Linear Private Set Union from Multi-Query Reverse Private Membership Test

Cong Zhang[1,2], Yu Chen[3,4,5(✉)], Weiran Liu[6], Min Zhang[3,4,5], and Dongdai Lin[1,2]

[1] State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China

{zhangcong,ddlin}@iie.ac.cn

[3] School of Cyber Science and Technology, Shandong University, Qingdao 266237, China

[4] State Key Laboratory of Cryptology, P.O. Box 5159, Beijing 100878, China

[5] Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao 266237, China

yuchen.prc@gmail.com, zm_min@mail.sdu.edu.cn

[6] Alibaba Group

weiran.lwr@alibaba-inc.com

Abstract. Private set union (PSU) protocol enables two parties, each holding a set, to compute the union of their sets without revealing anything else to either party. So far, there are two known approaches for constructing PSU protocols. The first mainly depends on additively homomorphic encryption (AHE), which is generally inefficient since it needs to perform a non-constant number of homomorphic computations on each item. The second is mainly based on oblivious transfer and symmetric-key operations, which is recently proposed by Kolesnikov et al. (ASIACRYPT 2019). It features good practical performance, which is several orders of magnitude faster than the first one. However, neither of these two approaches is optimal in the sense that their computation and communication complexity are not both $O(n)$, where n is the size of the set. Therefore, the problem of constructing the optimal PSU protocol remains open.

In this work, we resolve this open problem by proposing a generic framework of PSU from oblivious transfer and a newly introduced protocol called multi-query reverse private membership test (mq-RPMT). We present two generic constructions of mq-RPMT. The first is based on symmetric-key encryption and general 2PC techniques. The second is based on re-randomizable public-key encryption. Both constructions lead to PSU with linear computation and communication complexity.

We implement our two PSU protocols and compare them with the state-of-the-art PSU. Experiments show that our PKE-based protocol has the lowest communication of all schemes, which is $3.7-14.8\times$ lower depending on set size. The running time of our PSU scheme is $1.2-12\times$ faster than that of state-of-the-art depending on network environments.

1 Introduction

Private set union (PSU) enables two parties, each holding a private set of elements, to compute the union of the two sets while revealing nothing more than the union itself. PSU and its variants have numerous applications [LV04, HLS+16, RMY20, BS05, KS05, KRTW19, GMR+21, JSZ+22]. An important PSU application is IP blacklist and vulnerability data aggregation [HLS+16, RMY20]. Consider two organizations (i.e. the maintainers of the IP blacklists) want to compute their IP blacklist joint list, which will help minimize vulnerabilities in their infrastructure. However, it is not secure to let the organizations simply exchange their blacklists because each individual IP blacklist is generated according to the detection strategy formulated by the maintainer and cannot be leaked. Note that a curious organization may infer the detection strategy of another organization from the IP address in the intersection. Therefore, it is important to hide the intersection, which is exactly the functionality of PSU.

Another killer application of PSU is to construct Private-ID protocol [BKM+20, GMR+21]. The Private-ID 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. The main use of Private ID 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. Garimella et al. [GMR+21] gave a modular way to construct Private ID from Obivious PRF (OPRF) and PSU. Their experiments showed that the bottleneck of their Private ID is the underlining PSU instantiations.

In addition, PSU applications also include information security risk assessment [LV04], joint graph computation [BS05], distributed network monitoring [KS05], building block for private DB supporting full join [KRTW19] etc.

Over the last decade, there has been a significant amount of work on private set operation, especially private set intersection (PSI) [FNP04, PSZ14, KKRT16, PRTY19, CM20, PRTY20]. We refer the reader to [PSZ18] for an overview of different PSI paradigms. State-of-the-art semi-honest PSI protocols in the two-party setting [KKRT16, PRTY19, CM20, RS21, GPR+21] all mainly rely on symmetric-key operations, except for a few base OT operations in OT extension protocol [IKNP03, KK13]. Let n denote the size of input set, the communication complexity of these OT-based PSI protocols has been improved from initial nonlinear $O(n \log n)$ [PSZ14, PSSZ15, KKRT16] to linear complexity $O(n)$ [PRTY19, FNO19, GN19, CM20, RS21, GPR+21].

1.1 Motivation

In contrast to the affairs of PSI, the efficiency of the state-of-the-art PSU is less satisfactory. Roughly, there are two known approaches for constructing PSU protocols. The first is mainly based on public-key techniques. Existing constructions along this approach [KS05, Fri07, HN10, SM18] have to perform a non-constant number of additively homomorphic encryption (AHE) operations on each set element, rendering the overall protocols inefficient. The other is mainly based on symmetric-key techniques in combination with OT [KRTW19, GMR+21, JSZ+22], which is several orders of magnitude faster than AHE-based constructions. However, neither of the two approaches is optimal in the sense that their computation and communication complexity are not both $O(n)$, where n is the size of the set. We note that [DC17] is the work closest to optimal bound, but its communication and computation complexity additionally depend on the statistical security parameter λ. This leaves the following open problem:

Can we construct PSU protocols with linear computation and communication complexity?

1.2 Our Contribution

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

1. We revisit the PSU protocol [KRTW19] (KRTW protocol for short hereafter) in depth. Roughly, KRTW protocol is built upon two building blocks, namely oblivious transfer (OT) and reverse private membership test (RPMT). We figure out the root causing KRTW protocol non-optimal is that RPMT has linear communication complexity and super-linear computation complexity, and it has to be carried out n times independently, where n is the size of sender's private set.
2. To achieve linear complexity, we propose a new framework for constructing PSU protocols. The core building block is a newly introduced protocol called multi-query RPMT (mq-RPMT). We identify and overcome several technical difficulties for building optimal mq-RPMT, and give two realizations of mq-RPMT. Both the two concrete mq-RPMT protocols achieve linear communication and computation complexity.
3. We further abstract a new primitive called membership encryption (ME), which broadens the scope of the candidate encryption scheme, unifies our two constructions, and halves the communication complexity of our SKE-based construction on receiver side.
4. Combining OT and the above mq-RPMT, we eventually obtain SKE-based and PKE-based PSU protocols with optimal complexity for the first time. Experiments show that our PKE-based protocol has the lowest communication of all schemes, which is $3.7 - 14.8\times$ lower depending on set size. The running time of our PSU scheme is $1.2 - 12\times$ faster than that of state-of-the-art depending on network environments. In addition to our scheme, we also use Silent OT [BCG+19, YWL+20] to optimize the scheme of [GMR+21, JSZ+22], and provide different parameter selection of Ferret OT [YWL+20].

Figure 1 depicts the technical overview of our new PSU framework. We elaborate the details in the next subsection.

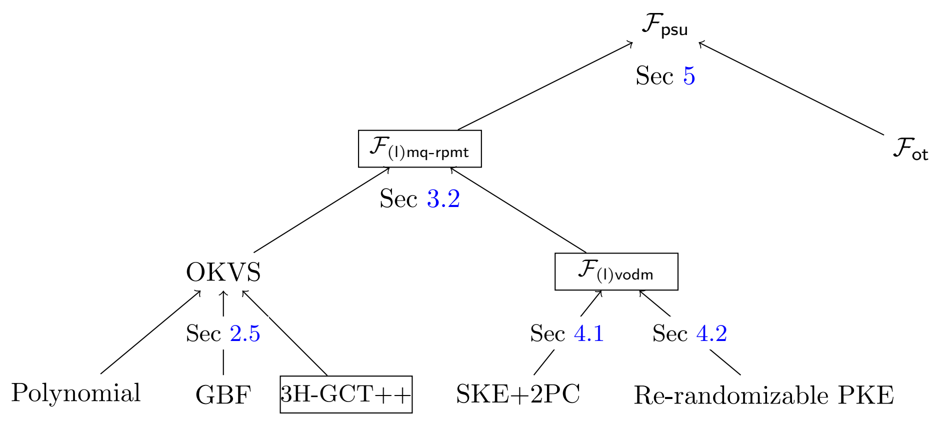
$\mathcal{F}_{\mathsf{psu}}$
Sec 5
$\mathcal{F}_{\mathsf{(l)mq\text{-}rpmt}}$
$\mathcal{F}_{\mathsf{ot}}$
Sec 3.2
OKVS
$\mathcal{F}_{\mathsf{(l)vodm}}$
Sec 2.5
Sec 4.1
Sec 4.2
Polynomial
GBF
3H-GCT++
SKE+2PC
Re-randomizable PKE

Fig. 1: Technical overview of our new PSU framework. The new primitives and functionalities are marked with rectangles.

1.3 Overview of Our Techniques

We provide the high-level technical overview for our new framework of PSU protocol.

KRTW protocol revisit. Our starting point is the recent PSU protocol of Kolesnikov et al. [KRTW19]. The core of KRTW protocol is a subprotocol called reverse private membership test (RPMT), which can test whether a sender's element y belongs to the receiver's input set X, and let the receiver obtain the result. After that, both parties execute OT protocol to let the receiver obtain $\{y\} \cup X$. The computation cost of original RPMT [KRTW19] is $O(n \log^2 n)$ and the communication cost is $O(n)$. For the purpose of computing the set union, the parties need to execute RPMT n times independently, which results in $O(n^2)$ communication and $O(n^2 \log^2 n)$ computation. The complexity can be further reduced to $O(n \log n)$ and $O(n \log n \log \log n)$ separately via hash to bin technology, but it is still *super-linear*. The bottleneck of the KRTW protocol is exactly RPMT.
Zoom in on the original RPMT. The original RPMT protocol employs an oblivious PRF (OPRF) functionality $\mathcal{F}_{\mathsf{oprf}}$ and a private equality test (PEQT) functionality $\mathcal{F}_{\mathsf{peqt}}$. In OPRF, the sender learns a random PRF key k and the receiver learns the PRF output $F_k(y_1), \ldots, F_k(y_n)$ on its inputs $y_1, \ldots, y_n \in Y$. In PEQT, the functionality receives two strings from the receiver and the sender respectively and tells the receiver whether the two strings are equal. Their RPMT uses an *indication string* s to indicate the membership of X.

More precisely, their RPMT protocol executes as follows with sender $\mathcal{S}$'s input y and receiver $\mathcal{R}$'s input $X = \{x_1, \ldots, x_n\}$: $\mathcal{S}$ and $\mathcal{R}$ execute the OPRF protocol first. The receiver $\mathcal{R}$ receives a PRF key k. The sender $\mathcal{S}$ inputs y, and receives $q^* = F_k(y)$. Next, $\mathcal{R}$ chooses a random indication string s. Then, $\mathcal{R}$ computes and sends the interpolation polynomial P which passes through points $\{(x_i, s \oplus F_k(x_i))\}_{i \in [n]}$ to the sender. After receiving P, $\mathcal{S}$ computes $s^* := q^* \oplus P(y)$. Now, $\mathcal{S}$ and $\mathcal{R}$ invoke the $\mathcal{F}_{\mathsf{peqt}}$-functionality with input s^* and s separately. Finally, $\mathcal{R}$ receives output from $\mathcal{F}_{\mathsf{peqt}}$.

If $y \in X$, i.e., there exists an x_i such that $y = x_i$, then we have $s^* = q^* \oplus P(y) = F_k(x_i) \oplus P(x_i) = s$. If $y \notin X$, then $q^* = F_k(y)$ is pseudorandom, which implies that $s^* = q^* \oplus P(y) \neq s$ with overwhelming probability.

To identify the root of the inefficiency of the original RPMT protocol, we first try to interpret it at an abstract level. Our first key observation is that the polynomial actually plays the role of oblivious key-value store (OKVS). Our second key observation is that the usage of OPRF is three-fold. Firstly, $\mathcal{R}$ uses an OPRF to derive n pseudorandom one-time pads, then encrypts the same indication string into n ciphertexts under these one-time pads. Secondly, $\mathcal{S}$ utilizes OPRF to decrypt a ciphertext obliviously. Finally, OPRF provides OKVS with randomness to ensure the correctness of the protocol.

Based on the above new interpretation, we are ready to describe our new mq-RPMT protocol in an incremental way over the original RPMT protocol.

Enhanced oblivious key-value store. One reason that accounts for the super-linear complexity of the original RPMT protocol is that the polynomial related operations are costly. More precisely, the complexity of polynomial interpolation is $O(n \log^2 n)$, and the amortized complexity of polynomial evaluation is $O(\log^2 n)$. According to our first observation, polynomial essentially plays the role of OKVS. This greatly increases the space of the concrete mapping schemes that can be used. A drop-in replacement of polynomial with more efficient OKVS candidates can reduce the computation complexity immediately. However, as we observed before, an additional randomness property should be satisfied now, since we do not use OPRF to provide randomness anymore. To achieve this goal, we enhance OKVS in two aspects: efficiency and security. (See Section 2.5 for the details.)

Oblivious decryption-then-matching. Another reason that accounts for the super-linear complexity is that the original RPMT protocol is one-time in nature. To see this, note that in the original RPMT protocol $\mathcal{S}$ learns the purported indication string. This design lets $\mathcal{S}$ learn more information than needed, and is exactly the reason that hinders multi-query. For example, if there are two distinct elements belonging to $\mathcal{R}$'s set, then $\mathcal{S}$ will obtain the same indication string. This will let $\mathcal{S}$ know that the two elements belong to the intersection, which violates security.

Based on the above discussion, the rough idea of making RPMT support multi-query is to encode the *ciphertext* of indication string in OKVS instead of the indication string itself. In this way, $\mathcal{S}$ will obtain some ciphertexts (i.e. the value of $\mathsf{OKVS}(y)$), and $\mathcal{R}$ has the corresponding key. We need to let $\mathcal{R}$ decrypt these ciphertexts, and match the results with the indication string. A naive attempt is to have $\mathcal{S}$ directly send the ciphertexts to $\mathcal{R}$, and in the sequel, $\mathcal{R}$ tries to decrypt and match. However, this rough idea is problematic since it is insecure even against a semi-honest receiver. Consider $\mathcal{R}$ records the correspondence between x_i and $\mathsf{OKVS}(x_i)$. In this way, $\mathcal{R}$ is able to learn $\mathcal{S}$'s private input y by simple look-up when $y \in X$, rather than merely the fact that $y \in X$. We overcome this difficulty in two steps. The first step is to re-factor the functionality of OPRF to encryption and oblivious decryption functionality. Let $\mathcal{R}$ encrypt the indication string locally. Then $\mathcal{R}$ computes the corresponding OKVS and sends it to $\mathcal{S}$. To ensure the overall protocol still constitutes an RPMT protocol, the second step is to merge the oblivious decryption functionality and PEQT into a new functionality, namely, vector oblivious decryption-then-matching (VODM) functionality. In this functionality, the sender inputs a vector of ciphertexts and the receiver inputs a key and a plaintext. The functionality decrypts these ciphertexts with the key and matches the results with the plaintext input by the receiver. If it matches, the receiver outputs 1, and outputs 0 otherwise.

Putting all the pieces together, we can build mq-RPMT protocol from OKVS, encryption, and VODM functionality in a modular way. (See Section 3 for the technical details).

Two generic constructions of mq-RPMT. Our first generic construction chooses probabilistic SKE as the encryption scheme, and resorts to general 2PC to implement the VODM functionality. See Section 4.1 for details. Our second generic construction chooses re-randomizable PKE as the encryption scheme and uses re-randomization technique to implement VODM functionality, without resorting to generic 2PC.

Our idea is to let $\mathcal{S}$ re-randomize all the ciphertexts and then send the results to $\mathcal{R}$. In this way, $\mathcal{R}$ fulfills the decryption-then-matching functionality in an oblivious manner for all $y_i \in X$. We note that this method will leak some information of $y \notin X$, however, as observed by KRTW, this leakage does not cause any harm to the PSU, since the PSU protocol releases that value anyway.

Looking ahead, one may doubt our PKE-based scheme is inefficient. We note that our PKE-based scheme can still be very efficient because we use PKE techniques in an entirely different way compared to prior PKE-based protocol [KS05, Fri07, DC17]. We only need to perform the encryption, rerandomization, and decryption operations per item, while they need to carry out many ciphertext homomorphism operations per item. See Section 4.2 for details.

Optimization with membership encryption. In the above framework, the underlying encryption schemes must be probabilistic to make the security proof go through. As a consequence, this incurs considerable overhead on communication costs due to ciphertext expansion. Observe that the VODM functionality reveals only one-bit information for every ciphertext. A second thought indicates that a full-fledged encryption scheme might be overkill for our construction of mq-RPMT protocol, and a new type of encryption scheme suffices. We propose the new encryption scheme as membership encryption (ME).

We sketch the definition of ME in the symmetric key setting as below. Let X be a string set. The encryption algorithm takes a key k and an element $x_i \in X$ as inputs, outputs a ciphertext c. The decryption algorithm takes a key k and a ciphertext c as inputs, outputs a bit to indicate if the encrypted element belongs to X. For the correctness, we require that for any $x_i \in X$ and any $c \leftarrow \mathsf{Enc}(k, x_i)$, we have $\mathsf{Dec}(k, c) = 1$. The security requirement is multi-element pseudorandomness, namely, $\{\mathsf{Enc}(k, x_i)\}_{x_i \in X}$ are computationally indistinguishable to C^n, i.e. the uniform distribution over ciphertext space. The consistency requirement is that a random ciphertext decrypts to "0" with overwhelming probability.

Membership encryption distills the right functionality we need for an encryption scheme in mq-RPMT protocol. It not only encompasses the constructions from randomized SKE and PKE in a unified manner, but also admits new construction from deterministic SKE, which enjoys compact ciphertext. As we elaborate in Section 4.3, this new construction helps to halve the communication complexity on the receiver side.

1.4 Related Work

We survey existing PSU protocols with security against semi-honest adversaries. Hereafter, unless otherwise declared, we calculate the efficiency by assuming a balanced setting, namely the sets of both sender and receiver are of size n.

Kissner and Song [KS05] proposed the first PSU protocol based on polynomial representations and additively homomorphic encryption (AHE). The polynomial representation of a set is to represent a set by a polynomial f, in which each set item is the root of the polynomial. The main observation of them is that when the set of two parties is represented by polynomials f and g, the root of fg is exactly the union items. Two parties compute the AHE of fg. Before decryption, they execute a reduction step to reduce the degree of roots. Then they decrypt the resulting polynomial and compute the roots to obtain the union. The communication and computation complexity of the protocol are both quadratic to the set size n, and the efficiency is very low due to the use of expensive AHE.

Frikken [Fri07] first uses polynomial representation to represent the receiver's set X as a polynomial f, then encrypt f via AHE, and let the receiver send the resulting polynomial encryption $\mathsf{Enc}(f)$ to the sender. Using additive homomorphic property, the sender computes the encryption of $(y\mathsf{Enc}(f(y)), \mathsf{Enc}(f(y)))$ for all $y \in Y$ and sends back to the receiver. The receiver decrypts these ciphertexts. As we can see, $y \in X$ if and only if $f(y) = 0$, receiver obtains two ciphertexts of 0, which contain no information about y. For $y \notin X, f(y) \neq 0$, the receiver decrypts $(y\mathsf{Enc}(f(y)), \mathsf{Enc}(f(y)))$ and computes $y := yf(y) \cdot (f(y))^{-1}$. The communication of this PSU protocol is linear with input size $O(n)$, however, the computation cost is expensive due to the multi-point evaluation on the encrypted polynomial, which is $O(n \log \log n)$.

Davidson and Cid [DC17] proposed a linear communication PSU based on Bloom Filter (BF) and AHE. The receiver first computes the BF of its input set X with λ hash functions and XORs the all-1 string with BF. Then he encrypts this reversed BF with an AHE and sends it to the sender. For each item y in the sender's set, the sender computes λ positions of BF with the same hash functions, and then adds the ciphertexts of corresponding positions. Let c denote the sum of these ciphertexts, the sender computes (c, yc) and sends it back to the receiver. The receiver decrypts these ciphertexts. Note that for $y \in X$, all the λ positions of BF is 0, the receiver obtains $(0, 0)$. For $y \notin X$, the ciphertext c is not an encryption of 0, the receiver could obtain the corresponding y. The communication of this PSU protocol is also linear with input size $O(n)$. However, the computation cost is $O(\lambda)$ public-key operations per item. The total computation is $O(\lambda n)$, which leads to low efficiency.

Garimella et al. [GMR$^+$21] recently proposed a new PSU protocol based on oblivious switching. The main construction of them is a permuted characteristic functionality $\mathcal{F}_{\mathsf{pc}}$. In this functionality, the sender inputs the set $X = \{x_1, \ldots, x_n\}$ and gets a permutation π over $[n]$ as the output. The receiver inputs the set Y and gets a vector $e \in \{0, 1\}^n$, where $e_i = 1$ if $x_{\pi(i)} \in Y$ and $e_i = 0$ otherwise. After that, both parties invoke n instances of OT to let the receiver obliviously retrieve items outside Y. Their core construction of $\mathcal{F}_{\mathsf{pc}}$ needs an oblivious switching network (OSN) subprotocol [MS13]. However, this OSN protocol also leads to a super-linear $O(n \log n)$ communication. In their construction, the receiver has to compute a degree-$3n$ interpolation polynomial. By using the hash to bin technology, the computational complexity is $O(n \log n)$.

Jia et al. [JSZ$^+$22] proposed two shuffle-based PSU protocols. The core primitive of their construction is also the oblivious switching network (OSN) subprotocol [MS13] (they called Permute + Share subprotocol), thus the performance of their protocols are similar to that of [GMR$^+$21].

Other PSU protocols focus on multi-party settings [KS05, HKK$^+$11, BA12, SCK12], malicious settings [Fri07, HN10, SCK12] and computation with untrusted third party's help [Bf12, CPPT14, SM18]. All of the above constructions rely heavily on expensive AHE or zero-knowledge proof, which are out of the scope of our consideration.

