Revisiting Shuffle-Based Private Set Unions with Reduced Communication

Jiseung Kim Jeonbuk National University jiseungkim@jbnu.ac.kr Hyung Tae Lee Chung-Ang University hyungtaelee@cau.ac.kr Yongha Son Sungshin Women's University yongha.son@sungshin.ac.kr

ABSTRACT

A Private Set Union (PSU) allows two parties having sets X and Y to securely compute the union $X \cup Y$ while revealing no additional information. Recently, there have been proposed so-called shuffle-based PSU protocols due to Garimella et. al. (PKC'21) and Jia et. al. (USENIX'22). Except a few base oblivious transfers, those proposals are fully based on symmetric key primitives and hence enjoy quite low computation costs. However, they commonly have drawbacks on large communication cost of $O(\ell n \log n)$ with input set size n and $\ell \ge O(\lambda + \log n)$ where λ is a statistical security parameter.

We propose two optimizations for each work that reduce communication cost while maintaining strength in computation cost; the first one optimizes Garimella et. al. to have $O(\ell n + n \log n)$, and the second one optimizes Jia et. al. by reducing the concrete value of ℓ by $\log n$. Concretely, the first (second, resp) optimization provides $3.3 - 3.9 \times (1.7 - 1.8 \times n)$ lower communication input set sizes $n = 2^{16} - 2^{20}$.

We demonstrate by comprehensive analysis and implementation that our optimization leads to better PSU protocol, compared to the state-of-the-art proposal of Zhang et. al. (USENIX'23) as well as previous shuffle-based PSUs. As a concrete amount of improvement, we see 1.4-1.5x speed up for 100Mbps network, and 1.8-2.2x speed up for 10Mbps network on input set sizes $n=2^{16}-2^{20}$.

KEYWORDS

private set union, oblivious key-value store, permute-and-share

1 INTRODUCTION

A (two-party) Private Set Union (PSU) allows two parties having sets X and Y to securely compute the union $X \cup Y$ while revealing no additional information. In particular, each party obtains no knowledge of whether each item is in the intersection $X \cap Y$ or not.

Even a decade after early PSU proposals [11, 18], there was comparably less attention on efficient PSU protocols than private set intersection (PSI). However, starting from the first scalable PSU construction [20], several protocols with enhanced efficiency [12, 17, 31] have sprung up recently.

In this work, we especially focus on what we call *shuffle-based* PSUs [12, 17]. At the current state, shuffle-based PSUs have significant weakness in the communication complexity of $O(n \log n)$ with input set size n, where the most efficient one [31] requires only O(n) communication complexity. Indeed, concrete communication costs differ by almost one order of magnitude; for example, [31] requires only 157MB but [17] requires 1338MB to union $n=2^{20}$ size sets. Meanwhile, compared to the other work [31] that exploits public-key techniques in some parts, shuffle-based PSUs may have an advantage in computation as they are purely based on symmetric-key techniques.

1.1 Our Contribution

Revisit Shuffle-based PSU. We revisit shuffle-based PSUs [12, 17] in modular view in relation with recent techniques for PSI, especially with Oblivious Key-Value Store (OKVS) [13] abstraction and recent advances on Oblivious Transfer (OT) extension [10]. In a nutshell, shuffle-based PSUs commonly require OKVS operations with O(n) items, and additional $O(n \log n)$ OTs. This explains the computational strength of Shuffle-based PSUs over [31] in an objective way; [31] also requires O(n) OKVS operations, but additional O(n) public key operations or O(tn) OTs with huge $t \approx 600 \gg \log n$ for realistic n.

