, Alice computes $\mathsf{id}(x^{(t)}_i) := \bar{H}(r^A(x_i)||t)$. Let $ID_X := \{\mathsf{id}(x^{(t)}_i) | i \in [m], t \in [\bar{u}^{\mathsf{x}}_i]\}$.
15. For $j \in [n]$, $t \in [\bar{u}^{\mathsf{y}}_j]$, Bob computes $\mathsf{id}(y^{(t)}_j) := \bar{H}(r^B(y_j)||t)$. Let $ID_Y := \{\mathsf{id}(y^{(t)}_j) | j \in [n], t \in [\bar{u}^{\mathsf{y}}_j]\}$.
16. **(Union)** Alice and Bob invoke the PSU functionality $\mathcal{F}_{\mathsf{psu}}$ with input ID_X and ID_Y respectively. As a result, Bob receives $R^* := ID_X \cup ID_Y$ and sends R^* to Alice.

Fig. 9: PMID Protocol Π_{PMID} from Sloppy OPRF. The highlighted parts could omit for one-sided version.

Parameters:

- Two parties: Alice and Bob.
- Ideal $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$ primitives specified in Figure 6.
- Ideal $\mathcal{F}_{\mathsf{mp\text{-}oprf}}$ primitives specified in Figure 4 and underlining PRF $F : \{0,1\}^* \to \{0,1\}^\sigma$.
- Ideal $\mathcal{F}_{\mathsf{psu}}$ primitives specified in Figure 5.
- Random oracle $\bar{H} : \{0,1\}^* \to \{0,1\}^l$

Input of Alice: $X = \{(x_1, u_1^{\mathsf{x}}), \dots, (x_m, u_m^{\mathsf{x}})\} \subset \{0,1\}^* \times [U_x]$. In the one-sided version, $u_1^{\mathsf{x}} = \cdots = u_m^{\mathsf{x}} = 1$, i.e. the input of Alice is a set. Let $X' := \{x_1 \dots, x_m\}$ be the set without duplication items corresponding to X.
Input of Bob: $Y = \{(y_1, u_1^{\mathsf{y}}), \dots, (y_n, u_n^{\mathsf{y}})\} \subset \{0,1\}^* \times [U_y]$. Let $Y' := \{y_1 \dots, y_n\}$ be the set without duplication items corresponding to Y.
Protocol:

1. **(OPRF)** Alice and Bob invoke the multi-point OPRF functionality $\mathcal{F}_{\mathsf{mp\text{-}oprf}}$. Alice acts as the sender and Bob acts as the receiver with input Y'. As a result, Alice receives a PRF key k_A and Bob receives $\{F_{k_A}(y) | y \in Y'\}$.
2. Alice and Bob invoke another multi-point OPRF functionality $\mathcal{F}_{\mathsf{mp\text{-}oprf}}$. Bob acts as the sender and Alice acts as the receiver with input X'. As a result, Bob receives a PRF key k_B and Alice receives $\{F_{k_B}(x) | x \in X\}$.
3. **(Program Multiplicity)** For $i \in [m]$, if $u_i^{\mathsf{x}} = 1$, Alice selects a random $c_i^{\mathsf{x}} \xleftarrow{\text{R}} \{0,1\}^\sigma$, else defines $c_i^{\mathsf{x}} := u_i^{\mathsf{x}}$. For $j \in [n]$, if $u_j^{\mathsf{y}} = 1$, Bob selects a random $c_j^{\mathsf{y}} \xleftarrow{\text{R}} \{0,1\}^\sigma$, else defines $c_j^{\mathsf{y}} := u_j^{\mathsf{y}}$. Note that here we pad u_i^{x} and u_j^{y} with 0 from $\log U_x$ and $\log U_y$ bits to σ bits.
4. The parties call $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$, where Bob is sender with input $\{(y_j, c_j^{\mathsf{y}})\}_{j \in [n]}$ and receives (k_B, hint_B), and Alice is receiver with input X'. As a result, Alice receives $\mathsf{hint}_B, \{d_i^B := F(k_B, \mathsf{hint}_B, x_i)\}_{i \in [m]}$.
5. For $i \in [m]$, if $1 < d_i^B \le U_y$, Alice defines $\bar{u}_i^{\mathsf{x}} := u_i^{\mathsf{x}} \cdot d_i^B$; else $\bar{u}_i^{\mathsf{x}} := u_i^{\mathsf{x}}$. Alice also defines $Y_{\mathsf{leak}} := \{(x_i, d_i^B) | i \in [m], 1 < d_i^B \le U_y\}$.
6. The parties call $\mathcal{F}_{\mathsf{dv\text{-}OPPRF}}$, where Alice is sender with input $\{(x_i, c_i^{\mathsf{x}})\}_{i \in [m]}$ and receives (k_A, hint_A), and Bob is receiver with input Y'. As a result, Bob receives $\mathsf{hint}_A, \{d_j^A := F(k_A, \mathsf{hint}_A, y_j)\}_{j \in [n]}$.
7. For $j \in [n]$, if $1 < d_j^A \le U_x$, Bob defines $\bar{u}_j^{\mathsf{y}} := u_j^{\mathsf{y}} \cdot d_j^A$; else $\bar{u}_j^{\mathsf{y}} := u_j^{\mathsf{y}}$. Bob also defines the leaked set $X_{\mathsf{leak}} := \{(y_j, d_j^A) | j \in [n], 1 < d_j^A \le U_x\}$. In the one-sided version, Bob defines $\bar{u}_j^{\mathsf{y}} := u_j^{\mathsf{y}}$ for $j \in [n]$.
8. **(ID computation)** For $i \in [m]$, $t \in [\bar{u}_i^{\mathsf{x}}]$, Alice computes $\mathsf{id}(x_i^{(t)}) := \bar{H}(r^A(x_i) || t)$. Let $ID_X := \{\mathsf{id}(x_i^{(t)}) | i \in [m], t \in [\bar{u}_i^{\mathsf{x}}]\}$.
9. For $j \in [n]$, $t \in [\bar{u}_j^{\mathsf{y}}]$, Bob computes $\mathsf{id}(y_j^{(t)}) := \bar{H}(r^B(y_j) || t)$. Let $ID_Y := \{\mathsf{id}(y_j^{(t)}) | j \in [n], t \in [\bar{u}_j^{\mathsf{y}}]\}$.
10. **(Union)** Alice and Bob invoke the PSU functionality $\mathcal{F}_{\mathsf{psu}}$ with input ID_X and ID_Y respectively. As a result, Bob receives $R^* := ID_X \cup ID_Y$ and sends R^* to Alice.

Fig. 10: PMID Protocol Π_{PMID} from Standard OPRF. The highlighted parts could omit for one-sided version.