Table 1 provides an asymptotic comparison of our design with the previous PSU works. We note that although the complexity of our SKE-based scheme is also related to t, where t is the number of AND gates in an SKE decryption circuit, we emphasize that t is a constant which is independent of n, that is, t remains the same no matter how n changes. In this sense, the complexity of our SKE-based scheme is strictly linear in n, though in practice t is larger than $\log n$. We leave the construction of a linear SKE-based PSU with a concrete complexity smaller than $\log n$ to future work.

Protocol	Communication	Computation
[KS05]	$O(\kappa^3 n^2)$	$O(n^2)$ pub
[Fri07]	$O(\kappa n)$	$O(n \log\log n)$ pub
[DC17]	$O(\kappa\lambda n)$	$O(\lambda n)$ pub
[KRTW19]	$O(\kappa n \log n)$	$O(n \log n \log\log n)$ sym
[GMR$^+$21]	$O(\kappa n \log n)$	$O(n \log n)$ sym
[JSZ$^+$22]	$O(\kappa n \log n)$	$O(n \log n)$ sym
Our SKE-based	$O((\kappa + t)n)$	$O(tn)$ sym
Our PKE-based	$O(\kappa n)$	$O(n)$ pub

Table 1: Asymptotic communication and computation costs of two-party PSU protocols in the semi-honest setting.
Pub: public-key operations; sym: symmetric cryptographic operations. n is the size of the parties' input sets. κ and λ are computational and statistical security parameter respectively (typically $\kappa = 128$ and λ=40). t is the number of AND gates in an SKE decryption circuit. We ignore the pub-key cost of κ base OTs.

2 Preliminaries

2.1 Notation

We denote the parties as receiver $\mathcal{R}$ and sender $\mathcal{S}$, and their respective input sets as X and Y with $|X| = n_{\mathsf{x}}$ and $|Y| = n_{\mathsf{y}}$. In the balanced setting, we often just assume that $n = n_{\mathsf{x}} = n_{\mathsf{y}}$. We use κ and λ to denote the computational and statistical security parameters, respectively. We use $[n]$ to denote the set $\{1, 2, \ldots, n\}$. For a bit string v we let v_i denote the ith bit. We use $\mathbb{F}_{2^\sigma}$ to denote finite field composed of all σ-long bit strings. We say that a function f is negligible in κ if it vanishes faster than the inverse of any polynomial in κ, and write it as $f(\kappa) = \mathsf{negl}(\kappa)$. We use the abbreviation PPT to denote probabilistic polynomial-time. By $a \xleftarrow{\text{R}} A$, we denote that a is randomly selected from the set A, $a \leftarrow \mathsf{A}(x)$ denotes that a is the output of the randomized algorithm A on input x, and $a := b$ denotes that a is assigned by b.

2.2 Security Model

This work, similar to most protocols for private set operation, 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. This is in contrast to malicious adversaries which are able to deviate arbitrarily from the protocol. PSU protocols for the malicious setting exist, e.g., [KS05, Fri07, HN10, BA12, SCK12], but they are less efficient than protocols for the semi-honest setting.

Semi-honest security. We use the standard security definition for two-party computation [Gol04] in this work.

Definition 1. *Let* $\mathsf{view}_{\mathcal{S}}^{\Pi}(X,Y)$ *and* $\mathsf{view}_{\mathcal{R}}^{\Pi}(X,Y)$ *be the views of* $\mathcal{S}$ *and* $\mathcal{R}$ *in the protocol, and let* $\mathsf{output}(X,Y)$ *be the output of both parties in 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*,*

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

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

2.3 Encryption Schemes

Our construction requires some encryption schemes. We use the standard definition of symmetric-key encryption (SKE) and re-randomizable public-key encryption (ReRand-PKE) schemes. For our purpose, we require a case-tailored security notion called *single-message multi-ciphertext pseudorandomness*. We give these definitions in Appendix A.

2.4 Oblivious Transfer

Oblivious transfer (OT) [Rab05] is an important cryptographic primitive used in various multiparty computation protocols.

We define the functionality 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 $\mathcal{F}_{\mathsf{ot}}$

2.5 Oblivious Key-Value Stores

A key-value store [PRTY20, GPR+21] 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:*

- $\mathsf{Encode}_H(\{(x_1,y_1),\ldots,(x_n,y_n)\})$*: 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* $\perp$*).*
- $\mathsf{Decode}_H(D,x)$ *: 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 all distinct $\{x_1^1,\ldots,x_n^1\}$, if Encode_H does not output $\perp$ for $\{x_1^0,\ldots,x_n^0\}$ or $\{x_1^1,\ldots,x_n^1\}$, then 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 $\{D | y_i \leftarrow \mathcal{V}, i\in[n], \mathsf{Encode}_H((x_1^1,y_1),\ldots,(x_n^1,y_n))\}$.

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 addition, our application requires OKVS to meet the *Randomness* property defined below to argue the correctness of our scheme.

Randomness. For any $A = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ and $x^* \notin \{x_1, \ldots, x_n\}$, the output of $\mathsf{Decode}_H(D, x^*)$ is statistically indistinguishable to that of uniform distribution over $\mathcal{V}$, where $D \leftarrow \mathsf{Encode}_H(A)$.

The efficiency of an OKVS scheme can be measured by following three parameters:

- **Rate:** Let ratio n/m be the rate of key-value store, where m is the size of object D. Note that the optimal rate is 1.
- **Encoding complexity:** The computational cost of the Encode_H algorithm, as a function of the number n of key-value pairs.
- **Decoding complexity:** The computational cost of the Decode_H algorithm.

We investigated the existing schemes and found that the main candidates for OKVS are: Polynomial, Garbled Bloom Filter (GBF) [DCW13] and Garbled Cuckoo Table (GCT) [PRTY20, RS21, GPR+21] etc. We give the general introduction and detailed comparisons of above OKVS in Appendix B.1.

Before instantiation, 3H-GCT recently proposed by Garimella et al. [GPR+21] could be a good candidate, which has linear encoding complexity $O(n)$ and a rate of 0.81. However, we find that the original 3H-GCT did not meet the *Randomness* we defined before because it was set to 0 in some positions of D. To solve this problem, a natural idea is to set random values in these positions like [RS21] does. We call this modified 3H-GCT as 3H-GCT++ and give the formal description in Figure 3. We give a proof that our 3H-GCT++ satisfies obliviousness and randomness in Appendix B.2.

2.6 Private Set Union

PSU is a special case of secure two-party computation. The ideal functionality for PSU is given in Figure 4.

3 Multi-Query Reverse Private Membership Test

3.1 Definition

We propose mq-RPMT and give the formal definition of mq-RPMT functionality in Figure 5. For generality we set $|Y| = n_\mathsf{y}$ and $|X| = n_\mathsf{x}$ in our definition.

We define the vector oblivious decryption-then-matching $\mathcal{F}_\mathsf{vodm}$ corresponding to encryption scheme $\mathcal{E}$=(Setup,KeyGen,Enc,Dec) in Figure 6, as a component of mq-RPMT.

3.2 Framework of Multi-Query RPMT

Now we describe our framework of mq-RPMT protocol. As we said in Section 1.3, the cryptographic primitives we use are a single-message multi-ciphertext pseudorandomness encryption scheme $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$, an OKVS scheme $(\mathsf{Encode}_H, \mathsf{Decode}_H)$ and the $\mathcal{F}_\mathsf{vodm}$ functionality.

Let $Y = \{y_1, \ldots, y_{n_\mathsf{y}}\}$ and $X = \{x_1, \ldots, x_{n_\mathsf{x}}\}$ be the input of mq-RPMT sender $\mathcal{S}$ and receiver $\mathcal{R}$. First, the receiver $\mathcal{R}$ picks an indication string s[7]. Then $\mathcal{R}$ chooses a random key k used in encryption scheme $\mathcal{E}$ to encrypt s for n_x times, and obtains $(s_1, \ldots, s_{n_\mathsf{x}})$. Next, $\mathcal{R}$ computes an OKVS $D := \mathsf{Encode}_H((x_1, s_1), \ldots, (x_{n_\mathsf{x}}, s_{n_\mathsf{x}}))$ and sends D to $\mathcal{S}$. After receiving D, $\mathcal{S}$ computes $s_i^* = \mathsf{Decode}_H(D, y_i)$ for $i \in [n_\mathsf{y}]$. Now $\mathcal{S}$ and $\mathcal{R}$ invoke the VODM functionality $\mathcal{F}_\mathsf{vodm}$. $\mathcal{S}$ acts as sender with input $S = \{s_1^*, \ldots, s_{n_\mathsf{y}}^*\}$ and $\mathcal{R}$ acts as receiver with input (k, s). As a result, $\mathcal{S}$ receives nothing and $\mathcal{R}$ receives $b \in \{0,1\}^{n_\mathsf{y}}$, satisfying $b_i = 1$ if and only if s_i^* decrypts to s. Now, we give our framework of mq-RPMT protocol in Figure 7.

Correctness. For all $i \in [n_\mathsf{y}]$, if $y_i \in X$, there is an $x_j \in X, j \in [n_\mathsf{x}]$ s.t. $y_i = x_j$. In this case, $s_i^* = \mathsf{Decode}_H(D, h(x_j)) = s_j$. Since $s_j = \mathsf{Enc}(k, s)$, we have $\mathsf{Dec}(k, s_j) = s$, which means $b_i = 1$. In the case $y_i \notin X$, if hash functions collide, that is, $h(y_i) = h(x)$ for some $y_i \notin X$, the correctness will be violated. By setting $\sigma = \lambda + \log n_\mathsf{x} n_\mathsf{y}$, a union bound shows probability of collision is negligible $2^{-\lambda}$. When no collision occurs, from the randomness of OKVS, $s_i^* = \mathsf{Decode}_H(D, h(y_i))$ is a random ciphertext, resulting in s_i^* is not the encryption of s with overwhelming probability. The union bound guarantees that for all $y_i \notin X$, the probability that there exists an s_i^* s.t. $\mathsf{Dec}(k, s_i^*) = s$ is negligible.

[7] In fact, our indication string s could be any fixed value, e.g. $s = 0$, while s in KRTW must be selected randomly.

Parameters:

- Computational security parameter κ and statistical security parameter λ.
- Input length n.
- A finite group $\mathbb{G}$.
- Random fuctions $h_1, h_2, h_3 : \{0,1\}^* \to [m']$ and $r : \{0,1\}^* \to \{0,1\}^{d+\lambda}$.
- Parameters $m' = 1.3n$ and $d = 0.5 \log n$, as shown in [GPR+21], where d upper bound the size of 2-core of a (m', n)-Cuckoo graph.
- Output length $m = m' + d + \lambda$.

<u>$\mathsf{Encode}_H(\{(x_1, y_1), \ldots, (x_n, y_n)\})$:</u>

1. Define $l(x) \in \{0,1\}^{m'}$ to be all zeroes except 1s at positions $h_1(x)$, $h_2(x), h_3(x)$. Here we assume the weight of $l(x)$ is 3. Let $\mathrm{row}(x) := l(x)||r(x)$,
$$M^{(0)} = \begin{bmatrix} l(x_1) \\ \vdots \\ l(x_n) \end{bmatrix} \in \{0,1\}^{n \times m'}, M^{(1)} = \begin{bmatrix} r(x_1) \\ \vdots \\ r(x_n) \end{bmatrix} \in \{0,1\}^{n \times (d+\lambda)}$$
and let
$$M = M^{(0)}||M^{(1)} = \begin{bmatrix} \mathrm{row}(x_1) \\ \vdots \\ \mathrm{row}(x_n) \end{bmatrix} \in \{0,1\}^{n \times m}.$$
2. Initialize empty vectors $D_L \in \mathbb{G}^{m'}$ and $D_R \in \mathbb{G}^{d+\lambda}$, let $D = D_L||D_R$.
3. Initialize stack P.
4. While there is a node $j \in [m']$ such that the set $\{x_i \notin P | j \in \{h_1(x_i), h_2(x_i), h_3(x_i)\}\}$ is a singleton: Let x_i be the element of that singleton, and push x_i onto P.
5. Let $S = \{x_i | x_i \notin P\}$, and let $R \subset [n]$ index the rows of M in S, i.e. $R = \{i | M^{(0)}_{i,h_1(x_i)} = M^{(0)}_{i,h_2(x_i)} = M^{(0)}_{i,h_3(x_i)} = 1 \wedge x_i \in S\}$. Let $\tilde{d} := |R|$ and abort if $\tilde{d} > d$.
6. Let $\tilde{M}^{(1)} \in \{0,1\}^{\tilde{d} \times (d+\lambda)}$ be the submatrix of $M^{(1)}$ obtained by taking the row indexed by R. Abort if $\tilde{M}^{(1)}$ does not contain an invertible $\tilde{d} \times \tilde{d}$ matrix. Otherwise let $\tilde{M}^*$ be one such matrix and $C \subset [d+\lambda]$ index the corresponding columns of $\tilde{M}^{(1)}$.
7. Let $C' := \{j | i \in R, M^{(0)}_{i,j} = 1\} \cup ([d+\lambda] \setminus C + m')$ and for $i \in C'$ assign $D_i \leftarrow \mathbb{G}$. For $i \in R$, define $y'_i := y_i - (MD^T)_i$ where D_i is assumed to be zero if unssigned.
8. Using Gaussian elimination solve the system $\tilde{M}^*(D_{m'+C_1}, \ldots, D_{m'+C_{\tilde{d}}})^T = (y'_{R_1}, \ldots, y'_{R_{\tilde{d}}})^T$.
9. While P not empty:
   (a) pop x_i from P.
   (b) D_L is undefined in at least one of the positions $h_1(x_i), h_2(x_i), h_3(x_i)$. Set the undefined position(s) so that $\langle \mathrm{row}(x_i), D \rangle = y_i$.
10. Set any empty position in D with a random value from $\mathbb{G}$.
11. Output D.

<u>$\mathsf{Decode}_H(D, x)$:</u>

1. Return $\langle \mathrm{row}(x), D \rangle$.

Fig. 3: 3H-GCT++ algorithm

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

- Wait for input $X = \{x_1, \ldots, x_{n_{\mathsf{x}}}\} \subset \{0,1\}^*$ from the receiver $\mathcal{R}$.
- Wait for input $Y = \{y_1, \ldots, y_{n_{\mathsf{y}}}\} \subset \{0,1\}^*$ from the sender $\mathcal{S}$.
- Give output $X \cup Y$ to the receiver $\mathcal{R}$.

Fig. 4: Private Set Union Functionality $\mathcal{F}_{\mathsf{psu}}$

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

- Wait for input $Y = \{y_1, \ldots, y_{n_{\mathsf{y}}}\} \subset \{0,1\}^*$ from the sender $\mathcal{S}$.
- Wait for input $X = \{x_1, \ldots, x_{n_{\mathsf{x}}}\} \subset \{0,1\}^*$ from the receiver $\mathcal{R}$.
- Set $b_i = 1$ if and only if $y_i \in X$ and $b_i = 0$ otherwise for $i \in [n_{\mathsf{y}}]$. Give output $b \in \{0,1\}^{n_{\mathsf{y}}}$ to the receiver $\mathcal{R}$.

Fig. 5: Multi-Query Reverse Private Membership Test Functionality $\mathcal{F}_{\mathsf{mq\text{-}rpmt}}$

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, set sizes n, an encryption scheme $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.
Functionality:

- Wait for input k and s from the receiver $\mathcal{R}$.
- Wait for input $\{s_1^*, \ldots, s_n^*\} \subset \{0,1\}^*$ from the sender $\mathcal{S}$.
- For $i \in [n]$:
  Compute $s_i' = \mathsf{Dec}(k, s_i^*)$. If $s_i' = s$, let $b_i = 1$, otherwise $b_i = 0$.
- Give output $b \in \{0,1\}^n$ to the receiver $\mathcal{R}$.

Fig. 6: Vector Oblivious Decryption-then-Matching Functionality $\mathcal{F}_{\mathsf{vodm}}$

Parameters:

- Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$.
- A single-message multi-ciphertext pseudorandomness encryption scheme $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.
- Ideal $\mathcal{F}_{\mathsf{vodm}}$ primitives specified in Figure 6.
- An OKVS scheme $(\mathsf{Encode}_H, \mathsf{Decode}_H)$.
- A collision-resistant hash function $h(x) : \{0,1\}^* \to \{0,1\}^\sigma$.

Input of $\mathcal{S}$: $Y = \{y_1, \ldots, y_{n_{\mathsf{y}}}\} \subset \{0,1\}^*$
Input of $\mathcal{R}$: $X = \{x_1, \ldots, x_{n_{\mathsf{x}}}\} \subset \{0,1\}^*$
Protocol:

1. $\mathcal{R}$ selects a random *indication string* $s \in \mathbb{F}_{2^\sigma}$. $\mathcal{R}$ also runs $pp \leftarrow \mathsf{Setup}(1^\kappa)$ and $\mathsf{KeyGen}(pp)$ to obtain a key k (public or symmetric key depend on concrete scheme). Then, $\mathcal{R}$ runs $\mathsf{Enc}(k, s)$ for n_{x} times to obtain $(s_1, \ldots, s_{n_{\mathsf{x}}})$.
2. $\mathcal{R}$ computes an OKVS
   $D := \mathsf{Encode}_H((h(x_1), s_1), \ldots, (h(x_{n_{\mathsf{x}}}), s_{n_{\mathsf{x}}}))$.
3. $\mathcal{R}$ sends D to $\mathcal{S}$.
4. $\mathcal{S}$ computes $s_i^* := \mathsf{Decode}_H(D, h(y_i))$ for $i \in [n_{\mathsf{y}}]$.
5. $\mathcal{S}$ and $\mathcal{R}$ invoke the VODM functionality $\mathcal{F}_{\mathsf{vodm}}$. $\mathcal{S}$ acts as sender with input $S = \{s_1^*, \ldots, s_{n_{\mathsf{y}}}^*\}$ and $\mathcal{R}$ acts as receiver with input k, s. As a result, $\mathcal{S}$ receives nothing and $\mathcal{R}$ receives $b \in \{0,1\}^{n_{\mathsf{y}}}$.

Fig. 7: General Construction of mq-RPMT Protocol $\Pi_{\mathsf{mq\text{-}rpmt}}$

We now prove the security properties of our mq-RPMT.

Theorem 1. *Assume the encryption scheme* $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *satisfies single-message multi-ciphertext pseudorandomness. The protocol in Figure 7 securely computes* $\mathcal{F}_{\mathsf{mq\text{-}rpmt}}$ *against semi-honest adversaries in the* $\mathcal{F}_{\mathsf{vodm}}$*-hybrid model.*

Proof Due to space limitation, we only sketch here the simulators for the two cases of corrupt $\mathcal{S}$ and corrupt $\mathcal{R}$, the full proof (via hybrid arguments) is deferred to Appendix C.

Corrupt sender: To simulate OKVS in Step 3, the simulator computes a random OKVS D by selecting n_{x} random key-value pairs. Then, the simulator sets $s_i^* := \mathsf{Decode}_H(D, h(y_i))$ and invokes underlying VODM simulator with inputs $(s_1^*, \ldots, s_{n_{\mathsf{y}}}^*)$.

Briefly, this simulation is indistinguishable for the following reasons: the single-message multi-ciphertext pseudorandomness of the encryption ensures that value (ciphertext) is indistinguishable from random, and then by the obliviousness of OKVS, D is distributed uniformly.

Corrupt receiver: The simulator for a corrupt $\mathcal{R}$ first obtains b from the ideal mq-RPMT functionality. The only message that needs to be simulated is the VODM functionality in Step 5. The simulator just executes Step 1 honestly and invokes the underlying VODM simulator with inputs (k, s, b).

□

4 Generic Constructions of Multi-Query RPMT

In this section, we give two generic constructions of mq-RPMT protocol. In the first construction, we use SKE as the encryption scheme and generic 2PC to implement VODM. The advantage is that this scheme only uses OT and symmetric operations. In the second construction, we use PKE and a re-randomization method to implement the encryption scheme and a leaky version of VODM respectively, which leads to a leaky version of mq-RPMT. However, as observed by KRTW, this leaky version can still be used to construct a secure PSU. Both schemes achieve linear computation and communication complexity.

4.1 Construction from SKE and 2PC

As we noted before, a single-message multi-ciphertext pseudorandom SKE and 2PC are sufficient for constructing mq-RPMT. The correctness and security can be directly derived from the general construction in Section 3.2. It is straightforward to show that PRF-based SKE satisfies the single-message multi-ciphertext pseudorandomness property. We give proof in the Appendix D for completeness.

We use the general 2PC as the implementation of VODM. Formally,

Theorem 2. *Taking the PRF-based SKE as the encryption scheme in Figure 7. Assuming that the 2PC implementing VODM is semi-honest secure, then the protocol in Figure 7 securely computes $\mathcal{F}_{\mathsf{mq\text{-}rpmt}}$ against semi-honest adversaries.*

This theorem immediately follows from Lemma 1 and Theorem 1.

4.2 Construction from Re-randomizable PKE

Now we consider a specialized way to construct $\mathcal{F}_{\mathsf{vodm}}$. Our main idea is that since the receiver cannot know the randomness used in each ciphertext, as long as the encryption scheme satisfies the property of rerandomization, the sender can re-randomize all ciphertexts and send the new ciphertexts to the receiver so that the receiver can not obtain additional information by comparing randomness. Note that another problem arises here. The property of re-randomization can only guarantee that for $y \in X$, the receiver is not allowed to learn which one is the sender's element. For $y \notin X$, the ciphertext s_i^* obtained by the sender is related to y, so the plaintext obtained by the receiver is also related to y, which will reveal the information of y. However, as observed by KRTW, in the case of $y \notin X$, we want (in the overall PSU protocol) the receiver to learn y anyway. Fully secure mq-RPMT is actually overkill for PSU, a relaxed version suffices. We define the leaky VODM functionality in Figure 8.