Reduce Communications. We then propose optimized shuffle-based PSUs that reduce communication costs, while maintaining the strength of computational cost. To be precise, the root cause of the huge communication cost of previous shuffle-based PSU is the term $O(\ell n \log n)$, where $\ell \geq \lambda + \log n$ for statistical security parameter λ . In this respect, we propose two independent ideas that optimize previous shuffle-based protocols respectively. The first one optimizes [12] to have $O(\ell n + n \log n)$ communication, at the additional computational cost for $O(\ell n)$ OT. The second one optimizes [17] to have $O((\ell - \log n) \cdot n \log n)$ communication, without any sacrifice on computational cost. Concretely, for $n = 2^{20}$ input size, the first one reduces from 572MB to 144MB, and the second one reduces from 970MB to 533MB.

Implementation and Evaluation. We provide comprehensive performance evaluation, along with implementation results. From this, we argue that our optimizations make shuffle-based PSUs have better concrete performance than linear complexity PSUs [31] and previous shuffle-based PSUs. Precisely, our first proposal that optimizes [12] is the best for a network slower than about 100Mbps, and then our second proposal that optimizes [17] is the best one for the other side; faster than 100Mbps. The absolute amount of improvement is substantial over medium to low networks; 1.4-1.5x speed up for 100Mbps network, and 1.8-2.2x speed up for 10Mbps network on input set sizes $n=2^{16}-2^{20}$. See Section 5.2 for detailed comparisons.

1.2 Technical Overview

Throughout this section, we denote the size n_X and n_y of input sets X and Y respectively, and assume that the Y holder is the receiver. We often assume n_X and n_y are equal to n.

Most recent PSU proposals are based on an abstraction called Reverse Private Membership Test (RPMT) [20], where the sender inputs an element x and the receiver inputs a set Y, and the receiver obtains the membership indicator $\mathbf{1}(x \in Y)$ (1 if $x \in Y$, 0 otherwise). Then the receiver obtains each $x \notin Y$ using oblivious transfer (OT),

where the sender inputs $m_0 = x$ and $m_1 = \bot$ for some dummy element \bot . As all known methods for RPMT takes $O(n_y)$ complexity, the naive PSU that applies RPMT on every $x \in X$ and Y results in $O(n_x n_y) = O(n^2)$ complexity. To enhance efficiency, Kolesnikov et al. [20] further applied a bucketing technique, where each party hashes their items into bins and applies the naive PSU for each bin. They used $O(n/\log n)$ bins that lead to each bin size $O(\log n)$, and hence the overall cost becomes $O(n/\log n \cdot \log^2 n) = O(n \log n)$.

One would naturally consider cuckoo hashing [3], which ensures that each bin has at least one item x, whose underlying idea is to use several hash functions to avoid the collision of bin index. Indeed, there are some PSU protocols that make use of cuckoo hashing, in both directions where the sender performs cuckoo hashing [12] and the receiver performs cuckoo hashing [17]. However, cuckoo hashing alone leads to additional information leakage to the receiver other than $X \cup Y$. The fundamental reason for the leakage is the fact that the receiver can correspond each membership indicator $\mathbf{1}(x \in Y)$ to some table index.

In this regard, the true novelty of both works [12, 17] lies on the idea of *shuffle* the receiver-side table in order to break the correspondence with $\mathbf{1}(x \in Y)$. The formal notion of shuffle is abstracted by *permute-and-share* functionality (P&S) [8, 23], where the sender inputs a permutation π and the receiver inputs a vector \mathbf{t} and two parties obtain secret-shared $\pi(\mathbf{t})$. The existing protocol [23] realizing P&S takes $O(n \log n)$ computation cost for a vector length n, and the communication cost further depends on the bit-length of each vector item ℓ that leads to $O(\ell \cdot n \log n)$ -bit communication.

Below we revisit both sender-cuckoo and receiver-cuckoo constructions in a modular view. Then we point out the reason for huge communication and present our ideas for communication reduction.