Since the SKE scheme is hard to re-randomize, we consider the use of public-key encryption (PKE) which is easier to re-randomize. We describe our PKE-based leaky VODM protocol in Figure 9.

We now state and prove the security of the above leaky VODM protocol.

Theorem 3. *Assume the security of the ReRand-PKE scheme. The protocol in Figure 9 securely computes $\mathcal{F}_{\mathsf{lvodm}}$ against semi-honest adversaries.*

Proof Because the sender does not receive messages in the protocol, we just need to simulate the view of the receiver. We exhibit simulator $\mathsf{Sim}_{\mathcal{R}}$ for simulating corrupt $\mathcal{R}$.

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, set sizes n, an encryption scheme $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.
Functionality:

- Wait for input k and s from the receiver $\mathcal{R}$.
- Wait for input $\{s_1^*, \ldots, s_n^*\} \subset \{0,1\}^*$ from the sender $\mathcal{S}$.
- For $i \in [n]$:
  Compute $s_i' = \mathsf{Dec}(k, s_i^*)$, if $s_i' = s$, let $b_i = 1$ otherwise $b_i = 0$.
- Give output $b \in \{0,1\}^n$ and $\{s_i' | b_i = 0\}$ to the receiver $\mathcal{R}$.

Fig. 8: Leaky VODM Functionality $\mathcal{F}_{\mathsf{lvodm}}$

Parameters:

- Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$.
- A re-randomizable PKE scheme
  $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{ReRand})$.

Input of $\mathcal{S}$: $(pk, S^* = \{s_1^*, \ldots, s_n^*\})$
Input of $\mathcal{R}$: $((pk, sk), s)$
Protocol:

1. $\mathcal{S}$ selects random $r_1', \ldots, r_n'$ and computes $\bar{s_i} := \mathsf{ReRand}(pk, s_i^*; r_i')$ for $i \in [n]$.
2. $\mathcal{S}$ sends $\bar{s_1}, \ldots, \bar{s_n}$ to $\mathcal{R}$.
3. $\mathcal{R}$ sets $b_i = 1$ if and only if $\mathsf{Dec}(sk, \bar{s_i}) = s$ for $i \in [n]$.

Fig. 9: PKE-based Leaky VODM Protocol Π_{lvodm}

Corrupt receiver: $\mathsf{Sim}_{\mathcal{R}}(pk, sk, s, b, \{s_i' | b_i = 0\})$ simulates the view of corrupt semi-honest receiver. Note that the only messages that need to be simulated by the simulator are ciphertexts $\{\bar{s_i}\}_{i \in [n]}$.

$\mathsf{Sim}_{\mathcal{R}}$ computes $\bar{s_i} := \mathsf{Enc}(pk, s; r_i)$ if $b_i = 1$ and $\bar{s_i} := \mathsf{Enc}(pk, s_i'; r_i)$ if $b_i = 0$ for $i \in [n]$. $\mathsf{Sim}_{\mathcal{R}}$ appends $\{\bar{s_i}\}_{i \in [n]}$ to the view.

The indistinguishability of ReRand-PKE scheme guarantees the view output by $\mathsf{Sim}_{\mathcal{R}}$ is indistinguishable from the real one. □

Note that the mq-RPMT constructed with the above leaky VODM is also a leaky version. We don't give a specific description of this leaky mq-RPMT. Instead, we use leaky VODM to construct PSU protocol directly and prove its security in Appendix E.

4.3 Unification with Membership Encryption

We have presented two generic constructions of mq-RPMT protocols from probabilistic SKE and probabilistic PKE respectively. It is intriguing to study if there is a unified way to encompass the two different constructions.

We retrospect the high level idea underlying our mq-RPMT protocol. If privacy is not a concern, reverse membership test can be simply done by having the receiver first create a membership relation R for his set Y, namely $\mathsf{R}(y) = 1$ iff $y \in Y$, then having the sender send his elements to the receiver in clear. To make the reverse membership test private, the receiver can "encrypt" his membership relation and send the "encoding" of resulting ciphertexts to the sender. After receiving the "encoding", the sender is able to retrieve the membership encryptions corresponding to his elements. In the sequel, the receiver can fulfill the reverse private membership test by decrypting the ciphertexts in an oblivious manner.

Based on the above discussion, we realize that the right encryption scheme needed in our mq-RPMT protocol is an abstract new notion called *membership encryption (ME)*. Roughly speaking, ME for set X encrypts an element x into a ciphertext, which decrypts to "1" if $x \in X$. We formalize the syntax and security notion of ME in the private-key setting as below.

Definition 3 (Membership Encryption). *Membership encryption for set X consists of four polynomial time algorithms satisfying the following properties.*

- $\mathsf{Setup}(1^\kappa)$*: on input a security parameter κ, outputs public parameters pp, which include the ciphertext space C.*
- $\mathsf{KeyGen}(pp, X)$*: on input public parameters pp and $X \subseteq \{0,1\}^*$, outputs a key k.*
- $\mathsf{Enc}(k, x)$*: on input a key k and an element $x \in X$, outputs a ciphertext $c \in C$. For uttermost generality, the behavior of* Enc *on $x \notin X$ is unspecified. Looking ahead, such treatment suffices for the construction of mq-RPMT protocol.*
- $\mathsf{Dec}(k, c)$*: on input a key k and a ciphertext $c \in C$, outputs "1" indicating c is an encryption of an element x in X and "0" if not.*

Correctness. For any $x \in X$, $\forall k \leftarrow \mathsf{KeyGen}(pp, X)$, $\mathsf{Dec}(k, c = \mathsf{Enc}(k, x)) = 1$.
Consistency. For any $x \notin X$, $\Pr[\mathsf{Dec}(k, c) = 0] = 1 - \epsilon(\kappa)$, where $pp \leftarrow \mathsf{Setup}(1^\kappa)$, $k \leftarrow \mathsf{KeyGen}(pp, X)$, $c \xleftarrow{\text{R}} C$. Here, ϵ is the consistency error, which must be negligible in κ.
Multi-element pseudorandomness. For any n distinct elements $x_1, \ldots, x_n \in X$, $\{\mathsf{Enc}(k, x_i)\}_{i \in [n]} \approx_c U_{C^n}$.

The ME notion naturally extends to the public-key setting by letting the KeyGen algorithm generate a keypair (pk, sk), in which pk is used to encrypt and sk is used to decrypt. We omit the details for its straightforwardness.

We then study the generic construction of ME. Note that the essence of ME is to encrypt element's membership relation, rather than the element itself. The membership relation can be created by establishing a mapping H from elements to the set under test. Basically, there are two extreme cases of mapping. The first is to select a single indication string s as the characteristic of the set, then map all elements to s, i.e., $\mathsf{H} : x_i \to s$, which we refer to as *lossy mapping*. The second is to select n indication strings s_i as the characteristic of the set, then map elements to distinct indication strings, i.e., $\mathsf{H} : x_i \to s_i$, which we refer to as *injective mapping*. With the above understanding in head, we present various constructions of ME by mixing encryption schemes and membership mapping.

ME from probabilistic SKE and lossy mapping. The construction is as below.

- $\mathsf{Setup}(1^\kappa)$: runs SKE.$\mathsf{Setup}(1^\kappa)$ to generate pp.
- $\mathsf{KeyGen}(pp, X)$: runs SKE.$\mathsf{KeyGen}(pp)$ to sample k_{ske}, picks a random element $s \in M$, where M is the message space of SKE, sets H be a mapping that maps all elements in X to s, outputs $k = (k_{\text{ske}}, \mathsf{H})$
- $\mathsf{Enc}(k, x)$: parses $k = (k_{\text{ske}}, \mathsf{H})$, outputs $c \leftarrow$ SKE.$\mathsf{Enc}(k_{\text{ske}}, \mathsf{H}(x))$.
- $\mathsf{Dec}(k, c)$: parses $k = (k_{\text{ske}}, \mathsf{H})$, outputs "1" iff SKE.$\mathsf{Dec}(k_{\text{ske}}, c) = s$.

ME from probabilistic PKE and lossy mapping. The construction is as below.

- $\mathsf{Setup}(1^\kappa)$: runs PKE.$\mathsf{Setup}(1^\kappa)$ to generate pp.
- $\mathsf{KeyGen}(pp, X)$: runs PKE.$\mathsf{KeyGen}(pp)$ to generate $(pk_{\text{pke}}, sk_{\text{ske}})$, picks a random element $s \in M$, where M is the message space of PKE, sets H be a mapping that maps all elements in X to s, outputs $pk = pk_{\text{pke}}$ and $sk = (sk_{\text{pke}}, \mathsf{H})$
- $\mathsf{Enc}(pk, x)$: parses $pk = pk_{\text{pke}}$, outputs $c \leftarrow$ PKE.$\mathsf{Enc}(pk_{\text{pke}}, \mathsf{H}(x))$.
- $\mathsf{Dec}(sk, c)$: parses $sk = (sk_{\text{pke}}, \mathsf{H})$, outputs '1' iff PKE.$\mathsf{Dec}(sk_{\text{pke}}, c) = s$.

Theorem 4. *If SKE (resp. PKE) satisfies single-message multi-ciphertext pseudorandomness, then the above ME construction satisfies multi-element pseudorandomness with consistency error $1/|M|$.*

The above ME constructions are exactly the backbones of our generic constructions of mq-RPMT protocol presented in Section 4.1 and 4.2. Since ME requires multi-element pseudorandomness, the use of lossy mapping inherently stipulates that the accompanying encryption schemes are probabilistic. Therefore, in this case the ciphertext expansion is unavoidable. For example, in PRF-based probabilistic SKE, the length of ciphertext is twice that of plaintext. In the design of our mq-RPMT protocol, the value in OKVS is exactly ciphertext. As a consequence, ciphertext expansion incurs overhead to the size of OKVS and thus also the communication cost on the receiver side. For this reason, reducing the ciphertext expansion factor will immediately improve the performance of the overall mq-RPMT protocol.

An important observation is that if we switch to injective mapping, then ME can be built from deterministic encryption schemes satisfying *multi-message multi-ciphertext pseudorandomness*. The constructions are similar as above except the decryption algorithm outputs '1' iff the decryption result falls into the prior-fixed indication string set $S = \{s_i\}_{i\in[n]}$. In instantiation, we take $\mathsf{H} : x_i \to i$ as the membership mapping, which renders efficient membership decryption by testing whether the decryption is less than n.

Formally, we have the following theorem:

Theorem 5. *If SKE (resp. PKE) satisfies multi-message multi-ciphertext pseudorandomness, then the ME construction satisfies multi-element pseudorandomness with consistency error $n/|M|$.*

If we instantiate the ME from the PRP-based deterministic SKE and injective mapping, the ciphertext expansion factor is optimal. Therefore, a drop-in replacement to the ME from PRF-based probabilistic SKE and lossy mapping will reduce the size of OKVS in the mq-RPMT protocol by half.

Due to space constraints, we put the description that how to construct mq-RPMT using the language of ME in the Appendix F.

5 Our PSU Protocol

In this section, we describe our PSU construction achieving linear complexity and prove its semi-honest security.

5.1 Generic Construction of PSU Protocols

With mq-RPMT and OT, we can simply combine them to construct a PSU protocol. We give the formal description in Figure 10.

Parameters:

- Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$.
- Ideal $\mathcal{F}_{\mathsf{mq\text{-}rpmt}}$ and $\mathcal{F}_{\mathsf{ot}}$ primitives specified in Figure 5, and Figure 2, respectively.

Input of $\mathcal{S}$: $Y = \{y_1, \dots, y_{n_\mathsf{y}}\} \subset \{0,1\}^*$
Input of $\mathcal{R}$: $X = \{x_1, \dots, x_{n_\mathsf{x}}\} \subset \{0,1\}^*$
Protocol:

1. $\mathcal{S}$ and $\mathcal{R}$ invoke the mq-RPMT functionality $\mathcal{F}_{\mathsf{mq\text{-}rpmt}}$ first. The sender $\mathcal{S}$ acts as the sender in mq-RPMT with input Y and receives nothing. The receiver $\mathcal{R}$ acts as the receiver in mq-RPMT with input X and receives $b \in \{0,1\}^{n_\mathsf{y}}$.
2. $\mathcal{R}$ initialize set $Z := \{\}$.
3. For $i \in [n_\mathsf{y}]$:
   (a) $\mathcal{S}$ and $\mathcal{R}$ invoke the OT functionality $\mathcal{F}_{\mathsf{ot}}$.
   (b) $\mathcal{S}$ acts as sender with input $(y_i, \perp)$.
   (c) $\mathcal{R}$ acts as receiver with input b_i.
   (d) $\mathcal{R}$ receives z_i from OT and sets $Z = Z \cup \{z_i\}$.
4. $\mathcal{R}$ outputs $X \cup Z$.

Fig. 10: Private Set Union Protocol Π_{psu}

We now state and prove the security properties of the above PSU protocol.

Theorem 6. *The protocol in Figure 10 securely computes $\mathcal{F}_{\mathsf{psu}}$ against semi-honest adversaries in the $(\mathcal{F}_{\mathsf{mq\text{-}rpmt}}, \mathcal{F}_{\mathsf{ot}})$-hybrid model.*

Proof We exhibit simulators $\mathsf{Sim}_{\mathcal{R}}$ and $\mathsf{Sim}_{\mathcal{S}}$ for simulating corrupt $\mathcal{R}$ and $\mathcal{S}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

Corrupt Sender: $\mathsf{Sim}_{\mathcal{S}}(Y = \{y_1, \ldots, y_{n_\mathsf{y}}\})$ simulates the view of corrupt semi-honest sender. It executes as follows:

1. $\mathsf{Sim}_{\mathcal{S}}$ invokes mq-RPMT simulator $\mathsf{Sim}^{S}_{\mathsf{mq\text{-}rpmt}}(Y)$ and appends the output to the view.
2. For $i \in [n_\mathsf{y}]$, $\mathsf{Sim}_{\mathcal{S}}$ invokes OT simulator $\mathsf{Sim}^{S}_{\mathsf{ot}}(y_i, \perp)$ and appends the output to the view.

Now we argue that the view output by $\mathsf{Sim}_{\mathcal{S}}$ is indistinguishable from the real one. This is obtained by the underlying simulators' indistinguishability directly.
Corrupt Receiver: $\mathsf{Sim}_{\mathcal{R}}(X = \{x_1, \ldots, x_{n_\mathsf{x}}\}, X \cup Y)$ simulates the view of corrupt semi-honest receiver. It executes as follows:

1. $\mathsf{Sim}_{\mathcal{R}}$ defines the set $Z := X \cup Y \setminus X$, i.e. the set of elements that Y "brings to the union". Next, it uses $\perp$ to pad Z to n_y elements and permutates these elements randomly. Let $Z = \{z_1, \ldots, z_{n_\mathsf{y}}\}$.
2. $\mathsf{Sim}_{\mathcal{R}}$ sets $b_i = 1$ if and only if $z_i \in X$ for $i \in [n_\mathsf{y}]$. Then, it invokes mq-RPMT simulator $\mathsf{Sim}^{R}_{\mathsf{mq\text{-}rpmt}}(X, b)$ and appends the output to the view.
3. For $i \in [n_\mathsf{y}]$, $\mathsf{Sim}_{\mathcal{R}}$ invokes OT simulator $\mathsf{Sim}^{R}_{\mathsf{ot}}(b_i, z_i)$ and appends the output to the view.

Now we argue that the view output by $\mathsf{Sim}_{\mathcal{R}}$ is indistinguishable from the real one. In the simulation, the way $\mathcal{R}$ obtains the elements in $Z = X \setminus Y$ is identical to the real execution. By the underlying simulators' indistinguishability, the simulated view is computationally indistinguishable from the real. □

5.2 Instantiation of PSU

For our SKE-based construction, we can use a PRP as we mentioned in Section 4.3 to instantiate SKE, which can achieve an optimal ciphertext expansion factor. Since we need to perform the 2PC decryption computation, we use the LowMC [ARS+15] as our PRP instantiation to minimize the circuit size. As for generic 2PC, there are two classical methods, e.g. garbled circuit [Yao86] or GMW [GMW87]. The former has a constant number of rounds, while the latter has a lower communication. Since the communication has a greater impact on our scheme, we consider instantiating 2PC by GMW.

For our PKE-based construction, we use the well-known ECC ElGamal [Gam85] scheme as our ReRand-PKE.

5.3 Communication Cost

Now we analyze the communication cost of our two PSU constructions. For the SKE-based construction, we use our ME optimization in Section 4.3.

Let's first analyze the size of decryption circuit in our SKE-based construction: the circuit needs to compute decryption of every $\{s_i^*\}_{i \in [n_\mathsf{y}]}$ and compare the result with n_x. If $\mathsf{Dec}(k, s_i^*) < n_\mathsf{x}$, it sets $b_i = 1$ and $b_i = 0$ otherwise. The total number of decryption computations is n_y. To compare whether a σ long string is less than n_x, we only need to compute whether the OR of its first $\sigma - \log n_\mathsf{x}$ bits are 1, which requires $\sigma - \log n_\mathsf{x} - 1$ AND gates (since $a \vee b = \bar{a} \wedge \bar{b}$). The total number of AND gates is $n_\mathsf{y}(t + \sigma - \log n_\mathsf{x}) = O(tn_\mathsf{y})$, where t is the number of AND gates in a PRP decryption circuit.

Now we are ready to calculate the communication of PSU protocol. Note that the communication of our protocol consists of OKVS, VODM protocol and OT protocol. We analyze their complexity respectively. We use the symbol Φ to represent the communication complexity, and its subscripts represent different components.

– OKVS in both constructions: as we showed in Section 2.5, we use 3H-GCT++ as our OKVS scheme:

$$\Phi_{\mathsf{okvs}}(n_\mathsf{x}) = (1.3n_\mathsf{x} + d + \lambda)|c|$$

, where $|c|$ is the size of ciphertext, $|c| = \lambda + \log n_\mathsf{x} n_\mathsf{y}$ and 4κ for SKE-based and PKE-based scheme respectively.

– Oblivious decryption:
  - In SKE-based construction: we use $\Phi^{\mathsf{ske}}_{\mathsf{vod}}(n_{\mathsf{y}}, n_{\mathsf{x}})$ to denote the communication of computing oblivious decryption circuit. As we said in Section 5.2, we use GMW as our 2PC instantiation, the communication consists of *input sharing*, *multiplication gate computation* and *output reconstruction*. In the input sharing phase, the communication is $\kappa + n_{\mathsf{y}}\sigma$ bits, and in the output reconstruction phase, it is n_{y} bits. Using Beaver triple [Bea91], $4n_{\mathsf{y}}(t + \sigma - \log n_{\mathsf{x}})$ bits are needed in multiplication phase. So we have $\Phi^{\mathsf{ske}}_{\mathsf{vod}}(n_{\mathsf{y}}, n_{\mathsf{x}}) = \kappa + n_{\mathsf{y}}\sigma + 4n_{\mathsf{y}}(t + \sigma - \log n_{\mathsf{x}}) + n_{\mathsf{y}}$
  - In PKE based construction: the communication of leaky VODM functionality, denoted by $\Phi^{\mathsf{pke}}_{\mathsf{lvodm}}(n_{\mathsf{y}}, n_{\mathsf{x}}) = 4n_{\mathsf{y}}\kappa$

– OT in both constructions:

$$\Phi_{\mathsf{ot}}(n_{\mathsf{y}}) = n_{\mathsf{y}}(\kappa + \sigma).$$

Let $\Phi^{\mathsf{ske}}_{\mathsf{psu}}(n_{\mathsf{y}}, n_{\mathsf{x}})$ denote communication of SKE-based construction and let $\Phi^{\mathsf{pke}}_{\mathsf{psu}}(n_{\mathsf{y}}, n_{\mathsf{x}})$ denote communication of PKE-based construction. The overall communication cost of our PSU protocol is:

$$\Phi^{\mathsf{ske}}_{\mathsf{psu}}(n_{\mathsf{y}}, n_{\mathsf{x}}) = \Phi_{\mathsf{okvs}}(n_{\mathsf{x}}) + \Phi^{\mathsf{ske}}_{\mathsf{vod}}(n_{\mathsf{y}}, n_{\mathsf{x}}) + \Phi_{\mathsf{ot}}(n_{\mathsf{y}})$$

$$\Phi^{\mathsf{pke}}_{\mathsf{psu}}(n_{\mathsf{y}}, n_{\mathsf{x}}) = \Phi_{\mathsf{okvs}}(n_{\mathsf{x}}) + \Phi^{\mathsf{pke}}_{\mathsf{lvodm}}(n_{\mathsf{y}}, n_{\mathsf{x}}) + \Phi_{\mathsf{ot}}(n_{\mathsf{y}})$$

5.4 Discussion: Difference between PSI and PSU

Although PSI and PSU are quite similar, as discussed in [KRTW19], the techniques they use are different, and building PSU is more challenging than building PSI.

Since the output of PSI is the elements of the receiver's own set, it is only necessary to test whether each element belongs to the sender's set (i.e., PMT), and the difficulty of PSU is how to retrieve the elements outside the intersection (i.e., RPMT + OT) without disclosing the intersection. In PSI, PMT can be easily obtained by OPRF: the sender obtains a PRF key k while the receiver obtains $F_k(y)$ on his input y, then the sender computes and sends $\{F_k(x)\}_{x \in X}$ to the receiver. The receiver tests whether $F_k(y) \in \{F_k(X)\}_{x \in X}$ to determine whether $y \in X$. As a result, OPRF is enough for PSI, and all the state-of-the-art PSI protocols [KKRT16, CM20, RS21] follow this paradigm and mainly focus on designing efficient OPRF protocols.

However, the conversion from PMT to RPMT is not trivial, as discussed in [KRTW19], this seemingly simple functionality adjustment (PMT $\rightarrow$ RPMT) doesn't seem to be fixable by a small tweak of PMT. Although OPRF is enough for PSI, this is not the case for PSU. In the state-of-the-art PSU [JSZ+22, GMR+21], OPRF is only one component, and the design of PSU protocol usually requires the use of a variety of different components, e.g., oblivious switch network functionality, and combine them in a clever method.

5.5 Discussion: the Relationship with Existing PSI/PSU-Related Primitives

Here we also discuss the relationship with existing PSI/PSU-related primitives.

OKVS. Garimella et al. [GPR+21] proposed the notion of Oblivious Key-Value Store (OKVS), which is useful in both PSI and PSU. The OKVS is a *data structure* in which a sender has a set of key-value mapping ($\{x_i, y_i\}$) with (pseudo)random y_i's, and she wishes to hand that mapping over to a receiver, allowing the receiver to evaluate the mapping on any input but without revealing the keys x_i. Correctness of the data structure must ensure that if the other party evaluates the OKVS on some $q = x_j$ then the result is y_j. Obliviousness here is that the receiver cannot tell what keys x_i's are encoded from a given OKVS. The most compact OKVS that one can think of is a polynomial. The recent excellent works on OKVS [PRTY20, GPR+21] make it very efficient to encode a large number of key-value pairs, for example, using 3H-GCT, it takes only about 4.9s to encode 2^{20} key-value pairs.
OTSA. Zhao and Chow [ZC15] proposed a primitives called oblivious transfer for a sparse array (OTSA), which can be used to construct a variant of PSI, i.e. threshold private set intersection (t-PSI). In fact, the OTSA is strictly stronger than OKVS. The OTSA is actually a *protocol* for obliviously decoding OKVS, that is, the input of receiver is a set I_r, the input of sender is OKVS $D := \mathsf{Encode}(\{(s_j, e_j)\}_{j \in [n_s]})$, the output of the receiver is $\{\mathsf{Decode}(D, r_j)\}_{j \in [n_r]}$. The main differences between OKVS and OTSA are:

– OTSA enforces the receiver to decode D on limited elements of queries, i.e. I_r, whereas OKVS is simply a data structure that is sent in the clear to the receiver, thus, no limit on the elements of decoding is set.
– In OTSA, the receiver does not know the correspondence between r_j and $\mathsf{Decode}(D, r_j)$ (i.e. sender indices privacy), while in OKVS, the receiver directly knows the relationship between r_j and $\mathsf{Decode}(D, r_j)$.

These limitations have a significant impact on their performance, for example, the experiment in [ZC15] showed that their most efficient OTSA protocol takes about 400s for input size $n = 2^{10}$. It is enough for our construction to use simpler and more efficient OKVS instead of OTSA.
OVDM. In our PSU construction, we proposed a new primitive called oblivious vector decryption-then-matching (OVDM), which is also a *protocol* aiming to decrypt a vector of ciphertexts obliviously and then match the decrypted ciphertext to a given string. The significant differences between OVDM and OTSA are:

– OTSA is the protocol for decoding an *OKVS*, while OVDM is the protocol for decrypting an *encryption scheme*.
– OTSA allows the party providing the decoding material (i.e. I_r) to obtain the decoding result (since the decoding algorithm is written as $\mathsf{Decode}(D, r_j)$, D can be regarded as a "key" in some sense), while OVDM allows the party providing the key to obtain the decryption result.
– The output of OVDM is only 1 bit information of plaintext, i.e. whether the plaintext is equal to a string input by the receiver.
– The order of the decryption results output by OVDM is the same as the order of the ciphertext input by the sender, while OTSA does not preserve this order (i.e. sender indices privacy).

Due to the above differences, the ideas for constructing OTSA and OVDM are different. Our OVDM is more efficient than OTSA because we only need PKE to meet the Re-rand property, while OTSA requires more complex homomorphic PKE.

One may wonder whether the construction of OVDM depends on the particular OKVS construction. We clarify that OVDM and OKVS are two different notions of different usages. We use the combination of OKVS and OVDM to construct mqRPMT, as shown in Section 3. Any OKVS instantiation that meets Randomness can be used for our mqRPMT construction. The only connection between OKVS and OVDM is that they share the same encryption scheme, that is, the value encoded by OKVS is the ciphertext of the encryption scheme, and the sender takes the ciphertext decoded from OKVS as her OVDM input. Since decryption is required, the construction of OVDM is related to the selection of encryption schemes (therefore, we classify our schemes as SKE-based and PKE-based).

6 Implementation and Performance

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

– SKE-PSU: PSU protocol with SKE-based mq-RPMT, where SKE and VODM are instantiated with PRP and GMW [GMW87] respectively.
– PKE-PSU: PSU protocol with PKE-based mq-RPMT, where ReRand-PKE is instantiated with ECC ElGamal encryption scheme.

The OKVS instantiation of both schemes uses the 3H-GCT++ in Figure 3. We focus on the case where $n_{\mathsf{y}} = n_{\mathsf{x}} = n$, i.e., both parties have equal-size sets.

6.1 Theoretical Analysis of Communication

In Table 2, we show the theoretical communication complexity of our protocol compared with the Frikken protocol [Fri07], the DC protocol [DC17], the KRTW protocol [KRTW19], the GMRSS protocol [GMR+21] and the JSZDG protocol [JSZ+22] (note that [JSZ+22] proposed two protocols, i.e. JSZDG-R and JSZDG-S, which focus on balanced and unbanlanced setting, respectively) 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,

Protocol	Communication	$n = n_{\mathsf{y}} = n_{\mathsf{x}}$		
		2^{14}	2^{18}	2^{22}
Frikken [Fri07]	$N + 2n_{\mathsf{x}}N + 4n_{\mathsf{y}}N$	$12288n$	$12288n$	$12288n$
DC [DC17]	$2\lambda n_{\mathsf{x}}N + 4n_{\mathsf{y}}N$	$172032n$	$172032n$	$172032n$
KRTW [KRTW19]	$\beta u(2\rho + \lambda + (u+2)\sigma) + \beta u(\kappa + \sigma)$	$14977n$	$16927n$	$18956n$
GMRSS [GMR+21]	$1.27n_{\mathsf{y}}\rho + 3n_{\mathsf{x}}\sigma + (1.27n_{\mathsf{y}} \log n_{\mathsf{y}} + n_{\mathsf{y}})(\kappa + \sigma)$	$5417n$	$6687n$	$7947n$
JSZDG-R [JSZ+22]	$\rho(\kappa + 2.18n_{\mathsf{x}}) + 4n_{\mathsf{y}}l_2 + (1.09n_{\mathsf{x}} \log n_{\mathsf{x}} + n_{\mathsf{y}})(\kappa + \sigma)$	$5757n$	$6931n$	$8105n$
JSZDG-S [JSZ+22]	$\rho(\kappa + 2.18n_{\mathsf{y}}) + 1.09n_{\mathsf{y}}(ul_2 + \sigma) + 2.18n_{\mathsf{y}} \log n_{\mathsf{y}}(\kappa + \sigma)$	$10640n$	$13140n$	$15658n$
SKE-PSU	$(1.3n_{\mathsf{x}} + d + \lambda)\sigma + \kappa + n_{\mathsf{y}}\sigma + 4n_{\mathsf{y}}(t + \sigma - \log n_{\mathsf{x}}) + n_{\mathsf{y}}(\kappa + \sigma)$	$3768n$	$3810n$	$3853n$
PKE-PSU	$4\kappa(1.3n_{\mathsf{x}} + d + \lambda) + 4\kappa n_{\mathsf{y}} + n_{\mathsf{y}}(\kappa + \sigma)$	$1373n$	$1381n$	$1389n$

Table 2: Theoretical communication costs of PSU protocols (in bits), calculated using computational security κ= 128 and statistical security λ= 40. Ignore costs of base OTs which are independent of input size. N is the size of the public key in Pallier encryption scheme (2048 is used here). β and u are the number of bins and maximum bin size respectively. ρ is the width of OT extension matrix (depends on n and protocol).

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 3.

For set sizes in the range 2^{14} to 2^{22}, our PKE-PSU variant has the least communication of any of the protocols we consider: up to an $8.8\times$ improvement of Frikken, $125\times$ improvement of DC, $10.9-13.6\times$ improvement of KRTW, $3.9-5.7\times$ improvement of GMRSS, and $4.2-11.3\times$ improvement of JSZDG. It means that our scheme has great advantages in low bandwidth scenarios.

For our SKE-based protocol, as mentioned in Section 5.2, we use LowMC [ARS+15] to minimize the number of AND gates. Though the communication of our SKE-PSU protocol is about $3\times$ heigher than PKE-PSU, it is still lower than all previous schemes.

6.2 Experimental Setup

We run our experiments on a single Intel Core i9-9900K with 3.6GHz and 128GB RAM. We simulate the network connection using Linux `tc` command. To better meet the potential deployment requirements, we use Netty[8] to maintain the communication channel. And we use Protocol Buffers[9] for data (de-)serialization. Refer to Appendix G.1 for details of Netty and Protocol Buffers.

6.3 Implementation Details

Existing PSU implementations are under different MPC frameworks and different experimental settings. For example, the [KRTW19] implementation is under 128-bit element length while the [GMR+21] implementation is under 64-bit element length. Also, the [KRTW19, JSZ+22] implementation supports multi-thread execution, while the [GMR+21] implementation does not. Further, the [GMR+21] and [JSZ+22] implementation heavily relies on 1-out-of-2 Oblivious Transfer (OT). Introducing recent silent OT technique may further reduce its communication cost [BCG+19, YWL+20]. However, existing efficient silent OT implementation [YWL+20] is available in emp-toolkit [WMK16]. Combining these implementations rely on relatively heavy source code modification works.

After carefully studying existing open-source codes, we fully re-implement state-of-the-art PSU protocols [KRTW19, GMR+21, JSZ+22] and their underlying basic protocols using Java, including base OT [NP01], OT extension [ALSZ13], silent OT [YWL+20], the specific OPRF variant [KKRT16], and GCT data structures.

We choose Java as our primary programming language for the following reasons. First, recent advances in MPC make this attractive data security technique from theory into practical usage. Introducing big data frameworks into MPC would further increase its efficiency and integrate MPC with existing data pipelines [BKC+22]. Current widely adopted big data analytical engines, for example, Hadoop and Spark, are built upon Java or JVM-based programming languages. We hope our implementation can help developers from the big data community leverage and deploy MPC in a more scaling manner. Second, one may think that Java is much slower than C/C++. It is shown

[8] https://netty.io/

[9] https://developers.google.com/protocol-buffers

that although there is some performance gap, most basic operations in Java and C/C++ have similar performances[10].

For operations that have a huge efficiency gap between Java and C/C++, we use the Java Native Interface (JNI) technique to invoke C/C++ libraries. The details can be found in Appendix G.2.

We note that our implementations support multi-thread executions for all the PSU schemes, including [GMR+21], achieved by using Java 'Stream.parallel()'. In our experiments, we limit the number of threads during the protocol execution by setting the JVM parameter 'java.util.concurrent.ForkJoinPool. common.parallelism', and submit all parallel executions into that common thread pool. In the single-thread setting, we let all procedures run in the main thread instead of simply setting the number of threads to be one under the multi-thread setting, thus avoiding additional costs for creating and destroying sub-threads. Our performance reports show that we obtained improved performance results for the [GMR+21] PSU scheme.

Although most operations in Java and C/C++ have similar performances, there are some operations in which Java operates much slower than C/C++. For example, our JSZDG performance results (See Table 3) are about 3 times slower than the report shown in the original work [JSZ+22]. We carefully analyze our implementation and find that the gap is from its underlying batch OPRF proposed by Chase and Miao [CM20]. Briefly speaking, this batched OPRF needs to map each element into a long pseudo-random byte array via a PRF and then convert that to be an integer array as coordinates in the random encoding matrix. In C/C++, the transformation can be done by simply changing the pointer type from uint8_t* to uint32_t* with almost no additional cost. However, such an operation is not supported in Java due to the memory protection mechanism. One has to explicitly convert byte[] to int[], involving dramatic costs. In addition, the type conversion operation cost is, unfortunately, lower than JNI invoking. Introducing JNI in this operation leads to even more costs. How to efficiently implement the batch OPRF proposed by Chase and Miao [CM20] in memory-safe programming language as in C/C++ remains an open problem in the implementation. We emphasize that designing a unified framework for all PSU protocols while compatible with widely adopted big data analytical engines under C/C++ would further lead to better performance results. We hope that our implementation can be a starting point. Our complete implementation is available on GitHub: http://github.com/alibaba-edu/mpc4j.

6.4 Experimental Details

The SKE-PSU protocol is instantiated with the LowMC encryption scheme [ARS+15] where the block size and the key length are both 128 bits, and the number of Sboxes is $m = 10$ (i.e., the SboxLayer is a 10-folded parallel application of the basic 3-bit Sbox on the first 30 bits of the state, and for the remaining 88 bits, the SboxLayer is the identity). The concrete parameters in LowMC are from the Mobile PSI implementations provided by Kales et al. [KRS+19][11]. We use the improved inverse of the SBoxLayer provided by Liu et al. [LIM21] and follow the SBoxLayer implementation idea by Kales et al. [KRS+19] to implement the (non-2PC) decryption procedure. The underlying OKVSs for our PSU protocols are instantiated with our 3H-GCT++ in Figure 3.

Since both [GMR+21] and [JSZ+22] protocols rely heavily on OSN [MS13] and involve a large number of OT. We further introduce Silent OT [BCG+19, YWL+20] in the GMRSS and JSZDG schemes. See details in Appendix G.3.

In SKE-PSU, we assume a commonly used setting where Boolean multiplication triples are precomputed offline and stored locally in a temporary file. This follows real scenarios where Boolean multiplication triples are pre-generated by parties themselves or with the help of a Trusted-Third Party under the Trusted Dealer model. For completeness, we give the costs of triple generation in Appendix G.4.

In PKE-PSU, the ReRand-PKE is instantiated with the ECC ElGamal encryption scheme under the curve SecP256K1. We found an interesting point in the implementation of PKE-PSU: In elliptic-curve-based cryptography, point compression is a standard trick, which can roughly reduce

[10] Our tests show that on Macbook Pro 2019, Java needs 0.095us for one AES operation, while C/C++ under AES instruction needs 0.071us. This is because Java would automatically use AES instruction if it detects that the current operating system supports it.

[11] https://github.com/contact-discovery/mobile_psi_cpp/blob/master/droidCrypto/lowmc/lowmc_128_128_20.c

n	Protocol	Comm. (MB)					Running time (s)															
		$\mathcal{R}$		$\mathcal{S}$		total	LAN				1Gbps				100Mbps				10Mbps			
							$T = 1$		$T = 8$		$T = 1$		$T = 8$		$T = 1$		$T = 8$		$T = 1$		$T = 8$	
		setup	online	setup	online		setup	online	setup	online	setup	online	setup	online	setup	online	setup	online	setup	online	setup	online
2^{14}	KRTW	0.02	4.17	0.01	29.63	33.8	0.07	3.5	0.03	1.07	0.49	16.13	0.37	14.06	0.83	27.36	0.72	24.66	0.81	55.9	0.73	55.32
	GMRSS	0.02	5.89	0.02	7.96	13.85	0.1	1.01	0.04	0.42	0.66	1.96	0.46	1.28	1	3.53	0.91	2.97	1.06	14.44	0.93	13.97
	JSZDG-R	0.01	4.65	0.01	5.63	10.28	0.07	1.81	0.02	0.52	0.27	2.65	0.23	1.34	0.49	4.19	0.41	2.66	0.45	12.08	0.37	10.63
	SKE-PSU	0.01	3.16	0	3.36	6.52	0.03	0.65	0.02	0.29	0.12	6.76	0.11	6.48	0.21	12.66	0.19	12.09	0.2	15.62	0.19	15.59
	PKE-PSU	0.01	1.16	0	1.59	2.75	4.6	2.37	4.58	1.07	4.78	2.63	4.75	1.34	4.92	3.02	4.9	1.77	4.99	4.43	4.91	3.79
	PKE-PSU*	0.01	2.16	0	2.9	5.05	4.6	1.96	4.6	0.59	4.75	2.36	4.76	1	4.95	2.76	4.91	1.54	4.92	5.72	4.93	5.31
2^{16}	KRTW	0.02	17.64	0.01	122.05	139.69	0.07	12.57	0.03	3.76	0.46	26.27	0.39	20.96	0.82	40.09	0.73	36.3	0.81	163.48	0.75	161.63
	GMRSS	0.02	25.95	0.02	34.11	60.06	0.11	4.79	0.04	1.95	0.64	6.61	0.48	4.25	1.11	12.67	0.92	9.78	1.04	60.75	0.94	57.5
	JSZDG-R	0.01	20.75	0.01	24.74	45.49	0.07	7.5	0.02	2.25	0.3	9.29	0.2	4.45	0.44	13.78	0.4	8.58	0.47	49.41	0.42	44.58
	SKE-PSU	0.01	12.61	0	13.41	26.03	0.04	2.66	0.02	1.15	0.13	8.66	0.11	7.32	0.2	15.84	0.19	14.39	0.2	31.79	0.19	30.98
	PKE-PSU	0.01	4.62	0	6.37	10.99	4.62	9.75	4.59	4.39	4.82	10.21	4.76	5.22	4.9	10.94	4.91	5.83	5.01	16.38	4.92	13.61
	PKE-PSU*	0.01	8.63	0	11.57	20.19	4.57	7.96	4.6	2.58	4.76	8.68	4.77	3.37	4.93	9.94	4.91	4.65	4.94	21.46	4.93	19.67
2^{18}	KRTW	0.02	69.29	0.01	562.76	632.05	0.08	63.02	0.03	17.67	0.52	85.56	0.39	45.31	0.76	111.14	0.71	113.83	0.84	660.33	0.74	664.93
	GMRSS	0.02	113.7	0.02	145.11	258.81	0.13	20.74	0.03	9.8	0.58	28.62	0.55	16.63	1.09	49.68	0.93	38.82	1.03	251.84	0.97	243.63
	JSZDG-R	0.01	92.67	0.01	107.89	200.56	0.07	41.15	0.03	10.71	0.25	43.17	0.21	16.84	0.42	64.06	0.4	33.8	0.53	221.27	0.39	191.2
	SKE-PSU	0.01	50.34	0	53.51	103.85	0.04	10.78	0.02	4.88	0.12	17.83	0.1	12.32	0.2	28.38	0.18	22.54	0.21	98.96	0.19	95.72
	PKE-PSU	0.01	18.5	0	25.45	43.95	4.6	41.5	4.59	19.82	4.79	42.37	4.75	20.97	4.92	44.8	4.91	23.38	4.92	66.68	4.9	54.39
	PKE-PSU*	0.01	34.5	0	46.26	80.76	4.61	34.63	4.58	12.26	4.78	37.1	4.75	13.99	4.92	40.62	4.92	18.45	4.91	85.31	4.92	79.22
2^{20}	KRTW	0.02	300.14	0.01	2305.8	2605.95	0.11	245.37	0.04	67.97	0.52	281.96	0.38	120.35	0.82	363.95	0.74	361.12	0.84	2643.84	0.75	2638.05
	GMRSS	0.02	493.2	0.02	615.9	1109.1	0.11	100.48	0.04	48.53	0.62	119.98	0.51	75.76	1.11	207.83	0.95	164.25	1.09	1074.33	0.95	1030.3
	JSZDG-R	0.01	405.53	0.01	467.26	872.79	0.08	173.07	0.04	54.41	0.48	184.63	0.2	73.28	0.47	266.51	0.73	146.13	0.47	941.5	0.72	825.16
	SKE-PSU	0.01	200.88	0	213.55	414.43	0.05	44.73	0.03	22.78	0.13	59.65	0.11	35.71	0.2	86.11	0.2	65.18	0.21	378.57	0.4	369.24
	PKE-PSU	0.01	74	0	101.8	175.8	4.65	168.79	4.6	79.95	4.78	169.18	4.79	86.49	4.97	179.58	4.94	96.32	4.97	269.32	4.87	216.19
	PKE-PSU*	0.01	138	0	185	323	4.64	144.24	4.58	50.56	4.75	146.41	4.74	60.5	4.9	161.26	5	76.33	4.99	345	4.9	313.37

Table 3: Communication cost (in MB) and running time (in seconds) comparing our protocols to KRTW GMRSS, and JSZDG-R. The LAN network has 10 Gbps bandwidth and 0.2 ms RTT latency. Communication cost of $\mathcal{S}/\mathcal{R}$ indicates the outgoing communication from $\mathcal{S}/\mathcal{R}$ to the other party. The best protocol within a setting is marked in blue.

the representation of an EC point by half. The cost of this trick is that one has to perform point decompression in the future, which is typically considered to be cheap. Somewhat surprisingly, it turns out that point decompression is very costly. According to existing implementations provided in MCL and OpenSSL libraries, point decompression is as expensive as point exponentiation. Due to this fact, we prefer to use standard point representation for better efficiency when bandwidth is not of first priority. In the implementation, we use **PKE-PSU*** to represent the version that does not perform point compression.