1.2.1 Previous Sender Cuckoo Protocol [12]. We start with an abstraction of the RPMT construction of Kolesnikov et al. [20]. In the first step, two parties convert the membership problem $x \in Y$ into the equality test problem $t_s = t_r$, using oblivious programmable PRF (OPPRF) [19, 26] functionality. Roughly, the receiver sets a random function $F(\cdot)$ such that F(y) = t for every $y \in Y$ with some random value t_r , and the sender obtains an evaluation $t_s = F(x)$. Note that $t_s = t_r$ if and only if $x \in Y$ with high probability. Then as the second part, two parties privately check the equality $t_s = t_r$ to let the receiver knows $\mathbf{1}(t_s = t_r)$.

An (insecure) cuckoo hashing-based optimization is as follows. The sender performs cuckoo hashing to have CT_X of size $\beta = O(n_X)$ and the receiver performs simple hashing to have ST_Y . Denoting $x_i := CT_X[i]$ and $Y_i := ST_Y[i]$, the goal is to let the receiver knows $\mathbf{1}(x_i \in Y_i)$ for each bin index i. In this case, OPPRF can be batched into only one call with whole ST_Y of size $O(n_y)$ by programming F by $F(Y_i) = t_{r,i}$. After (batched) OPPRF, the sender and the receiver respectively obtain \mathbf{t}_S and \mathbf{t}_T of length β such that $\mathbf{1}(x_i \in Y_i) = \mathbf{1}(t_{S,i} = t_{r,i})$. Then two parties engage private equality test (PEqT) protocol so that the receiver knows the vector of membership indicators $\mathbf{1}(x_i \in Y_i)$.

This cuckoo hashing version really makes the total complexity $O(n_X + n_y) = O(n)$. However, [20] pointed out the information leakage of this protocol; the receiver obtains $x_i \in X \setminus Y$ in the end but further knows that the bin index i where the item x_i is mapped to. Since the bin index of x_i is dependent on the entire set X, the view of the receiver cannot be simulated without the knowledge of X, which can be understood as information leakage.

Later, Garimella et al.[12] resolves this issue by inserting P&S phase to convert t_r into an additive share of $\pi(t_r)$, say s and $p_r := s \oplus \pi(t_r)$, while π is only known to the sender. The sender then can compute $p_s := s \oplus \pi(t_s)$ by itself, since it knows s, π and s. Two parties perform PEqT on inputs p_s and p_r , so that the receiver ends with membership indicators $1(x_i \in Y_i)$ in permuted order. This prevents the receiver to correspond the received item x with the bin index, which solves the problem of the cuckoo hashing only protocol. Figure 1a illustrates the summary. As the sender knows the order of $1(x_i \in Y_i)$, it can send the corresponding item x_i using OT.

The P&S target vector \mathbf{t}_r has length $\beta = O(n_X)$, and hence it takes $O(\beta \log \beta) = O(n_X \log n_X)$ complexity, this increases the total complexity by $O(n_X \log n_X)$. Even worse, the communication cost also depends on the bit-length ℓ of OPPRF output, which is taken $\ell \approx \lambda + \log n_X$ to prevent false positive. This is the main reason for the huge communication cost of [12].

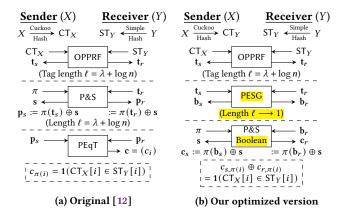


Figure 1: Our sender-cuckoo optimization

1.2.2 Our Sender Cuckoo Protocol. The starting point of our optimization is to switch the execution order of the shuffling and the private equality test, in order to compress the input of P&S to a Boolean vector. However, a naive switching is definitely insecure; the receiver knows the membership indicator in hash table order right after private equality test, and the information leakage pointed out in [20] occurs again.

To remedy this, we change the intermediate PEqT phase into the private equality share generation (PESG) phase that generates Boolean shares of $(x_i \in Y_i)$. Clearly the resulting protocol becomes secure, because each membership indicator is not revealed to the receiver because it is secret-shared, although the order remains unchanged. Also remark that this change from PEqT to PESG incurs only a small overhead in a cost respect, thanks to recent advances in OT extension [10, 30] that significantly reduces the communication cost of GMW protocol.