The simulated network settings include typical LAN (10Gbps bandwidth and 0.02ms RTT latency) and WAN (including 1Gbps with 40ms latency, 100Mbps and 10Mbps bandwidth with 80ms latency). In our KRTW implementation, we follow the pipelining optimization shown in [KRTW19] with 2^8 pipelining size when the receiver sends polynomials to the sender. In our PKE-PSU, we also leverage the pipelining optimization with the same 2^8 pipelining size when the sender sends ReRand outputs to the receiver.

We divide all protocols into two phases: the one-time setup phase and the online phase. As the name suggests, the one-time setup phase does necessary operations before actual protocol execution, including key distribution, base OT execution, and the one-time setup phase for Ferret OT [YWL+20]. The online phase does subsequent protocol executions. Note that in our PKE-PSU, the receiver can send the public key to the sender in the one-time setup phase, and all fixed-point precomputations related to the public key can also be done in that phase. We emphasize that fixed-point precomputations only need to be performed once, regardless of the number of subsequent protocol executions.

Since the JSZDG-S scheme [JSZ+22] focus on unbalanced setting and its perfomance is about $2\times$ worse than the JSZDG-R scheme, we only compare our schemes with JSZDG-R here. Detailed comparisons for set sizes $2^{14}, 2^{16}, 2^{18}, 2^{20}$ and controlled network configurations are shown in Table 3. To be more intuitive, we show the variation of the running time with the bandwidth in different setting in Figure 11.

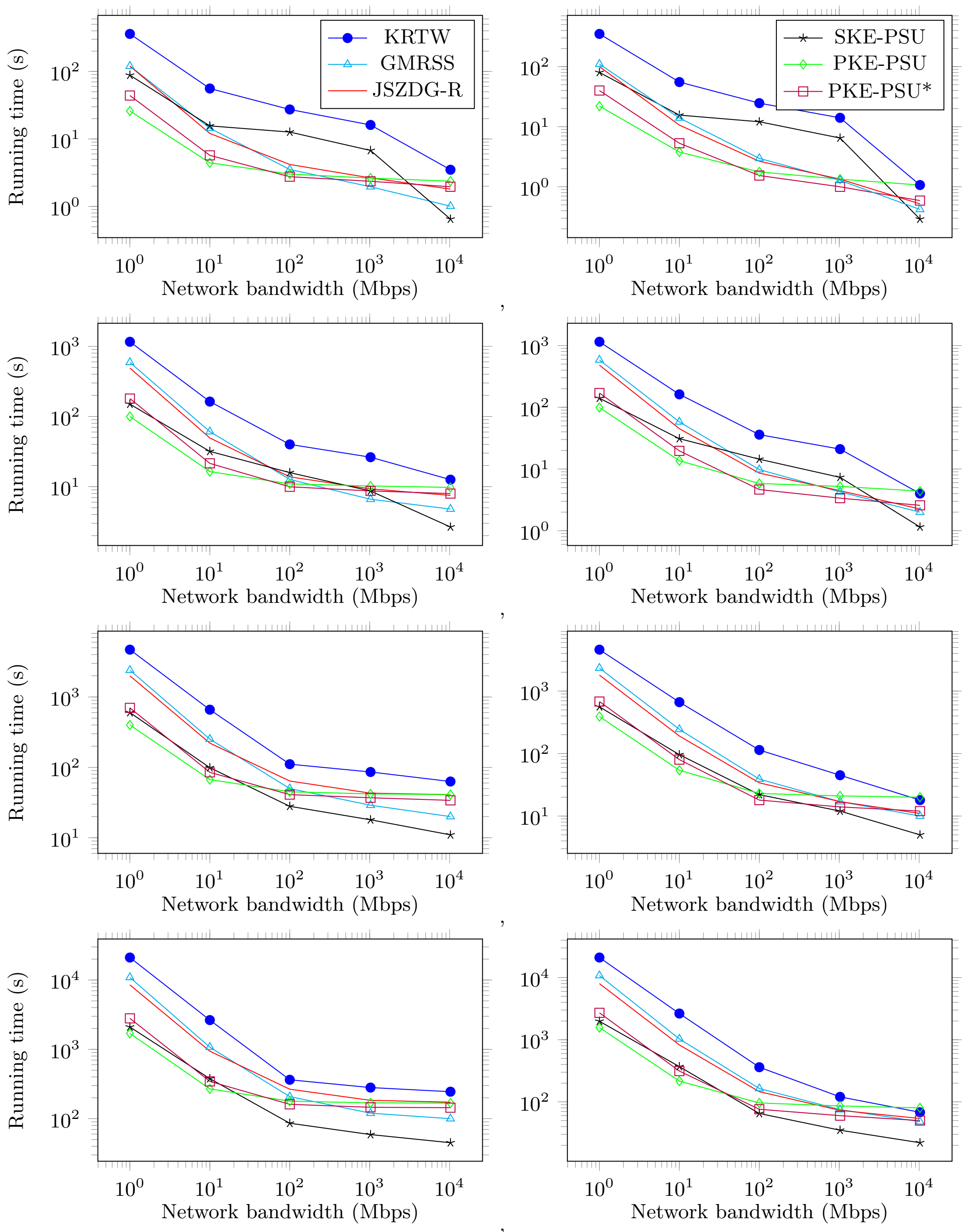

Fig. 11: Decline of running time (in seconds) on increasing network bandwidth for our protocols compared with KRTW, GMRSS and JSZDG-R. Both x and y-axis are in log scale. The four figures on the left correspond to $T = 1$ and the right correspond to $T = 8$. The corresponding set sizes from the first row to the last row are $n = 2^{14}, 2^{16}, 2^{18}, 2^{20}$ respectively.

6.5 Performance Evaluation

Communication improvement. As shown in Table 3, our PKE-PSU protocol has the lowest communication among all protocols, which is $12.3 - 14.8\times$ lower than KRTW, $5.1 - 6.3\times$ lower than GMRSS and $3.7 - 5\times$ lower than JSZDG-R. The communication of PKE-PSU* is about $2\times$ higher than that of PKE-PSU, which is due to the absence of point compression. The communication of

our SKE-PSU is about 2.5× higher than that of PKE-PSU. Nevertheless, all our schemes have lower communication than that of KRTW, GMRSS and JSZDG-R schemes. Since the communication costs of all our protocols are linear with the parties' set sizes, while the communication costs of the other protocols are not. The larger the parties' set sizes are, the larger the communication cost ratios are.
Computation improvement. As shown in Table 3 and Figure 11, our SKE-PSU performs best when the set size and the bandwidth are large. For example, for $n = 2^{20}$ with $T = 1$ thread in LAN setting, SKE-PSU requires 44.73 seconds, achieving a 5.5× improvement over KRTW, a 2.2× improvement over GMRSS, and a factor of 3.9× improvement over JSZDG-R.

Our PKE-PSU and PKE-PSU* could be seen as a trade-off between communication and computation. Both schemes perform better in lower bandwidth. Our PKE-PSU scheme is the fastest one under 10Mbps, which is due to its lowest communication, e.g., for $n = 2^{20}$, PKE-PSU requires 216.19 seconds with $T = 8$ threads, while KRTW requires 2638.05 seconds, a 12.2× improvement, GMRSS requires 1030.3 seconds, a 4.8× improvement, and JSZDG-R requires 825.16 seconds, a 3.8× improvement. Our PKE-PSU* performs better in medium bandwidth (100Mbps and 1Gbps). For example, for $n = 2^{18}$ with $T = 8$ threads in 100Mbps, PKE-PSU* requires 18.45 seconds, while KRTW requires 113.83 seconds, a 6.2× improvement, GMRSS requires 38.82 seconds, a 2.1× improvement, and JSZDG-R requires 33.8 seconds, a 1.8× improvement. We also noticed that the performance of PKE-PSU* improved significantly (about 3× speedup) in the case of multithreading because of its heavy computation cost.

6.6 Applications

We further gave the experiment results of two PSU applications introduced in Section 1, namely IP blacklist aggregation and Private ID. Due to space limitations, the detailed experiment is shown in Appendix H.

Acknowledgement

We are grateful for the helpful comments from the anonymous reviewers. 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). Yu Chen and Min Zhang are supported by the National Key Research and Development Program of China (Grant No. 2021YFA1000600) and the National Natural Science Foundation of China (Grant No. 62272269).

References

[ALSZ13] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. More efficient oblivious transfer and extensions for faster secure computation. In *CCS 2013*, 2013.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In *EUROCRYPT 2015*, 2015.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. In *ASIACCS 2012*, 2012.

[BCG+19] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. Efficient two-round OT extension and silent non-interactive secure computation. In *CCS 2019*, 2019.

[Bea91] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *CRYPTO 1991*, 1991.

[Bf12] M. Burkhart and Xenofontas Dimitropoulos fontas. Fast private set operations with sepia. 2012.

[BKC+22] Saikrishna Badrinarayanan, Ranjit Kumaresan, Mihai Christodorescu, Vinjith Nagaraja, Karan Patel, Srinivasan Raghuraman, Peter Rindal, Wei Sun, and Minghua Xu. A plug-n-play framework for scaling private set intersection to billion-sized sets. Cryptology ePrint Archive, Paper 2022/294, 2022. https://eprint.iacr.org/2022/294.

[BKM+20] Prasad Buddhavarapu, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Vlad Vlaskin. Private matching for compute. Cryptology ePrint Archive, Paper 2020/599, 2020. https://eprint.iacr.org/2020/599.

[BS05] Justin Brickell and Vitaly Shmatikov. Privacy-preserving graph algorithms in the semi-honest model. In *ASIACRYPT 2005*, 2005.

[CM20] Melissa Chase and Peihan Miao. Private set intersection in the internet setting from lightweight oblivious PRF. In *CRYPTO 2020*, 2020.

[CPPT14] Ran Canetti, Omer Paneth, Dimitrios Papadopoulos, and Nikos Triandopoulos. Verifiable set operations over outsourced databases. In *PKC*, 2014.

[DC17] Alex Davidson and Carlos Cid. An efficient toolkit for computing private set operations. In *ACISP 2017*, 2017.

[DCW13] Changyu Dong, Liqun Chen, and Zikai Wen. When private set intersection meets big data: an efficient and scalable protocol. In *CCS 2013*, 2013.

[FNO19] Brett Hemenway Falk, Daniel Noble, and Rafail Ostrovsky. Private set intersection with linear communication from general assumptions. In *WPES@CCS 2019*, 2019.

[FNP04] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *EUROCRYPT 2004*, 2004.

[Fri07] Keith B. Frikken. Privacy-preserving set union. In *ACNS 2007*, 2007.

[Gam85] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theory*, 31(4):469–472, 1985.

[GMR$^+$21] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. Private set operations from oblivious switching. In *PKC 2021*, 2021.

[GMW87] 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.

[GN19] Satrajit Ghosh and Tobias Nilges. An algebraic approach to maliciously secure private set intersection. In *EUROCRYPT 2019*, 2019.

[Gol04] Oded Goldreich. *The Foundations of Cryptography - Volume 2: Basic Applications*. Cambridge University Press, 2004.

[GPR$^+$21] 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.

[HKK$^+$11] Jeongdae Hong, Jung Woo Kim, Jihye Kim, Kunsoo Park, and Jung Hee Cheon. Constant-round privacy preserving multiset union. Cryptology ePrint Archive, Report 2011/138, 2011. https://ia.cr/2011/138.

[HLS$^+$16] Kyle Hogan, Noah Luther, Nabil Schear, Emily Shen, David Stott, Sophia Yakoubov, and Arkady Yerukhimovich. Secure multiparty computation for cooperative cyber risk assessment. In *SecDev 2016*, 2016.

[HN10] Carmit Hazay and Kobbi Nissim. Efficient set operations in the presence of malicious adversaries. In *PKC 2010*, 2010.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. In *CRYPTO 2003*, 2003.

[JSZ$^+$22] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. Shuffle-based private set union: Faster and more secure. In *USENIX Security 22*, 2022.

[KK13] Vladimir Kolesnikov and Ranjit Kumaresan. Improved OT extension for transferring short secrets. In *CRYPTO 2013*, 2013.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. In *CCS 2016*, 2016.

[KRS$^+$19] Daniel Kales, Christian Rechberger, Thomas Schneider, Matthias Senker, and Christian Weinert. Mobile private contact discovery at scale. In *USENIX Security 2019*, 2019.

[KRTW19] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. Scalable private set union from symmetric-key techniques. In *ASIACRYPT*, 2019.

[KS05] Lea Kissner and Dawn Xiaodong Song. Privacy-preserving set operations. In *CRYPTO 2005*, 2005.

[LIM21] Fukang Liu, Takanori Isobe, and Willi Meier. Cryptanalysis of full lowmc and lowmc-m with algebraic techniques. In *CRYPTO 2021*, 2021.

[LV04] Arjen K. Lenstra and Tim Voss. Information security risk assessment, aggregation, and mitigation. In *ACISP 2004*, 2004.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In *EUROCRYPT 2013*, 2013.

[NP01] Moni Naor and Benny Pinkas. Efficient oblivious transfer protocols. In *Proceedings of the Twelfth Annual Symposium on Discrete Algorithms*, 2001.

[PRTY19] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *CRYPTO 2019*, 2019.

[PRTY20] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. PSI from paxos: Fast, malicious private set intersection. In *EUROCRYPT 2020*, 2020.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In *USENIX Security 2015*, 2015.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *USENIX Security*, 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. *IACR Cryptol. ePrint Arch.*, 2005:187, 2005.

[Reg05] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *STOC 2005*, 2005.

[RMY20] Sivaramakrishnan Ramanathan, Jelena Mirkovic, and Minlan Yu. BLAG: improving the accuracy of blacklists. In *NDSS*, 2020.

[RS21] Peter Rindal and Phillipp Schoppmann. VOLE-PSI: fast OPRF and circuit-psi from vector-ole. In *EUROCRYPT 2021*, 2021.

[SCK12] Jae Hong Seo, Jung Hee Cheon, and Jonathan Katz. Constant-round multi-party private set union using reversed laurent series. In *PKC 2012*, 2012.

[SM18] Katsunari Shishido and Atsuko Miyaji. Efficient and quasi-accurate multiparty private set union. In *SMARTCOMP 2018*, 2018.

[WMK16] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[Yao86] Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, 1986.

[YWL$^+$20] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. Ferret: Fast extension for correlated OT with small communication. In *CCS 2020*, 2020.

[ZC15] Yongjun Zhao and Sherman S. M. Chow. Are you the one to share? secret transfer with access structure. Cryptology ePrint Archive, Paper 2015/929, 2015. https://eprint.iacr.org/2015/929.

Appendix

A Encryption Schemes

A.1 Symmetric-key Encryption

A symmetric-key encryption (SKE) scheme is a tuple of four algorithms:

- $\mathsf{Setup}(1^\kappa)$: on input the security parameter κ outputs public parameters pp, which include the description of the message and ciphertext space M, C.
- $\mathsf{KeyGen}(pp)$: on input public parameters pp, outputs a key k.
- $\mathsf{Enc}(k, m)$: on input a key k and a plaintext $m \in M$, outputs a ciphertext $c \in C$.
- $\mathsf{Dec}(k, c)$: on input a key k and a ciphertext $c \in C$, outputs a message $m \in M$ or an error symbol $\perp$.

Correctness. For any $pp \leftarrow \mathsf{Setup}(1^\kappa)$, any $k \leftarrow \mathsf{KeyGen}(pp)$, any $m \in M$, and any $c \leftarrow \mathsf{Enc}(k, m)$, it holds that $\mathsf{Dec}(sk, c) = m$.
Security. For our purpose, we require a case-tailored security notion called *single-message multi-ciphertext pseudorandomness*. Formally, a SKE scheme is single-message multi-ciphertext pseudorandom if for any PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$:

$$\mathsf{Adv}_{\mathcal{A}}(1^\kappa) = \Pr\left[\beta = \beta' : \begin{array}{l} pp \leftarrow \mathsf{Setup}(1^\kappa); \\ k \leftarrow \mathsf{KeyGen}(pp); \\ (m, state) \leftarrow \mathcal{A}_1(pp); \\ \beta \xleftarrow{\text{R}} \{0,1\}; \\ \text{ for } i \in [n] : c^*_{i,0} \leftarrow \mathsf{Enc}(k, m), c^*_{i,1} \xleftarrow{\text{R}} C; \\ \beta' \leftarrow \mathcal{A}_2(pp, state, \{c^*_{i,\beta}\}_{i \in [n]}) \end{array}\right] - \frac{1}{2}$$

is negligible in κ.

Remark 1. The single-message multi-ciphertext pseudorandomness is a mild security notion that is satisfied by most IND-CPA secure SKE schemes, for instance, the classical PRF-based SKE.

A.2 Re-randomizable PKE

A re-randomizable PKE (ReRand-PKE) scheme is a tuple of five algorithms:

- $\mathsf{Setup}(1^\kappa)$: on input the security parameter κ outputs public parameters pp, which include the description of the message and ciphertext space M, C.
- $\mathsf{KeyGen}(pp)$: on input public parameter pp, outputs a keypair (pk, sk).
- $\mathsf{Enc}(pk, m)$: on input a public key pk and a message $m \in M$, outputs a ciphertext $c \in C$.
- $\mathsf{Dec}(sk, c)$: on input a secret key sk and a ciphertext $c \in C$, outputs a message $m \in M$ or an error symbol $\perp$.
- $\mathsf{ReRand}(pk, c)$: on input a public key pk and a ciphertext $c \in C$, outputs another ciphertext $c' \in C$.

Correctness. For any $pp \leftarrow \mathsf{Setup}(1^\kappa)$, any $(pk, sk) \leftarrow \mathsf{KeyGen}(pp)$, any $m \in M$, any $c \leftarrow \mathsf{Enc}(pk, m)$, and any $c' \leftarrow \mathsf{ReRand}(pk, c)$, it holds that $\mathsf{Dec}(sk, c) = \mathsf{Dec}(sk, c') = m$.
Indistinguishability. For any $pp \leftarrow \mathsf{Setup}(1^\kappa)$, any $(pk, sk) \leftarrow \mathsf{KeyGen}(pp)$, and any $m \in M$, the distribution $c_0 \leftarrow \mathsf{Enc}(pk, m)$ and the distribution $c_1 \leftarrow \mathsf{ReRand}(pk, c_0)$ are identical.
Security. For our purpose, we require a case-tailored security notion called *single-message multi-ciphertext pseudorandomness*. Formally, a PKE scheme is single-message multi-ciphertext pseudorandom if for any PPT $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$:

$$\mathsf{Adv}_{\mathcal{A}}(1^\kappa) = \Pr\left[\beta = \beta' : \begin{array}{l} pp \leftarrow \mathsf{Setup}(1^\kappa); \\ (pk, sk) \leftarrow \mathsf{KeyGen}(pp); \\ (m, state) \leftarrow \mathcal{A}_1(pp, pk); \\ \beta \xleftarrow{\text{R}} \{0,1\}; \\ \text{ for } i \in [n] : c^*_{i,0} \leftarrow \mathsf{Enc}(pk, m), c^*_{i,1} \xleftarrow{\text{R}} C; \\ \beta' \leftarrow \mathcal{A}_2(pp, state, \{c^*_{i,\beta}\}_{i \in [n]}) \end{array}\right] - \frac{1}{2}$$

is negligible in κ.

Remark 2. We remark that single-plaintext multi-ciphertext pseudorandomness is a very mild property for PKE. This is because most natural IND-CPA secure PKE constructions satisfy single-message single-ciphertext pseudorandomness, which further implies single-plaintext multi-ciphertexts pseudorandomness via a standard hybrid argument.

It is straightforward to verify that the DDH-based ElGamal PKE [Gam85] and Regev's LWE-based PKE [Reg05] are re-randomizable PKE schemes satisfying the above correctness, indistinguishability, and single-plaintext multi-ciphertext pseudorandomness.

B Oblivious Key-Value Store Scheme

B.1 Instantiation of OKVSs

We recall some instantiations of OKVS and analyze their parameters.

Polynomial. Polynomial can be seen as a natural OKVS: to insert n key-value pairs $\{(x_i, y_i)\}_{i\in[n]}$, one computes P as the polynomial which passes through points $\{(x_i, y_i)\}_{i\in[n]}$.

The advantage of polynomial is that its rate reaches optimal 1, which induces the lowest communication in the protocol. However, its encoding and decoding are less efficient. Using the optimization of [PRTY19], the encoding and decoding complexity are respectively $O(n \log^2 n)$ and $O(\log n)$. Another disadvantage of polynomial is that it only satisfies correctness and obliviousness, not randomness because polynomial generation is a deterministic algorithm.

Garbled Bloom Filter. Garbled Bloom Filter (GBF) was introduced in [DCW13] in the context of PSI protocols. The values are taken from $\mathbb{F}_{2^\sigma}$. A GBF is an m-long array D associated with k random functions $h_1, \ldots, h_k : \{0,1\}^* \to [m]$. Let $D[j]$ denote the jth component of array D. To insert a key-value pair (x, y) in a GBF, one chooses random $D[h_i(x)]$ for $i \in [k]$ conditioned on $y = \oplus_{i\in[k]} D[h_i(x)]$.

In [DCW13], they showed that if the GBF has size $m = O(\lambda n)$ then the generation of GBF succeeds with probability $1 - 2^{-\lambda}$. Therefore, the rate of GBF is $O(1/\lambda)$. The encoding complexity is $O(\lambda n)$ and decoding requires λ XOR at most.

Garbled Cuckoo Table. Garbled Cuckoo Table (GCT) was introduced in [PRTY20] as an optimization of GBF. The idea of GCT is similar to GBF, and the difference is that GCT uses only two hash functions instead of λ. However, two hash functions will cause a non-negligible probability of failure. To solve this problem, they introduce some additional positions and use a new random function to map the key to these positions. They use cuckoo graph to analyze the probability of success, and finally they achieve a better rate, which is about 0.42.