¹In fact, [17] also proposed another protocol where the sender performs cuckoo hashing, but [12] version is much efficient so we only focus on the receiver-cuckoo version in this paper.

After generating Boolean shares of $\mathbf{b} := (x_i \in Y_i)$, we can convert Boolean shares of \mathbf{b} into Boolean shares of $\mathbf{c} = \pi(\mathbf{b})$ using P&S similar to the [12], but notably the input bit-length of P&S drops to 1 (from ℓ) now. This significantly drops the previously heaviest part P&S of $O(\ell \cdot n_X \log n_X)$ -bit communication to $O(n_X \log n_X)$. Finally, the sender sends its share to the receiver who finally obtains membership indicators in randomly permuted order, which is exactly the same situation with [12]. Figure 1b illustrates the summary.

1.2.3 Previous Receiver Cuckoo Protocol [17]. It suffices to overview the RPMT construction of Jia et al. [17], which are illustrated as Figure 2a. First the receiver performs cuckoo hashing on Y to have CT_Y of size $\beta = O(n_y)$ using γ hash functions $h_1, \cdots, h_\gamma : \{0, 1\}^* \to [\beta]$. Two parties engage P&S to have an additive share of $\pi(CT_Y)$, say \mathbf{s} to the sender and $\mathbf{p} := \pi(CT_Y) \oplus \mathbf{s}$ to the receiver. Their main observation is, when defining $I_X := \{x \oplus s_{id_i} : id_i = \pi(h_i(x))\}_{i \in [\gamma]}$ for each $x \in X$, it holds that

$$I_X \cap \{p_i\}_{i \in [\beta]} \neq \emptyset \Longleftrightarrow x \in Y \tag{1}$$

with high probability. To securely exploit this observation, two parties first execute OPRF so that the sender obtains PRF key k and the receiver obtains $\{F_k(p_i)\}_{i\in[\beta]}$. Then the sender sends $F_k(I_x)$, so that the receiver determines $\mathbf{1}(x\in Y)$ using eq. (1). It is easy to see that shuffling is essential for security. If the same methodology is applied without shuffle, the receiver knows the exact cuckoo hash index i where x intersects with \mathbf{t}_y , which lets the receiver conclude $t_y[i]$ is also in the sender set X. This protocol also suffers from the large communication cost owing to P&S. By denoting the original item length by ℓ_0 , the first P&S call takes $O(\ell_0 \cdot n_y \log n_y)$ -bit communication. Several PSU works including [17] assumes $\ell_0 = 128$, which results in devastating communication cost.

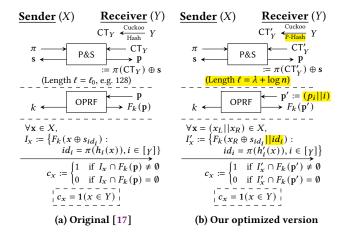


Figure 2: Our receiver-cuckoo optimization

1.2.4 Our Receiver Cuckoo Protocol. As a starting observation, the RPMT result $\mathbf{1}(x \in Y)$ remains same even if the original items are hashed into shorter string. The hash output length ℓ_1 is set to avoid hash collision, and roughly $\ell_1 \approx \lambda + \log n_x n_y$ ensures failure probability less than $2^{-\lambda}$. This readily decreases communication cost for P&S, as the item length drops from $\ell_0 = 128$ to 80 for $n = 2^{20}$ and $\lambda = 40$.

One may easily think of permutation-based hashing (phashing) [3, 25] in order to further reduces the item length by $\log \beta \approx \log n_y$ bits. Instead of a naive hashing method that inserts an item x into the bin of index h(x), the phashing represents $x = x_L || x_R$ with $|x_L| \approx \log \beta$, and inserts only x_R into the bin of index $h'(x) := h(x_R) \oplus x_L$. At first glance, this is seemingly well-compatible with the aforementioned RPMT method, and easily reduces P&S input bit-length by $|x_L| \approx \log \beta$, and the sender has $\mathbf{s} \in \{0,1\}^{\ell_1 - \log n}$ and the receiver has $\mathbf{p} := \pi(\mathsf{CT}_Y) \oplus \mathbf{s} \in \{0,1\}^{\ell_1 - \log n}$. However, if the receiver naively follow the remaining procedures, it would implicitly use the set $I_{x_R} := \{x_R \oplus s_{i_k} : i_k = \pi(h'_k(x))\}_{k \in [\gamma]}$ and check

$$I_{x_R} \cap \{p_i\}_{i \in [\beta]} \neq \emptyset. \tag{2}$$

Then the receiver determines $\mathbf{1}(x \in Y) = 1$ only when $x_R = y_R$ for some $y \in Y$. This causes a significantly larger false positive probability. More precisely, for the initial hash item length choice $\ell_1 \approx \lambda + \log n_x n_y$, this leads to $2^{-\lambda + \log n}$ false positive probability.

As an intuitive explanation why this happen, note that the correctness of phasing holds because the removed part x_L is reflected in the bin index $h_i'(x) = h(x_R) \oplus x_L$. It enables the other party can implicitly check whether x_L is same when computing hash bin index. However, the logic eq. (2) does not work in the way where the other party computes hash index $h_i'(x)$, but simply check whether x_R is in the whole other table.

To resolve this problem, we propose a tweak on the membership check logic eq. (2) that enables to enjoy the length saving of phashing, without any harm on false positive. The underlying idea of our tweak is to restore the bin index information, which is removed by phashing. It may seem impossible because the receiver does not know about the permutation π of P&S, but it is sufficient to attach the permuted hash table index to the item in the phashing bin; more precisely we change the definition of the set I_X by $I_X' := \{(x_R \oplus s_{id_i} | | id_i) : id_i = \pi(h_i'(x))\}_{i \in [\gamma]}$. As a corresponding change on the receiver, we let the receiver check whether

$$I_X' \cap \{p_{i,R} | | i\}_{i \in [\beta]} \neq \emptyset. \tag{3}$$

This determines $\mathbf{1}(x \in Y)$ if $x_R = y_R$ and $h_k'(x) = h_k'(y)$ for some $k \in [\gamma]$, where $x_L = y_L$ is implicitly checked by the latter condition. This change can be easily realized in the protocol by letting the receiver execute OPRF with input $\{p_{i,R}||i\}$, and the sender to computes F_k on input $\{x_R \oplus s_i||i\}$. The resulting protocol is illustrated as Figure 2b.

To summarize, our tweak enables to partially enjoy the benefit of phashing to reduce the heavy communication of P&S, by restoring the bin index information of log *n*-bit removed by phashing after P&S. We think this is an interesting situation that comes from the combination of phasing and P&S. It can be easily checked that our modification has a negligible impact on computational cost. It only consists of pre-phashing phase and a slight change in OPRF input sets.

As a final remark, the benefit of phashing optimization is significant when the original input length ℓ_0 is small. To be precise, the naive hashing brings no improvement when the collision-avoiding length $\ell_1 \approx \lambda + \log n_x n_y$ is longer than ℓ_0 . Considering this, we may

²This representation assumes that β is a power of two. For a general case, one may use the unique representation $x = x_L \cdot \beta + x_R$ where $0 \le x_R < \beta$.

assume that the RPMT input item length is $\ell' = \min(\ell_1, \ell_0)$. In this situation, phashing can further reduce the RPMT input length by $\log n_y$, no matter what is ℓ' . The effect of this reduction is clearly much larger when ℓ_0 is small; when $\ell_0 = 32$ and $n = 2^{16}$, phashing reduces almost 50% communication for P&S.