However, as Rindal and Schoppmann [RS21] pointed out, the original GCT scheme [PRTY20] does not meet the obliviousness properties we defined before. The main reason is that the original GCT needs to solve a linear equation to satisfy the key-value constraint. However, the free variable in the equation is set to zero, which means keys are no longer randomly shared in some positions like GBF. As a result, the GCT has some zeros depending on the key's set. They made a little modification to make GCT (they called XoPaXoS) meet this property. The main idea is to first assign random values to the free variables, and then solve the remaining full rank equations. Recently, Garimella et al. [GPR+21] improved original GCT to 3H-GCT, the rate is increased to 0.81 by using three hash functions. However, the original 3H-GCT still assigns zero to the free variables in linear equation. We use similar modifications to make 3H-GCT meet Obliviousness and Randomness, as we described in Figure 3.

We summarize the parameters and properties of the above schemes in Table 4.

B.2 Property Proof

The correctness is obvious. Now we prove the Obliviousness and Randomness of our 3H-GCT++.

Theorem 7. *3H-GCT++ in Figure 3 satisfies the Obliviousness and Randomness.*

Proof. **Obliviousness**: As we described before, 3H-GCT++ is generated by additive secret sharing of values at the random position mapped by hash function, and selecting random value at the point not

scheme	rate	encoding	decoding	obliviousness	randomness
Polynomial	1	$O(n\log^2 n)$	$O(\log n)$	$\surd$	$\times$
GBF [DCW13]	$O(1/\lambda)$	$O(\lambda n)$	$O(\lambda)$	$\surd$	$\surd$
2H-GCT [PRTY20]	$0.42-o(1)$	$O(\lambda n)$	$O(\lambda)$	$\times$	$\times$
XoPaXoS [RS21]	$0.42-o(1)$	$O(\lambda n)$	$O(\lambda)$	$\surd$	$\surd$
3H-GCT [GPR+21]	$0.81-o(1)$	$O(\lambda n)$	$O(\lambda)$	$\times$	$\times$
3H-GCT++ in Figure 3	$0.81-o(1)$	$O(\lambda n)$	$O(\lambda)$	$\surd$	$\surd$

Table 4: A comparison between the different OKVS schemes. n is the number of key-value pairs, λ is a statistical security parameter (e.g.,$\lambda = 40$).

mapped. Since the value are uniform distribution, we have that $\{(D_1,\ldots,D_m)|D_i \leftarrow \mathbb{G}, i \in [m]\} \equiv \{(D_1,\ldots,D_m)|y \leftarrow \mathbb{G}, D_i \leftarrow \mathbb{G}, i \in [m-1], D_m := y - \sum_{i\in[m-1]} D_i\}$, which implies Obliviousness.

Randomness: Let $X = \{x_1,\ldots,x_n\}$ denote the key's set. For any $x^* \notin X$, let $\mathrm{row}(x^*)$ defined as before. There are three cases:

Case 1: $\exists x_i \in X$ such that $\mathrm{row}(x_i) = \mathrm{row}(x^*)$. By the parameter of GCT scheme [PRTY20, GPR+21], this probability is $2^{-\lambda}$.

Case 2: Let $o(x_i) \subset [m]$ be the set of positions that are 1s of $row(x_i)$, $i \in [n]$ and let $O := \cup_{i\in[n]} o(x_i)$. In this case, $o(x^*) \subset O$, that is, all the 1 positions of $\mathrm{row}(x^*)$ have been mapped when generating D. Now we can divide $o(x^*)$ into several groups according to which key is mapped to that location. If there is a location i mapped by both different keys, then the location i can be randomly put into one of the groups. Since the different positions of a key mapped to corresponds to an additive secret sharing of the corresponding value, the sum of each group should be a uniformly random element in $\mathbb{G}$. Therefore $\mathsf{Decode}_H(D, x^*)$ is a uniformly random string.

Case 3: $\exists j \in [m]$ such that $j \in o(x^*) \wedge j \notin O$, that is, there are some positions of $\mathrm{row}(x^*)$ were not mapped when generating D. By the generation of GCT, those positions not mapped are assigned with a random value. Therefore $\mathsf{Decode}_H(D, x^*)$ is a uniformly random string.

In summary, with probability $1 - 2^{-\lambda}$, $\mathsf{Decode}_H(D, x^*)$ is a uniformly random string.

C Proof of Theorem 1

Below we give the details of the proof of Theorem 1.

Proof. We exhibit simulators $\mathsf{Sim}_\mathcal{R}$ and $\mathsf{Sim}_\mathcal{S}$ for simulating corrupt $\mathcal{R}$ and $\mathcal{S}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

Corrupt sender: $\mathsf{Sim}_\mathcal{S}(Y = \{y_1,\ldots,y_{n_\mathsf{y}}\})$ simulates the view of corrupt semi-honest sender. It executes as follows:

1. $\mathsf{Sim}_\mathcal{S}$ selects n_x random key-value pairs $(x_i, s_i)_{i\in[n_\mathsf{x}]}$, where x_i and s_i are random item and ciphertext respectively. Then $\mathsf{Sim}_\mathcal{S}$ computes OKVS $D := \mathsf{Encode}_H((h(x_1), s_1),\ldots,(h(x_{n_\mathsf{x}}), s_{n_\mathsf{x}}))$ and appends it to the view.
2. $\mathsf{Sim}_\mathcal{S}$ computes $s_i^* = \mathsf{Decode}_H(D, h(y_i))$ for $i \in [n_\mathsf{y}]$. Then, it invokes VODM simulator $\mathsf{Sim}^S_{\mathsf{vodm}}(s_1^*,\ldots,s_{n_\mathsf{y}}^*)$ and appends the output to the view.

Now we argue that the view output by $\mathsf{Sim}_\mathcal{S}$ is indistinguishable from the real one. We formally prove this by a standard hybrid argument method. We define four hybrid transcripts T_0, T_1, T_2, T_3 where T_0 is real view of $\mathcal{S}$, and T_3 is the output of $\mathsf{Sim}_\mathcal{S}$.

– Hybrid$_0$. The first hybrid is the real interaction described in Figure 7. Here, an honest $\mathcal{R}$ uses input X, honestly interacts with the corrupt $\mathcal{S}$. Let T_0 denote the real view of $\mathcal{S}$.
– Hybrid$_1$. Let T_1 be the same as T_0, except that $(s_1,\ldots,s_{n_\mathsf{x}})$ are replaced by n_x random ciphertexts. This hybrid is computationally indistinguishable from T_0 by the single-message multi-ciphertext pseudorandomness of the encryption scheme.
  Specifically, if there is a distinguisher $\mathcal{D}$ can distinguish T_0 and T_1 with non-negligible probability, then we can construct a PPT adversary $\mathcal{A}$ to break the single-message multi-ciphertext pseudorandomness of encryption scheme. $\mathcal{A}$ works as follows: when $\mathcal{A}$ receives pp from challenger, $\mathcal{A}$ selects

a random s as challenge message. Then $\mathcal{A}$ receives ciphertexts $\{c_i^*\}_{i\in[n_\mathsf{x}]}$ from challenger. Now $\mathcal{A}$ executes as an honest receiver with the corrupt $\mathcal{S}$ except step 2. In this step, $\mathcal{A}$ computes OKVS as $D := \mathsf{Encode}_H(\{h(x_i), c_i^*\}_{i\in[n_\mathsf{x}]})$. Now $\mathcal{A}$ invokes $\mathcal{D}$ with the sender's view in the above interaction and outputs $\mathcal{D}$'s output. Note that if $\{c_i^*\}_{i\in[n_\mathsf{x}]}$ are the encryption of s, the view of corrupt sender is exactly the real view, which corresponds to T_0. If $\{c_i^*\}_{i\in[n_\mathsf{x}]}$ are random ciphertexts, the view corresponds to T_1. Therefore, $\mathcal{A}$ can break the security of the encryption scheme with the same advantages as $\mathcal{D}$.
– Hybrid$_2$. Let T_2 be the same as T_1, except that the inputs of the receiver $\mathcal{R}$ are replaced by n_x random items. Note that the selection of value in OKVS has been replaced with random ciphertexts in T_1. By the obliviousness property of OKVS, T_1 and T_2 are statistically indistinguishable.
– Hybrid$_3$. Let T_3 be the same as T_2, except that the VODM execution is replaced by simulator $\mathsf{Sim}_{\mathsf{vodm}}^{S}$. The security of VODM functionality guarantees the view is indistinguishable from real execution.

Corrupt receiver: $\mathsf{Sim}_{\mathcal{R}}(X = \{x_1, \ldots, x_{n_\mathsf{x}}\}, b)$ simulates the view of corrupt semi-honest receiver. It executes as follows:

1. $\mathsf{Sim}_{\mathcal{R}}$ selects a random $s \leftarrow \{0,1\}^\sigma$ and generates a random encryption key k as the semi-honest receiver does in the real protocol. Then, it invokes VODM simulator $\mathsf{Sim}_{\mathsf{vodm}}^{R}(k, s, b)$ and appends the output to the view.

The view output by $\mathsf{Sim}_{\mathcal{R}}$ is indistinguishable from the real one by the underlying simulators' indistinguishability.

D SKE-based Multi-Query RPMT

Now we show that PRF-based SKE satisfies the single-message multi-ciphertext pseudorandomness property.

Let $F := \{f_k : \{0,1\}^\kappa \to \{0,1\}^\kappa\}_{k\in K}$ be a PRF family. The PRF-based SKE scheme is as follows:

– $\mathsf{Setup}(1^\kappa)$: on input the security parameter κ outputs public parameters pp, which include the description of the message and ciphertext space $M = \{0,1\}^\kappa, C = \{0,1\}^{2\kappa}$.
– $\mathsf{KeyGen}(pp)$: on input public parameter pp, outputs a key $k \xleftarrow{\text{R}} K$.
– $\mathsf{Enc}(k, m)$: on input a key k and a plaintext $m \in M$, chooses $r \xleftarrow{\text{R}} \{0,1\}^\kappa$, outputs a ciphertext $c = (r, f_k(r) \oplus m)$.
– $\mathsf{Dec}(k, c)$: on input a key k and a ciphertext $c = (r, c_2)$, outputs $m = f_k(r) \oplus c_2$.

Next, we prove the single-message multi-ciphertext pseudorandomness of the above PRF-based SKE scheme.

Lemma 1. *The PRF-based SKE satisfies the single-message multi-ciphertext pseudorandomness property defined in Section A.1.*

Proof. If there is a PPT adversary $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ can break the single-message multi-ciphertext pseudorandomness of SKE scheme, then we can construct a PPT adversary $\mathcal{B}$ to break the security of PRF. In particular, $\mathcal{B}$ runs Setup to obtain pp, then it invokes $\mathcal{A}_1(pp)$ to obtain $(m, state)$. Now, $\mathcal{B}$ selects $r_i \xleftarrow{\text{R}} \{0,1\}^\kappa$ and queries the oracle with r_i to obtain $f(r_i)$ for $i \in [n]$. Then, $\mathcal{B}$ sets $c_i = (r_i, f(r_i) \oplus m)$ and invokes $\mathcal{A}_2(pp, state, \{c_i\}_{i\in[n]})$ to obtain a bit β'. Finally, $\mathcal{B}$ outputs β'.

If f is PRF: $\{c_i\}_{i\in[n]}$ are exactly the n times encryption of m, which correspond to $\beta = 0$.

If f is random function: $f(r)$ is also a random string on $\{0,1\}^\kappa$, which means c_i is a random distribution in ciphertext space, corresponding to $\beta = 1$.Therefore, $\mathcal{B}$ distinguishes PRF with the same probability as $\mathcal{A}$ in single-message multi-ciphertext pseudorandomness experiment.

E PSU Construction from Leaky VODM

We give the PSU protocol from leaky VODM in Figure 12.

Parameters:

- Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$.
- A ReRand-PKE scheme ($\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{ReRand}$).
- An OKVS scheme ($\mathsf{Encode}_H, \mathsf{Decode}_H$).
- A collision-resistant hash function $h(x) : \{0,1\}^* \to \{0,1\}^\sigma$.

Input of $\mathcal{S}$: $Y = \{y_1, \ldots, y_{n_\mathsf{y}}\} \subset \{0,1\}^*$
Input of $\mathcal{R}$: $X = \{x_1, \ldots, x_{n_\mathsf{x}}\} \subset \{0,1\}^*$
Protocol:

1. $\mathcal{R}$ selects a random *indication string* $s \in \mathbb{F}_{2^\sigma}$. $\mathcal{R}$ also generates a random key pair $pp \leftarrow \mathsf{Setup}$, $(pk, sk) \leftarrow \mathsf{KeyGen}(pp)$, a randomness set $R = \{r_1, \ldots, r_{n_\mathsf{x}}\}$ and computes $s_i := \mathsf{Enc}(pk, s; r_i)$ for $i \in [n_\mathsf{x}]$.
2. $\mathcal{R}$ computes an OKVS $D := \mathsf{Encode}_H((h(x_1), s_1), \ldots, (h(x_{n_\mathsf{x}}), s_{n_\mathsf{x}}))$.
3. $\mathcal{R}$ sends D and pk to $\mathcal{S}$.
4. $\mathcal{S}$ computes $s_i^* := \mathsf{Decode}_H(D, h(y_i))$ for $i \in [n_\mathsf{y}]$.
5. $\mathcal{S}$ and $\mathcal{R}$ invoke the leaky VODM functionality $\mathcal{F}_{\mathsf{lvodm}}$. The sender $\mathcal{S}$ acts as sender in leaky VODM with input $\{s_i^*\}_{i\in[n_\mathsf{y}]}$ and receives nothing. The receiver $\mathcal{R}$ acts as receiver in leaky VODM with input (pk, sk, s) and receives $b \in \{0,1\}^{n_\mathsf{y}}$ and $\{s_i' | b_i = 0\}$.
6. $\mathcal{R}$ initialize set $Z := \{\}$.
7. For $i \in [n_\mathsf{y}]$:
   (a) $\mathcal{S}$ and $\mathcal{R}$ invoke the OT functionality $\mathcal{F}_{\mathsf{ot}}$
   (b) $\mathcal{S}$ acts as sender with input $(y_i, \perp)$.
   (c) $\mathcal{R}$ acts as receiver with input b_i.
   (d) $\mathcal{R}$ obtains the OT output z_i and sets $Z = Z \cup \{z_i\}$
8. $\mathcal{R}$ outputs $X \cup Z$.

Fig. 12: PSU from leaky VODM Π_{psu}

Theorem 8. *Assume the Re-Rand PKE scheme* $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *satisfies single-message multi-ciphertext pseudorandomness. The protocol in Figure 12 securely computes* $\mathcal{F}_{\mathsf{psu}}$ *against semi-honest adversaries in the* $(\mathcal{F}_{\mathsf{lvodm}}, \mathcal{F}_{\mathsf{ot}})$*-hybrid model.*

Proof. We exhibit simulators $\mathsf{Sim}_\mathcal{R}$ and $\mathsf{Sim}_\mathcal{S}$ for simulating corrupt $\mathcal{R}$ and $\mathcal{S}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

Corrupt Sender: $\mathsf{Sim}_\mathcal{S}(Y = \{y_1, \ldots, y_{n_\mathsf{y}}\})$ simulates the view of corrupt semi-honest sender. It executes as follows:

1. $\mathsf{Sim}_\mathcal{S}$ selects n_x random key-value pairs $(x_i, s_i)_{i\in[n_\mathsf{x}]}$, where x_i and s_i are random item and ciphertext respectively. Then the simulator $\mathsf{Sim}_\mathcal{S}$ generates $pp \leftarrow \mathsf{Setup}$, $(pk, sk) \leftarrow \mathsf{KeyGen}(pp)$, computes $D = \mathsf{Encode}_H((h(x_1), s_1), \ldots, (h(x_{n_\mathsf{x}}), s_{n_\mathsf{x}}))$ and appends (pk, D) to the view.
2. $\mathsf{Sim}_\mathcal{S}$ computes $s_i^* = \mathsf{Decode}_H(D, h(y_i))$ for $i \in [n_\mathsf{y}]$. Then, it invokes leaky VODM simulator $\mathsf{Sim}^S_{\mathsf{lvodm}}(s_1^*, \ldots, s_{n_\mathsf{y}}^*)$ and appends the output to the view.
3. For $i \in [n_\mathsf{y}]$, $\mathsf{Sim}_\mathcal{S}$ invokes OT simulator $\mathsf{Sim}^S_{\mathsf{ot}}(y_i, \perp)$ and appends the output to the view.

Now we argue that the view output by $\mathsf{Sim}_\mathcal{S}$ is indistinguishable from the real one. We formally prove this by a standard hybrid argument method. We define four hybrid transcripts T_0, T_1, T_2, T_3 where T_0 is real view of $\mathcal{S}$, and T_3 is the output of $\mathsf{Sim}_\mathcal{S}$.

- Hybrid_0. The first hybrid is the real interaction described in Figure 12. Here, an honest $\mathcal{R}$ uses input X, honestly interacts with the corrupt $\mathcal{S}$. Let T_0 denote the real view of $\mathcal{S}$.
- Hybrid_1. Let T_1 be the same as T_0, except that $(s_1, \ldots, s_{n_\mathsf{x}})$ are replaced by n_x random ciphertexts. This hybrid is computationally indistinguishable from T_0 by the single-message multi-ciphertext pseudorandomness of the encryption scheme.
  Specifically, if there is a distinguisher $\mathcal{D}$ can distinguish T_0 and T_1 with non-negligible probability, then we can construct a PPT adversary $\mathcal{A}$ to break the single-message multi-ciphertext pseudorandomness of encryption scheme. $\mathcal{A}$ works as follows: when $\mathcal{A}$ receives pp from challenger, $\mathcal{A}$ selects

a random s as challenge message. Then $\mathcal{A}$ receives ciphertexts $\{c_i^*\}_{i\in[n_\mathsf{x}]}$ from challenger. Now $\mathcal{A}$ executes as an honest receiver with the corrupt $\mathcal{S}$ except step 2. In this step, $\mathcal{A}$ computes OKVS as $D := \mathsf{Encode}_H(\{h(x_i), c_i^*\}_{i\in[n_\mathsf{x}]})$. Now $\mathcal{A}$ invokes $\mathcal{D}$ with the sender's view in the above interaction and outputs $\mathcal{D}$'s output. Note that if $\{c_i^*\}_{i\in[n_\mathsf{x}]}$ are the encryption of s, the view of corrupt sender is exactly the real view, which corresponds to T_0. If $\{c_i^*\}_{i\in[n_\mathsf{x}]}$ are random ciphertexts, the view corresponds to T_1. Therefore, $\mathcal{A}$ can break the security of the encryption scheme with the same advantages as $\mathcal{D}$.
- Hybrid$_2$. Let T_2 be the same as T_1, except that the inputs of the receiver $\mathcal{R}$ are replaced by n_x random items. Note that the selection of value in OKVS has been replaced with random ciphertexts in T_1. By the obliviousness property of OKVS, T_1 and T_2 are statistically indistinguishable.
- Hybrid$_3$. Let T_3 be the same as T_2, except that the leaky VODM and OT execution is replaced by simulator $\mathsf{Sim}^{S}_{\mathsf{lvodm}}$ and $\mathsf{Sim}^{S}_{\mathsf{ot}}$. The security of leaky VODM and OT functionality guarantee the view is indistinguishable from real execution.

Corrupt Receiver: $\mathsf{Sim}_{\mathcal{R}}(X = \{x_1, \ldots, x_{n_\mathsf{x}}\}, X \cup Y)$ simulates the view of corrupt receiver. It executes as follows:

1. $\mathsf{Sim}_{\mathcal{R}}$ executes first two steps as an honest receiver and obtains $s, (pk, sk), D$.
2. $\mathsf{Sim}_{\mathcal{R}}$ define the set $Z := X \cup Y \setminus X$, i.e. the set of elements that Y "brings to the union". Next, it uses $\perp$ to pads Z to n_y elements and permutates these elements randomly. Let $Z = \{z_1, \ldots, z_{n_\mathsf{y}}\}$.
3. $\mathsf{Sim}_{\mathcal{R}}$ sets $b_i = 1$ if and only if $z_i \in X$ for $i \in [n_\mathsf{y}]$. For $z_i \notin X$, $\mathsf{Sim}_{\mathcal{R}}$ computes $s_i' := \mathsf{Dec}(sk, \mathsf{Decode}_H(D, h(z_i)))$ Then, it invokes leaky vectro ODM simulator $\mathsf{Sim}^{R}_{\mathsf{lvodm}}(s, b, \{s_i' | b_i = 0\})$ and appends the output to the view.
4. For $i \in [n_\mathsf{y}]$, $\mathsf{Sim}_{\mathcal{R}}$ invokes OT simulator $\mathsf{Sim}^{R}_{\mathsf{ot}}(b_i, z_i)$ and appends the output to the view.

Now we argue that the view output by $\mathsf{Sim}_{\mathcal{R}}$ is indistinguishable from the real one. In the simulation, the way $\mathcal{R}$ obtains the elements in $Z = X \setminus Y$ is identical to the real execution. By the underlying simulators' indistinguishability, the simulated view is computationally indistinguishable from the real one.

F Multi-Query RPMT Based on Membership Encryption

We describe how to construct mq-RPMT using the language of Membership Encryption (ME). As we mentioned in Section 4.3, this will help us reduce the communication by half when sending OKVS. We first define the vector oblivious decryption (VOD) functionality in Figure 13.