1.3 Further Related Works

We overview other recent PSU protocols, especially semi-honest proposals.

1.3.1 Zhang et. al. [31]. The construction of [31] is based on the following clever idea that achieves linear complexity without the hash bin technique. Roughly speaking, the receiver produces a ciphertext per each item $y \in Y$, and then computes OKVS encoding of the ciphertexts. The receiver sends the OKVS encoding to the sender who decodes OKVS on its items $x \in X$, and obtains a valid ciphertext only when $x = y \in X$. In a (re-randomizable) Public Key Encryption (PKE) version, the sender re-randomize the decoded ciphertexts and sends them to the receiver, so that the receiver decrypts the ciphertext to obtain RPMT result $1(x \in Y)$. This scheme arguably achieves $O(n_x + n_y)$ complexity on both computation and communication. As a downside, each computation unit is rather heavy PK operation, and each communication unit is also quite large $O(\kappa)$. In a Secret Key Encryption (SKE) version, the sender and the receiver engage 2PC that privately decrypts the ciphertext, in the sense that that the receiver only obtains RPMT result $\mathbf{1}(x \in Y)$, not the decryption result itself. The computation complexity includes $O(n_x)$ times of 2PC evaluation of underlying SKE decryption circuit. The authors used LowMC [1] cipher that has small number of AND gates, concretely $t \approx 600$ AND gates.

1.3.2 Gordon. et. al. [15]. More recently, another PSU protocol [15] based on a different idea is proposed. In this protocol, the receiver consider a polynomial $P(x) := \prod_{y \in Y} (x - y)$. Then it sends additive homomorphic encryption (AHE) of each coefficient of P(x) to the sender who then homomorphically computes encryption of P(x) for each $x \in X$. The sender sends back all ciphertexts, then the receiver decrypts and obtains P(x). It holds that P(x) = 0 if $x \in Y$, from which the receiver knows the membership indicator $\mathbf{1}(x \in Y)$. This core idea is actually adapted from [11], but the authors further optimize this using a hash table, in a similar way to [20]. The authors of [15] particularly exploit RLWE-AHE [7] that supports SIMD operations for fast evaluation.

We currently see some issues in performance reports of [15] which are briefly described here with further details in Appendix C. As the first issue, we found that it only provide performance reports with input item length $\ell_0=64$, whereas other PSU protocols considers $\ell_0=128$. As their performance seems to linearly depend on the input item length, this leads to unfair comparison. Of course, this issue is not a big deal, as we can simply run our proposals with $\ell_0=64$ to compare with [15]. However, we would raise a security issue in the [15] construction. Roughly speaking, the receiver should only know the decryption result P(x) when decrypts the returning ciphertext, which is called *function privacy* sometimes. The authors of [15] seems to aware of the necessity of function privacy, but the current treat seems to be insufficient because of the use of RLWE-based AHE. For a completeness though, one can

find some comparisons with as-is performance reports of [15] in Appendix \mathbb{C} .

2 PRELIMINARIES AND BUILDING BLOCKS

2.1 Notations

We write vectors by bold lowercase letters. For any real number x, we denote $\lfloor x \rceil$ by the round-off to an integer. The i-th component of a vector \mathbf{v} is denoted by v_i or v[i]. For an integer k, we denote the set $\{1,\cdots,k\}$ by $\lfloor k \rfloor$. The logarithm function log is assumed to have base 2 without special mention. For any statement T that can be determined by true or false (Boolean), we denote $\mathbf{1}(T)$ to be the truth value for the equality, i.e., it is 1 if T is true and 0 else.

2.2 Security Notion and Definition

2.2.1 Semi-honest Security. We use a standard notion of security against semi-honest adversaries, and provide a simplified description for the 2-party case. Consider a two-party protocol Π that computes an ideal functionality $f(x_1, x_2)$ where party P_i has input x_i . For each party P_i , define view $_i(x_1, x_2)$ denote the view of party P_i during an execution of Π on input x_1, x_2 . Precisely, it consists of input x_i , messages that are sent or received, sampled random tape during protocol execution, and the output of the protocol Π .

Definition 2.1. We say a (two-party) protocol Π for P_1 and P_2 computes f against semi-honest adversary, if there exists simulators Sim_1 and Sim_2 such that, for any inputs x_1, x_2 and i = 1, 2,

$$\operatorname{Sim}_{i}(x_{i}, f(x_{1}, x_{2})) \cong_{c} \operatorname{view}_{i}(x_{1}, x_{2}),$$

where \cong_c represents the computational indistinguishability.

2.2.2 *Private Set Union.* Private Set Union (PSU) lets two parties of input X and Y privately compute the union set $X \cup Y$. Typically it is defined by one-sided output where only one party obtains the result, but it is trivial to extend this to two-sided output in a semi-honest setting.

Many private set operation literature implicitly (or explicitly) assume that both parties know at least the size of the input of the opposite party. In particular for PSU case, the receiver then naturally knows the intersection size from $|X \cap Y| = |X| + |Y| - |X \cup Y|$. Thus the desired ideal functionality $\mathcal{F}_{\mathsf{PSU}}$ is defined so that additionally output $|X \cap Y|$ to the receiver.

Parameters: A sender set size n_x and a receiver set size n_y . **Input:** A sender with a set X of size n_x and a receiver with a set Y of size n_y .

Functionality: Output $X \cup Y$ (and $|X \cap Y|$) to the receiver.

Figure 3: Ideal Functionality \mathcal{F}_{PSU} of Private Set Union

2.3 Oblivious Transfer and Vector-OLE

Oblivious Transfer (OT) functionality lets the sender inputs two messages m_0 and m_1 , and lets the receiver obtains a message m_b for a choice bit $b \in \{0, 1\}$. For a later convenience, we specify two OT variants; *correlated* OT (COT $_\ell$) where the sender inputs one correlation $\delta \in \{0, 1\}^\ell$ to have $m_0 = r$ and $m_1 = r \oplus \delta$ for a random

 $r \in \{0, 1\}^{\ell}$, and $random \text{ OT } (\mathsf{ROT}_{\ell})$ where the sender has no inputs and obtain random messages m_0 and m_1 of length ℓ as outputs. To prevent confusion, we denote the original OT by standard OT (SOT_{ℓ}) . We sometimes denote m calls of XOT with message length ℓ by XOT_{ℓ}^m , where X can be C, R, S.

When huge amounts of OTs are required, it is typical to consider *OT extension* framework that assumes small (polynomials in κ) numbers of *base* OTs and extends them to a huge amount of ROT $_{\kappa}$. The state-of-the-art OT extension protocols [10, 30] require almost negligible communication, for example, less than 0.1 bit per each ROT. One can perform SOT $_{\ell}$ and COT $_{\ell}$ with additional communication $2\ell+1$ and $\ell+1$, using offline ROT as an ephemeral secret as described in [4]. As state-of-the-art ROT extension enable tremendous amounts of ROT in bulk with negligible communication cost [10, 30], we assume that total communication cost of SOT $_{\ell}$ and COT $_{\ell}$ by $2\ell+1$ and $\ell+1$.

In a (random) Vector Oblivious Linear Evaluation (VOLE) functionality, the sender obtains random vectors \mathbf{a} and \mathbf{b} over \mathbb{F} and the receiver obtains $x \in \mathbb{F}$ and $\mathbf{c} := x \cdot \mathbf{a} + \mathbf{b}$ for some field \mathbb{F} . There have been reported *silent* protocols based on pseudorandom correlation generator (PCG) [6, 10]. Those protocols require only a tiny amount of communication to build a correlated seed, and the desired output of VOLE is locally computed. The state-of-the-art protocol [10] requires only $2^{14.5}\kappa$ -bit communication³ (empirically estimated), and extremely fast local computation.