Parameters: Sender $\mathcal{S}$, Receiver $\mathcal{R}$, set sizes n, a ME scheme $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.
Functionality:

- Wait for input k from the receiver $\mathcal{R}$.
- Wait for input $\{s_1^*, \ldots, s_n^*\} \subset \{0,1\}^*$ from the sender $\mathcal{S}$.
- For $i \in [n]$:
    Compute $b_i = \mathsf{Dec}(k, s_i^*)$.
- Give output $b \in \{0,1\}^n$ to the receiver $\mathcal{R}$.

Fig. 13: Vector Oblivious Decryption Functionality $\mathcal{F}_{\mathsf{vod}}$

Now, we use the language of ME to describe how to construct mq-RPMT. The formal protocol is described in Figure 14.

Correctness. For all $i \in [n_\mathsf{y}]$, if $y_i \in X$, there is an $x_j \in X, j \in [n_\mathsf{x}]$ s.t. $y_i = x_j$. In this case, $s_i^* = \mathsf{Decode}_H(D, h(x_j)) = s_j$. Since $s_j = \mathsf{Enc}(k, x_j)$, we have $\mathsf{Dec}(k, s_j) = 1$. In the case $y_i \notin X$, if hash functions collide, that is, $h(y_i) = h(x)$ for some $y_i \notin X$, the correctness will be violated. By setting $\sigma = \lambda + \log n_\mathsf{x} n_\mathsf{y}$, a union bound shows probability of collision is negligible $2^{-\lambda}$. When no collision occurs, from the randomness of OKVS, $s_i^* = \mathsf{Decode}_H(D, h(y_i))$ is a random ciphertext, result in $\mathsf{Dec}(k, s_i^*) = 0$ with overwhelming probability. The union bound guarantees that for all $y_i \notin X$, the probability that there exists an s_i^* s.t. $\mathsf{Dec}(k, s_i^*) = 1$ is negligible.

We now state and prove the security properties of the above mq-RPMT protocol.

Theorem 9. *Assume* $\mathcal{E} = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ *is a membership encryption scheme as we defined in section 4.3. The protocol in Figure 14 securely computes* $\mathcal{F}_{\mathsf{mq\text{-}rpmt}}$ *against semi-honest adversaries in the* $\mathcal{F}_{\mathsf{vod}}$*-hybrid model.*

Proof. We exhibit simulators $\mathsf{Sim}_{\mathcal{R}}$ and $\mathsf{Sim}_{\mathcal{S}}$ for simulating corrupt $\mathcal{R}$ and $\mathcal{S}$ respectively, and argue the indistinguishability of the produced transcript from the real execution.

Corrupt sender: $\mathsf{Sim}_{\mathcal{S}}(Y = \{y_1, \ldots, y_{n_\mathsf{y}}\})$ simulates the view of corrupt semi-honest sender. It executes as follows:

1. $\mathsf{Sim}_{\mathcal{S}}$ selects n_x random key-value pairs $(x_i, s_i)_{i \in [n_\mathsf{x}]}$, where x_i and s_i are random item and ciphertext respectively. Then $\mathsf{Sim}_{\mathcal{S}}$ computes OKVS $D := \mathsf{Encode}_H((h(x_1), s_1), \ldots, (h(x_{n_\mathsf{x}}), s_{n_\mathsf{x}}))$ and appends it to the view.
2. $\mathsf{Sim}_{\mathcal{S}}$ computes $s_i^* = \mathsf{Decode}_H(D, h(y_i))$ for $i \in [n_\mathsf{y}]$. Then, it invokes VOD simulator $\mathsf{Sim}_{\mathsf{vod}}^S(s_1^*, \ldots, s_{n_\mathsf{y}}^*)$ and appends the output to the view.

Now we argue that the view output by $\mathsf{Sim}_{\mathcal{S}}$ is indistinguishable from the real one. We formally prove this by a standard hybrid argument method. We define four hybrid transcripts T_0, T_1, T_2, T_3 where T_0 is real view of $\mathcal{S}$, and T_3 is the output of $\mathsf{Sim}_{\mathcal{S}}$.

– Hybrid$_0$. The first hybrid is the real interaction described in Figure 14. Here, an honest $\mathcal{R}$ uses input X, honestly interacts with the corrupt $\mathcal{S}$. Let T_0 denote the real view of $\mathcal{S}$.
– Hybrid$_1$. Let T_1 be the same as T_0, except that $(s_1, \ldots, s_{n_\mathsf{x}})$ are replaced by n_x random ciphertexts. This hybrid is computationally indistinguishable from T_0 by the multi-elements pseudorandomness of the membership encryption scheme.
  Specifically, if there is a distinguisher $\mathcal{D}$ can distinguish T_0 and T_1 with non-negligible probability, then we can construct a PPT adversary $\mathcal{A}$ to break the multi-elements pseudorandomness of membership encryption scheme. $\mathcal{A}$ works as follows: when $\mathcal{A}$ receives ciphertexts $\{c_i^*\}_{i \in [n_\mathsf{x}]}$ from challenger, $\mathcal{A}$ executes as an honest receiver with the corrupt $\mathcal{S}$ except step 2. In this step, $\mathcal{A}$ computes OKVS as $D := \mathsf{Encode}_H(\{h(x_i), c_i^*\}_{i \in [n_\mathsf{x}]})$. Now $\mathcal{A}$ invokes $\mathcal{D}$ with the sender's view in the above interaction and outputs $\mathcal{D}$'s output. Note that if $\{c_i^*\}_{i \in [n_\mathsf{x}]}$ are the encryption of x_i, the view of corrupt sender is exactly the real view, which corresponds to T_0. If $\{c_i^*\}_{i \in [n_\mathsf{x}]}$ are random ciphertexts, the view corresponds to T_1. Therefore, $\mathcal{A}$ can break the multi-elements pseudorandomness of the membership encryption scheme with the same advantages as $\mathcal{D}$.
– Hybrid$_2$. Let T_2 be the same as T_1, except that the inputs of the receiver $\mathcal{R}$ are replaced by n_x random items. Note that the selection of value in OKVS has been replaced with random ciphertexts in T_1. By the obliviousness property of OKVS, T_1 and T_2 are statistically indistinguishable.
– Hybrid$_3$. Let T_3 be the same as T_2, except that the VOD execution is replaced by simulator $\mathsf{Sim}_{\mathsf{vod}}^S$. The security of VOD functionality guarantees the view is indistinguishable from real execution.

Corrupt receiver: $\mathsf{Sim}_{\mathcal{R}}(X = \{x_1, \ldots, x_{n_\mathsf{x}}\}, b)$ simulates the view of corrupt semi-honest receiver. It executes as follows:

1. $\mathsf{Sim}_{\mathcal{R}}$ generates a random encryption key k as the semi-honest receiver does in the real protocol. Then, it invokes VOD simulator $\mathsf{Sim}_{\mathsf{vod}}^R(k, b)$ and appends the output to the view.

The view output by $\mathsf{Sim}_{\mathcal{R}}$ is indistinguishable from the real one by the underlying simulators' indistinguishability.

G Implementation Detail

G.1 Detail for Netty and Protocol Buffers

Netty is an asynchronous event-driven network application framework for the rapid development of maintainable high-performance protocols that are widely used for real applications. We design a unified data package format. Each data package contains a 256-bit length header and the actual payload bytes. The header is defined as follows:

Parameters:

- Two parties: sender $\mathcal{S}$ and receiver $\mathcal{R}$.
- Ideal $\mathcal{F}_{\mathsf{vod}}$ primitives specified in Figure 13.
- A ME scheme $(\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$.
- An OKVS scheme $(\mathsf{Encode}_H, \mathsf{Decode}_H)$
- A collision-resistant hash function $h(x) : \{0,1\}^* \to \{0,1\}^\sigma$.

Input of $\mathcal{S}$: $Y = \{y_1, \ldots, y_{n_\mathsf{y}}\} \subset \{0,1\}^*$
Input of $\mathcal{R}$: $X = \{x_1, \ldots, x_{n_\mathsf{x}}\} \subset \{0,1\}^*$
Protocol:

1. $\mathcal{R}$ uses a ME scheme to generate a random key: $pp \leftarrow \mathsf{Setup}(X, 1^\kappa), k \leftarrow \mathsf{KeyGen}(pp)$ and computes $s_i := \mathsf{Enc}(k, x_i)$ for $i \in [n_\mathsf{x}]$.
2. $\mathcal{R}$ computes an OKVS $D := \mathsf{Encode}_H((h(x_1), s_1), \ldots, (h(x_{n_\mathsf{x}}), s_{n_\mathsf{x}}))$.
3. $\mathcal{R}$ sends D to the sender $\mathcal{S}$.
4. $\mathcal{S}$ computes $s_i^* := \mathsf{Decode}_H(D, h(y_i))$ for $i \in [n_\mathsf{y}]$.
5. $\mathcal{S}$ and $\mathcal{R}$ invoke the vector oblivious decryption functionality $\mathcal{F}_{\mathsf{vod}}$. $\mathcal{S}$ acts as sender with input $S = \{s_1^*, \ldots, s_{n_\mathsf{y}}^*\}$ and $\mathcal{R}$ acts as receiver with input k. As a result, $\mathcal{S}$ receives nothing and $\mathcal{R}$ receives $b \in \{0,1\}^{n_\mathsf{y}}$.

Fig. 14: ME-based Multi-Query Reverse Private Membership Test Protocol $\mathit{\Pi}_{\mathsf{mq\text{-}rpmt}}$

- Task ID: 64-bit long.
- Protocol ID: 32-bit integer.
- Step ID: 32-bit integer.
- Extra Info: 64-bit long[12].
- Sender ID: 32-bit integer.
- Receiver ID: 32-bit integer.
- Payload: List¡byte[]¿ supporting arbitrary size.

Protocol Buffers is Google's language-neutral, platform-neutral, extensible mechanism for serializing structured data and are fully compatible with Netty. Note that Protocol Buffers introduce lengths of each byte array in Payload Bytes in its serialization. Therefore, the actual communication costs are higher than the theoretical communication costs. The results reported in our setting would reflect the actual costs when deploying protocols in real situations. The detailed protocol buffer definition is as follows:

```
‘‘‘protobuf
syntax = "proto3";
message DataPacketProto {
    // the package contains head and
    // payload, separately defined by
    //DataPacketSpecProto and PacketProto.
    HeaderProto headerProto = 1;
    PayloadProto payloadProto = 2;
    // head definition
    message HeaderProto {
        // task ID
        int64 taskId = 1;
        // protocol ID
        int32 ptoId = 2;
        // step ID
        int32 stepId = 3;
        // extra information
        int64 extraInfo = 4;
```

[12] Extra information for each step. For example, the current number of AND operations in SKE-2PC LowMC.

```
        // sender ID
        int32 senderId = 5;
        // receiver ID
        int32 receiverId = 6;
    }
    // payload definition
    message PacketProto {
        // repeated means the payload
        //contains an array of byte[]
        repeated bytes payloadBytes = 1;
    }
}
'''
```

G.2 JNI Technique

As mentioned in Section 6.2, we use the Java Native Interface (JNI) technique to invoke C/C++ libraries for speeding up performances. These include:

- Bit matrix transpose (used in OT extension and SKE-PSU). We follow the ideas provided by Mischasan[13] and adjust the implementation given in EMP-toolkit[14] to implement bit matrix transpose operations. The bit matrix is represented in the big-endian byte ordering, thus compatible with Java.
- Polynomial operations (used in KRTW and GMRSS). We tried the pure-Java Rings polynomial implementation[15] but found that its efficiency is not acceptable. We instead use the NTL library[16] with GMP library and GF2X library[17] for speeding up the performance. We adjust the polynomial representation to make the results returned from NTL compatible with Rings.
- ECC operations (used in base OT and PKE-PSU). We compared the ECC operation performances via different libraries, including the pure-Java Bouncy Castle[18], the C/C++ Relic[19], and the C/C++ MCL[20]. We found that (at least in our experiment platform) MCL library performs best, especially for the fixed-point multiplication operation. However, the ECC addition operation in Bouncy Castle is faster than MCL in our platforms. Therefore, we adjust the ECC point representation returned from MCL to make it compatible with the ECC point representation in Bouncy Castle to directly use Bouncy Castle to do the addition operations in Java.
- Switching Network programming (used in GMRSS and JSZDG). We used the code base open-sourced by Garimella et al. [GMR+21][21] as a starting point. We replaced the switching node representation from 'int' to 'int8_t' to reduce the memory cost.

G.3 Performance of GMRSS and JSZDG Using Silent OT

We denote GMRSS, JSZDG-R and JSZDG-S schemes with Silent OT by **GMRSS***, **JSZDG-R*** and **JSZDG-S***, respectively. The challenge is that current Silent OT implementations only provide Learning-Parity-with-Noise (LPN) parameters for large COT output sizes but not for small COT output sizes. For example, the Ferrot OT [YWL+20] only provides LPN parameters that can output 10 million COTs. We follow a similar strategy introduced in [YWL+20] to find LPN parameters to output $2^{14}, 2^{16}, 2^{18}, 2^{20}, 2^{22}$ COTs in the regular-index setting, while all known attacks (e.g., Gaussian

[13] https://mischasan.wordpress.com/2011/10/03/the-full-sse2-bit-matrix-transpose-routine/
[14] https://github.com/emp-toolkit/emp-tool/blob/master/emp-tool/utils/block.h
[15] (https://rings.readthedocs.io/
[16] https://libntl.org/
[17] https://gitlab.inria.fr/gf2x/gf2x
[18] (https://www.bouncycastle.org/java.html
[19] https://github.com/relic-toolkit/relic
[20] https://github.com/herumi/mcl
[21] https://github.com/osu-crypto/PSI-analytics/blob/master/psi_analytics_eurocrypt19/common/benes.cpp

elimination, low-weight parity-check and information set decoding) requires at least 2^{128} arithmetic operations. The parameters are shown in Table 5. We refer readers to see [YWL+20] for details on setting these parameters in Ferret OT.

# of COTs	One-time Setup			Iteration		
	k_0	n_0	t_0	k	n	t
2^{14}	1152	8792	581	1408	25167	1475
2^{16}	2304	12832	409	4352	78354	1411
2^{18}	2432	27451	872	15232	289584	1526
2^{20}	4864	71040	1131	55680	1119616	1536
2^{22}	12160	237343	1508	218880	4431616	1536
2^{24}	43776	882063	1533	860160	17658880	1536

Table 5: Extended Parameters in Ferret OT [YWL+20]

We report the performance of these schemes in Table 6. Taking $n = 2^{20}$ and $T = 1$ as an example, we show the variation of the running time with the bandwidth of three schemes and their silent OT version in Figure 15. As shown in Table 6, we find that the communication of silent OT version is about 60% of the original schemes, while the running time is slower in the high bandwidth setting. This is due to the characteristics of silent OT, that is, the computational complexity is higher, but the communication is lower than the IKNP OT extension. As shown in Figure 15, two lines meet between 10Mbps and 100Mbps in all three schemes. Therefore, we could consider substituting the IKNP OT extension with silent OT in these schemes as a trade-off between communication and computation.

n	Protocol	Comm. (MB)					Running time (s)															
		$\mathcal{R}$		$\mathcal{S}$		total	LAN				1Gbps				100Mbps				10Mbps			
		setup	online	setup	online		$T = 1$		$T = 8$		$T = 1$		$T = 8$		$T = 1$		$T = 8$		$T = 1$		$T = 8$	
							setup	online	setup	online	setup	online	setup	online	setup	online	setup	online	setup	online	setup	online
2^{14}	GMRSS	0.02	5.89	0.02	7.96	13.85	0.1	1.01	0.04	0.42	0.66	1.96	0.46	1.28	1	3.53	0.91	2.97	1.06	14.44	0.93	13.97
	GMRSS*	0.24	1.82	0.22	8.11	9.93	0.18	1.47	0.09	0.77	1.14	2.36	0.99	1.73	2.11	3.86	1.73	3.19	2.49	11.64	2.03	11.17
	JSZDG_R	0.01	4.65	0.01	5.63	10.28	0.07	1.81	0.02	0.52	0.27	2.65	0.23	1.34	0.49	4.19	0.41	2.66	0.45	12.08	0.37	10.63
	JSZDG_R*	0.19	0.98	0.21	5.78	6.75	0.15	1.96	0.07	0.76	0.85	3.38	0.72	1.51	1.3	4.39	1.22	2.83	1.67	9.64	1.5	8.1
	JSZDG_S	0.01	9.41	0.01	10.64	20.04	0.07	2.17	0.03	0.69	0.37	3.5	0.31	1.85	0.64	5.53	0.55	4.15	0.62	21.7	0.57	20.27
	JSZDG_S*	0.26	5.88	00.26	7.11	12.99	0.16	2.7	0.08	1.19	0.91	3.71	0.84	2.19	1.7	5.6	1.4	4.08	1.94	15.82	1.79	14.85
2^{16}	GMRSS	0.02	25.95	0.02	34.11	60.06	0.11	4.79	0.04	1.95	0.64	6.61	0.48	4.25	1.11	12.67	0.92	9.78	1.04	60.75	0.94	57.5
	GMRSS*	0.38	6.66	0.37	33.45	40.11	0.25	5.93	0.13	3.06	1.36	8.56	1.02	5.24	2.11	12.07	1.79	9.39	2.57	45.6	2.38	42.34
	JSZDG_R	0.01	20.75	0.01	24.74	45.49	0.07	7.5	0.02	2.25	0.3	9.29	0.2	4.45	0.44	13.78	0.4	8.58	0.47	49.41	0.42	44.58
	JSZDG_R*	0.33	3.18	0.34	24.29	27.47	0.23	9.74	0.12	3.64	0.93	11.54	0.68	5.58	1.4	14.69	1.31	8.48	1.97	37.21	1.82	30.8
	JSZDG_S	0.01	42.02	0.01	47.43	89.45	0.07	9.4	0.02	3.49	0.39	12.03	0.33	6.72	0.63	20.45	0.57	14.65	0.66	92.22	0.53	86.25
	JSZDG_S*	0.53	24.72	0.53	30.12	54.84	0.44	13.19	0.18	6.24	1.02	15.04	0.96	8.57	1.83	21.63	1.58	14.11	2.44	66.34	2.64	60.4
2^{18}	GMRSS	0.02	113.7	0.02	145.11	258.81	0.13	20.74	0.03	9.8	0.58	28.62	0.55	16.63	1.09	49.68	0.93	38.82	1.03	251.84	0.97	243.63
	GMRSS*	0.72	26.04	0.6	140.99	167.03	0.53	30.33	0.38	15.55	1.61	38.04	1.33	21.73	2.55	50.81	2.44	36.67	3.66	184.55	3	172.48
	JSZDG_R	0.01	92.67	0.01	107.89	200.56	0.07	41.15	0.03	10.71	0.25	43.17	0.21	16.84	0.42	64.06	0.4	33.8	0.53	221.27	0.39	191.2
	JSZDG_R*	0.72	13.04	0.56	104.56	117.59	0.49	58.53	0.34	18.6	1.21	62.7	1.19	23.36	1.91	73.96	1.81	33.14	2.83	169.16	2.52	130.23
	JSZDG_S	0.01	185.73	0.01	212.56	398.29	0.08	47.88	0.03	17.2	0.44	56.28	0.31	28.3	0.63	90.87	0.56	63.01	0.59	417.5	0.58	379.63
	JSZDG_S*	1.02	106.34	1.01	133.17	239.5	1	77.03	0.62	31.07	1.97	73.65	1.86	38	3.03	104.8	2.34	60.38	4.04	293.68	3.69	258.33
2^{20}	GMRSS	0.02	493.2	0.02	615.9	1109.1	0.11	100.48	0.04	48.53	0.62	119.98	0.51	75.76	1.11	207.83	0.95	164.25	1.09	1074.33	0.95	1030.3
	GMRSS*	1.19	103.22	0.77	598.21	701.43	1.23	144.79	0.72	71.89	2.19	162.16	1.85	92.65	3.07	212.15	2.76	149.52	4.6	779.92	4.1	718.97
	JSZDG_R	0.01	405.53	0.01	467.26	872.79	0.08	173.07	0.04	54.41	0.48	184.63	0.2	73.28	0.47	266.51	0.73	146.13	0.47	941.5	0.72	825.16
	JSZDG_R*	1.14	51.3	0.68	452.75	504.05	0.89	273.25	0.82	85.62	1.64	281.78	1.73	97.24	2.27	325.41	2.22	139.31	3.75	737.4	3.54	550.06
	JSZDG_S	0.01	813.5	0.01	929.78	1743.29	0.08	217.9	0.05	89.18	0.33	249.1	0.28	129.09	0.66	393.02	0.56	269.69	0.67	1820.49	0.56	1653.72
	JSZDG_S*	1.41	460.14	1.4	576.42	1036.56	1.68	369.11	1.35	145.83	2.98	330.48	2.21	167.1	3.4	474.78	3.37	255.16	5.52	1276.32	4.98	1111.8

Table 6: Communication cost (in MB) and running time (in seconds) comparing GMRSS, JSZDG protocols and their silent OT version. The LAN network has 10 Gbps bandwidth and 0.2 ms RTT latency. The 100Mbps and 10Mbps network have 80ms RTT latency, while the 1Gbps network has 40ms RTT. Communication cost of $\mathcal{S}/\mathcal{R}$ indicates the outgoing communication from $\mathcal{S}/\mathcal{R}$ to the other party.

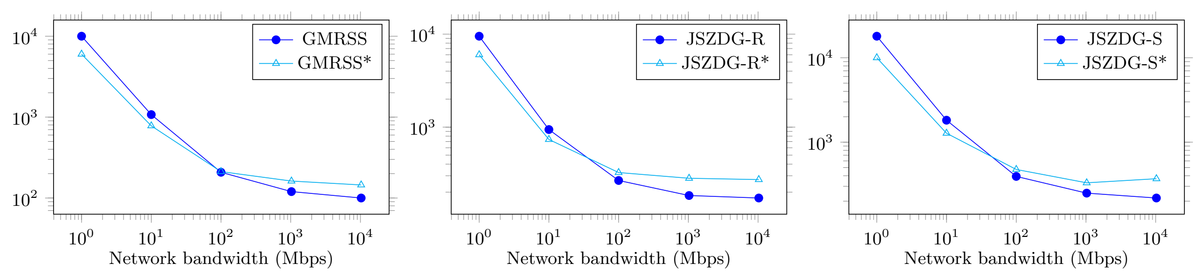

Fig. 15: Decline of running time (in seconds) on increasing network bandwidth for GMRSS, JSZDG-R and JSZDG-S compared with their silent OT version. Both x and y-axis are in log scale. The set size $n = 2^{20}$ and the number of threads $T = 1$.

G.4 The Costs of Triple Generation of SKE-PSU

Here we report the costs of triple generation in our SKE-PSU. We use Ferret OT [YWL+20] and the techniques introduced in [ALSZ13]. Note that when the set size is relatively large, the total number of needed Boolean triples can be larger than 2^{24}, requiring more than 2^{24} OTs, while the maximum number of OTs supported for our Ferret OT parameter is 2^{24}. Our strategy is to generate Boolean triples (OTs) on-the-fly if the total number of needed Boolean triples is beyond 2^{24}. In this way, the computation and communication costs of *Boolean triple generation* are included in Tables 7.

n	Comm. (MB)			Running time (s)							
	$\mathcal{R}$	$\mathcal{S}$	total	LAN		1Gbps		100Mbps		10Mbps	
				$T = 1$	$T = 8$	$T = 1$	$T = 8$	$T = 1$	$T = 8$	$T = 1$	$T = 8$
2^{14}	3.16	3.16	6.32	51.51	29.4	53.8	31.01	56.05	32.21	60.06	32.22
2^{16}	5.1	5.1	10.2	163.73	93.81	167.17	95.9	169.78	98.92	175.54	108.09
2^{18}	11	11	22	574.85	320	581.07	330.29	583.54	329.78	600.64	353.61
2^{20}	34.21	34.21	68.42	2214.98	1210.69	2230.5	1233.36	2236.57	1262.55	2274.38	1302.01

Table 7: Communication cost (in MB) and running time (in seconds) of triple generation stage in SKE-PSU. The LAN network has 10 Gbps bandwidth and 0.2 ms RTT latency. The 100Mbps and 10Mbps network have 80ms RTT latency, while the 1Gbps network has 40ms RTT. Communication cost of $\mathcal{S}/\mathcal{R}$ indicates the outgoing communication from $\mathcal{S}/\mathcal{R}$ to the other party.

As showed in Table 7, the communication cost in the triple generation stage is small. And thus the running time under different bandwidth is almost the same. When increases from $T = 1$ to 8, the running time of triple generation improves about 1.8×.

H Experiment Results of PSU Applications

H.1 IP Blacklist Aggregation

IP blacklist aggregation [HLS+16, RMY20] is a direct application of PSU, in which the input sets of both parties are their respective IP blacklists, and the output is the union blacklist. Next, we run PSU to fulfill the IP blacklist aggregation task, demonstrating its the concrete efficiency in a realistic scenario.

The input IP blacklist comes from the public BlackIP project[22]. This project provides two black IP sets, namely blackip (with 3, 176, 636 IPv4 addresses) and oldip (with 2, 514, 551 IPv4 addresses). We assume $\mathcal{S}$ maintains the blackip set and $\mathcal{R}$ maintains the oldip set. Note that an IPv4 address is represented as a 32-bit length binary string, while the input sets we tested in Section 6 contain 128-bit length binary strings. The experiment results are shown in Table 8.

[22] https://github.com/maravento/blackip

Protocols	Comm. (MB)			Running time (s)	
	$\mathcal{R}$	$\mathcal{S}$	Total	LAN	WAN
KRTW	1082.03	9613.29	10695.32	232.99	1406.25
GMRSS	1541.4	1846.69	3388.09	207.86	490.24
JSZDG-R	1044.6	713.38	1757.98	145.55	324.84
SKE-PSU	594.91	632.3	1227.21	74.23	170.62
PKE-PSU	212.06	254.23	466.29	197.42	229.76
PKE-PSU*	405.95	453.75	859.7	125.06	179.51

Table 8: Communication cost (in MB) and running time (in seconds) comparing our protocols to KRTW GMRSS, and JSZDG-R in the applications of IP blacklist aggregation. The input size of $\mathcal{R}$ and $\mathcal{S}$ are 2514551 and 3176636 respectively. The LAN network has 10 Gbps bandwidth and 0.2 ms RTT latency and the WAN network has 100 Mbps bandwidth and 80 ms RTT latency. Communication cost of $\mathcal{S}/\mathcal{R}$ indicates the outgoing communication from $\mathcal{S}/\mathcal{R}$ to the other party. The number of thread $T = 8$. The best protocol in this setting is marked in blue.

In this application, the input set size of both partis are larger than 2^{20}. Nevertheless, we find that the experimental results are consistent with the experiment in Section 6: our PKE-PSU has the lowest communication, and our SKE-PSU performs best in running time. In addition, all of our PSU protocols have a better communication cost ratio. The main reason is that the communication complexity of our protocols is linear $O(n)$, while other protocols are $O(n \log n)$.

H.2 Private ID

Private ID (PID) [BKM+20, GMR+21] is also an important PSU application. The definition of Private ID functionality $\mathcal{F}_{\mathsf{pid}}$ is given in Figure 16. Garimella et al. [GMR+21] proposed a framework of constructing PID protocol from OPRF and PSU. Their main idea is as follows, the parties execute two 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 identifiers are defined as $R(x) := F_{k_A}(x) \oplus F_{k_B}(x)$. The parties compute the identifiers of the items in their set and finally they execute a PSU protocol to obtain the whole identifier set. We give this basic[23] PID protocol in Figure 17.

Parameters: Two parties: Alice and Bob. Number of items n for the Alice and Bob; length of identifiers l.
Functionality:

- Wait for input $X = \{x_1, \dots, x_n\} \subset \{0,1\}^*$ from Alice.
- Wait for input $Y = \{y_1, \dots, y_n\} \subset \{0,1\}^*$ from Bob.
- For every $z \in X \cup Y$, choose a random identifier $R(z) \xleftarrow{\text{R}} \{0,1\}^l$.
- Define $R^* := \{R(z) | z \in X \cup Y\}$.
- Give output $(R^*, R(x_1), \dots, R(x_n))$ to Alice.
- Give output $(R^*, R(y_1), \dots, R(y_n))$ to Bob.

Fig. 16: Private ID Functionality $\mathcal{F}_{\mathsf{pid}}$.

The bottleneck of the above PID protocol is exactly the underlining PSU protocol. As shown in the experiment of [GMR+21], about 90% of the PID running-times is on the PSU. Therefore, we believe that replacing it with our PSU, the resulting PID protocol will be more efficient. The experiment results are shown in Table 9. The notions of SKE-PID, PKE-PID, PKE-PID* denote that the PSU protocol is replaced by SKE-PSU, PKE-PSU and PKE-PSU* in the PID protocol 17.

[23] For better efficiency, [GMR+21] further introduces a “sloppy OPRF” technique to generate identifiers. 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. See [GMR+21] for more details

Parameters:

– Two parties: Alice and Bob.
– Ideal $\mathcal{F}_{\mathsf{oprf}}$ and $\mathcal{F}_{\mathsf{psu}}$ primitives.

Input of Alice: $X = \{x_1, \ldots, x_n\} \subset \{0,1\}^*$.
Input of Bob: $Y = \{y_1, \ldots, y_n\} \subset \{0,1\}^*$.
Protocol:

1. **(OPRF)** Alice and Bob invoke the OPRF functionality $\mathcal{F}_{\mathsf{oprf}}$. 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 OPRF functionality $\mathcal{F}_{\mathsf{oprf}}$. 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. **(Identifiers computation)** For $i \in [n]$, Alice computes $R^A(x_i) := F_{k_A}(x_i) \oplus F_{k_B}(x_i)$.
4. For $i \in [n]$, Bob computes $R^B(y_i) := F_{k_A}(y_i) \oplus F_{k_B}(y_i)$.
5. **(Union)** Alice and Bob invoke the PSU functionality $\mathcal{F}_{\mathsf{psu}}$ with input $\{R^A(x)|x \in X\}$ and $\{R^B(y)|y \in Y\}$ respectively. As a result, they obtain output R^* and output the $(R^*, \{R^A(x_i)\}_{i\in[n]})$ and $(R^*, \{R^B(y_i)\}_{i\in[n]})$ respectively.

Fig. 17: Basic Private ID Protocol $\mathit{\Pi}_{\mathsf{pid}}$

n	Protocols	Comm. (MB)			Running time (s)	
		Alice	Bob	Total	LAN	WAN
2^{14}	GMRSS-PID	9.31	11.58	20.89	0.7	4.11
	SKE-PID	6.58	6.98	13.56	0.58	14.3
	PKE-PID	4.58	5.21	9.79	1.37	3.13
	PKE-PID*	5.58	6.51	12.09	0.86	3.07
2^{16}	GMRSS-PID	39.49	48.41	87.9	3.18	13.34
	SKE-PID	26.14	27.71	53.85	3.06	19.25
	PKE-PID	18.15	20.67	38.81	5.65	10.2
	PKE-PID*	22.15	25.87	48.02	3.81	9.02
2^{18}	GMRSS-PID	174.82	209.46	384.28	14.97	53.78
	SKE-PID	111.31	117.86	229.17	11.6	42.77
	PKE-PID	79.47	89.8	169.27	25.58	40.27
	PKE-PID*	95.47	110.6	206.07	17.17	35.17
2^{20}	GMRSS-PID	733.61	869.21	1602.82	75.09	222.9
	SKE-PID	440.68	466.86	907.54	51.43	133.2
	PKE-PID	313.81	355.11	668.92	106.13	158.38
	PKE-PID*	377.81	438.32	816.12	74.67	135.72

Table 9: Communication cost (in MB) and running time (in seconds) comparing our PIDs with GMRSS-PID. The LAN network has 10 Gbps bandwidth and 0.2 ms RTT latency and the WAN network has 100 Mbps bandwidth and 80 ms RTT latency. Communication cost of *Alice*/*Bob* indicates the outgoing communication from *Alice*/*Bob* to the other party. The number of thread $T = 8$. The best protocol within a setting is marked in blue.

As shown in Table 9, the experimental results of PID protocols are consistent with those of PSU: our PKE-PID has the lowest communication among all protocols; the SKE-PID performs best in the LAN setting; the PKE-PID* performs better in the WAN setting due to the loewer communication cost.

USENIX'23 Artifact Appendix: Linear Private Set Union from Multi-Query Reverse Private Membership Test

Cong Zhang[1,2], Yu Chen[3,4,5] (✉), Weiran Liu[6], Min Zhang[3,4,5] and Dongdai Lin[1,2]

[1]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China*
{zhangcong,ddlin}@iie.ac.cn
[3]*School of Cyber Science and Technology, Shandong University, Qingdao 266237, China*
[4]*State Key Laboratory of Cryptology, P.O. Box 5159, Beijing 100878, China*
[5]*Key Laboratory of Cryptologic Technology and Information Security, Ministry of Education, Shandong University, Qingdao 266237, China*
yuchen.prc@gmail.com, zm_min@mail.sdu.edu.cn
[6]*Alibaba Group*
weiran.lwr@alibaba-inc.com

A Artifact Appendix

A.1 Abstract

We introduce our open-source project mpc4j, an efficient and easy-to-use Secure Multi-Party Computation (MPC) library mainly written in Java. Package psu in mpc4j-s2pc-pso of mpc4j contains the implementations, along with configurations needed to replicate our experiments from Section 6. In particular, our artifact supports running and comparing Private Set Union (PSU) protocols with element set sizes up to 2^{20} on machines having 128GB memory. We also provide guidelines for installing dependencies and compiling native libraries needed by mpc4j on different platforms, including x86_64 MacBook, MacBook with M1 chip, Ubuntu 20.04, and CentOS 8. The project is licensed under Apache License 2.0. The source code is available online at `https://github.com/alibaba-edu/mpc4j`. The stable version for the artifact evaluation is available at `https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.4`.

In this artifact appendix, we first introduce the minimal hardware and software requirements to get performance reports shown in our paper using mpc4j. Then, we introduce how to install and run mpc4j on different platforms. We note that there are some performance gaps between different platforms, and having complete comparisons for different protocols is very challenging. Aside from that, mpc4j still tries to provide a library for having relatively unified comparisons. We welcome suggestions and performance reports on other platforms with future reproducibility.

A.2 Description & Requirements

We introduce our open-source project mpc4j (Multi-Party Computation for Java), an efficient and easy-to-use Secure Multi-Party Computation (MPC) library mainly written in Java. mpc4j aims to provide an academic library for researchers to study and develop MPC and related protocols in a unified manner. As mpc4j tries to provide state-of-the-art MPC implementations, researchers could leverage the library to have quick and unified comparisons between the proposed and existing protocols.

Package psu in mpc4j-s2pc-pso of mpc4j contains the implementations, along with configurations needed to replicate our experiments from Section 6. Existing Private Set Union (PSU) implementations are under different MPC frameworks and different experimental settings. After carefully studying existing open-source codes, we fully re-implement exisiting PSU protocols and their underlying basic protocols using Java. Evaluators can test PSU protocols on mpc4j by simply using different configuration files. All experiment results shown in Section 6 of our paper are obtained by running mpc4j.

Evaluators can compile and run mpc4j on different 64-bit platforms. We provide guidelines for installing dependencies and compiling native libraries needed by mpc4j on different platforms, including x86_64 MacBook, MacBook with M1 chip, Ubuntu 20.04, and CentOS 8. Note that successfully running all PSU experiments with large element size (i.e., $n = 2^{20}$) requires 128GB RAM. We run our experiments on a single Intel Core i9-9900K with 3.6GHz and 128GB RAM. We note that there are some performance gaps between different platforms. We welcome suggestions and performance reports on other platforms with future reproducibility.

In the full version of our paper, we further provide experiment results on two PSU applications, namely IP blacklist aggregation and Private ID. The related source code has been merged into version v1.0.5[1].

[1]`https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.5`

A.2.1 How to access

mpc4j is available online on GitHub at https://github.com/alibaba-edu/mpc4j. Evaluators can visit the stable version v1.0.4 (https://github.com/alibaba-edu/mpc4j/releases/tag/v1.0.4) to reproduce the experiment results shown in the paper.

A.2.2 Hardware dependencies

mpc4j currently support 64-bit macOS, Ubuntu, and CentOS systems. Evaluators may meet errors when compiling mpc4j on a 32-bit or less system. The reason is that mpc4j uses some 64-bit Single instruction, multiple data (SIMD) operations.

A.2.3 Software dependencies

mpc4j leverages native C/C++ codes to speed up cryptographic operations. The native codes and Java codes are interacted by the Java Native Interface (JNI) technique.

We separate native C/C++ codes into two modules, namely mpc4j-native-tool and mpc4j-native-fhe. mpc4j-native-tool contains native codes for basic cryptographic operations, while mpc4j-native-fhe contains native codes for Fully Homomorphic Encryption (FHE) using SEAL[2]. All basic cryptographic operations in mpc4j-native-tool have alternative pure-Java implementations in mpc4j with the same functionalities and the same data representations. Note that if evaluators only run mpc4j for PSU, there is no need to install SEAL and compile mpc4j-native-fhe. mpc4j-native-tool relies on the following C/C++ libraries:

- GMP (https://gmplib.org/): An efficient library for operations with arbitrary precision integers, rationals, and floating-point numbers.
- NTL (https://libntl.org/): A high-performance, portable C++ library providing data structures and algorithms for manipulating signed, arbitrary length integers and for vectors, matrices, and polynomials over the integers and over finite fields, developed by Victor Shoup (https://shoup.net/). Note that one can further introduce GF2X (https://gitlab.inria.fr/gf2x/gf2x) for more efficient operations in a Galois Field. However, since the installation procedure for GF2X is rather complicated, we use NTL by default.
- MCL (https://github.com/herumi/mcl): A portable and fast pairing-based cryptography library. MCL also includes fast Elliptic Curve implementations, especially the optimized implementation for the elliptic curve secp256k1.
- libsodium (https://doc.libsodium.org): A modern, easy-to-use software library for encryption, decryption, signatures, password hashing, and more. libsodium includes efficient implementations for the elliptic curve Curve25519 with APIs for X25519 and Ed25519.
- OpenSSL (https://www.openssl.org/): a robust, commercial-grade, full-featured toolkit for general-purpose cryptography and secure communication. OpenSSL includes many efficient cryptographic primitive implementations.

[2]https://github.com/microsoft/SEAL

A.3 Set-up

A.3.1 Installation

Installing mpc4j-native-tool might be a bit complicated for ones who are not that familiar with Unix-like systems, since the procedures differ across platforms. The documentation (README.md) in package mpc4j-native-tool provides instructions for installing mpc4j-native-tool on macOS (x86_64 / aarch64), Ubuntu, and CentOS, respectively.

A.3.2 Basic Test

We develop mpc4j using Intellij IDEA (https://www.jetbrains.com/idea/) and CLion (https://www.jetbrains.com/clion/). After successfully compiling mpc4j-native-tool (Please see readme.md in these modules for more details on how to compile them), evaluators only need the community version of Intellij IDEA to run all basic tests.

Evaluators need to configure IDEA with the following procedures so that IDEA can link to the complied mpc4j-native-tool native libraries.

1. Open "Run->Edit Configurations..."
2. Open "Edit Configuration templates..."
3. Select "JUnit".
4. Add the following command into "VM Options": -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH.

After that, evaluators can run tests of any submodule by pressing the green arrows showing on the left of the source code in test packages. See Section **Demonstration** of readme.md in mpc4j on details for running the tests.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): In our paper, we claimed that we fully re-implement state-of-the-art PSU protocols and their underlying basic protocols using Java. This can be verified by running basic tests in psu (See Section A.3.2 for details), or running experiments with different configuration files (See Section A.4.2 for details).

(C2): In our paper, we claimed that although there is some performance gap, most basic operations in Java and C/C++ have similar performances. This can be verified by running all efficient tests in mpc4j-common-tool (test classes with names end with "EfficiencyTest"). For example, try running "PrpEfficiencyTest" in the package edu.alibaba.mpc4j.common.tool.crypto.prp of the submodule mpc4j-common-tool, evaluators can see the performance comparisons between using AES provided by Java and by AES-NI invoked with JNI.

A.4.2 Experiments

(E1): *[Generate jar file] [5 human-minutes + 5 compute-minutes]: Generate mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar containing the main function entry.*
How to: On the charms bar of IDEA, evaluators can find a button with name "Maven". Press that botton, double-click "mpc4j -> Lifecycle -> package", IDEA would automatically compile and generate mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar containing the main function entry.
Preparation: Evaluators need to successfully running basic tests before generating the jar file.
Execution: Just double-click "mpc4j -> Lifecycle -> package".
Results: The generated file would be located in "mpc4j/mpc4j-s2pc-pso/target".

(E2): *[(optimal) Config network settings] [5 human-minutes + 1 compute-minute]: Config network settings using tc.*
How to: Open a terminal, and execute the following command: "tc qdisc add dev lo root netem rate 10Mbit latency 80ms". Then, the local network is configured as 10Mbit bandwidth with 80ms latency. Evaluators can try other network settings with other parameters, e.g., 100Mbit/80ms, 1Gbit/40ms, 10Gbit/0.02ms.
Preparation: None
Results: Execute "sudo tc qdisc show dev lo" to see if the network is configured correctly.

(E3): *[Run experiments] [10 human-minutes + 5 compute-hour]: Run experiments using different configuration files.*
How to: Open two terminals, one for the PSU server and one for the PSU client. Switch to the dictionary where mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar located (Evaluators can also copy the generated jar file to other dictionaries). For the server's terminal, execute "java -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 -jar mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar CONFIG_SERVER_FILE.txt". For the client's terminal, execute "java -Djava.library.path=/YOUR_ABS_NATIVE_LIB_PATH -Djava.util.concurrent.ForkJoinPool.common.parallelism=8 -jar mpc4j-s2pc-pso-1.0.4-jar-with-dependencies.jar CONFIG_CLIENT_FILE.txt". The corresponding server/client configuration files are in "mpc4j-s2pc-pso/conf/psu". Note that evaluators must first run server and then run client.
Preparation: None.
Note: It would take a long time to run if the network has limited bandwidth, long latency, and/or a large set size. See the performance results of our paper to estimate the total running time. Evaluators may find that the setup of SKE-PSU time is quite different from the result presented in Table 3 of our paper. This is because in the paper, we assume Boolean multiplication triples are pre-computed offline and stored locally in a temporary file. Therefore, the setup phase only contains loading Boolean multiplication triples into the memory. In our artifact, we dynamically generate Boolean multiplication triples in the setup phase using silent Oblivious Transfer techniques. In the full version of the paper, we provide the triple generation costs for SKE-PSU, which would be similar to the costs in the setup phase evaluators obtained using the artifact.
Results: Java would run the experiments and generate the performance reports under the current dictionary.

A.5 Notes on Reusability

Evaluators can check and modify server/client configuration files to change IP addresses, port numbers, the element byte length used for PSU. We also provide other configuration examples (marked with #) for specific PSU protocols.

A.6 Version

Based on the LaTeX template for Artifact Evaluation V20220926. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2023/.