2.3.1 Gate Evaluation. Assume that two parties have Boolean shares (or additive shares) of bits x and y. The Boolean share of XOR $x \oplus y$ evaluation can be easily computed by locally. For AND $x \land y$ evaluation, we make use GMW protocol [14] that computes it using two COT₁, which requires 4-bit communication plus negligible communication for random-OTs. Sometimes in literature including [31], GMW protocol is differently described in terms of Beaver multiplication triple [5]. However, these two methods, Beaver triple based one and COT based one, have almost similar costs, because one (Boolean) Beaver multiplication triple generation takes two ROT₁ [4] and the input-dependent phase takes 4-bit communication per AND gate.

2.3.2 Private Equality Share Generation. We especially define a functionality by Figure 4 that given input strings $x, y \in \{0, 1\}^{\ell}$ from each party, computes Boolean shares of $\mathbf{1}(x = y)$. This can be efficiently realized by evaluating the equality circuit on a and b, which consists of $\ell - 1$ AND gates. Using GMW protocol, it translates into $2(\ell - 1)$ times of COT₁.

Parameters: An item length ℓ

Input: A sender with a string $x \in \{0, 1\}^{\ell}$ and a receiver with a string $y \in \{0, 1\}^{\ell}$.

Functionality: Denoting $b := \mathbf{1}(x = y)$, the functionality samples a random bit r, and output a sender r and the receiver $r \oplus b$.

Figure 4: Ideal Functionality \mathcal{F}_{PESG} of Private Equality Share Generation

2.3.3 Permute and Share. Permute and Share (P&S) is functionality that obliviously performs shuffle, whose definition is given as Figure 5. Writing $\pi(\mathbf{x}) = (x_{\pi(1)}, \cdots, x_{\pi(n)})$ for a vector \mathbf{x} of length n, this functionality is simply represented to output additive shares of $\pi(\mathbf{x})$. There are known several protocols [8, 23] that realize this functionality, but we especially focus on the protocol of [23] in this paper, as it shows much better performance on PSU applications. See Appendix A for details. It requires about $n \log n$ times of $\mathrm{COT}_{2\ell}$, where ℓ is bit-length of input vector entry.

Parameters: A permutation target size n and an item length ℓ **Input:** A sender with a permutation π over [n], and a receiver with input vector $\mathbf{t} \in (\{0,1\}^{\ell})^n$.

Functionality: The functionality samples a random vector $\mathbf{s} \in (\{0,1\}^{\ell})^n$, and sends \mathbf{s} to the sender and $\pi(\mathbf{t}) \oplus \mathbf{s}$ to the receiver.

Figure 5: Ideal Functionality $\mathcal{F}_{P\&S}$ of Permute and Share

2.4 Oblivious Pseudo-Random Functions

2.4.1 Oblivious Pseudo-Random Function (OPRF). In Oblivious Pseudo-Random Function (OPRF), the sender with no input obtains a key k that determines a PRF F_k , and the receiver with an input set Y obtains $F_k(Y)$. The sender learns no information of Y (other than |Y|), which explains the term oblivious. The formal definition is described in Figure 6.

Parameters: A receiver set size n_y **Input:** A receiver with input Y of size n_y . **Functionality:** The functionality samples a PRF key for F_k : $\{0,1\}^* \to \{0,1\}^\ell$ and sends $F_k(Y) := \{F_k(y) : y \in Y\}$ to the receiver, and k to the sender.

Figure 6: Ideal Functionality \mathcal{F}_{OPRF} of OPRF

2.4.2 Oblivious Programmable PRF (OPPRF). Oblivious Programmable PRF (OPPRF) is a special sort of OPRF where the sender can *program* some PRF values. The sender feeds an input set X and corresponding values $V = \{v_x : x \in X\}$, so that F_k satisfies $F_k(x) = v_x$ for $x \in X$. The formal definition is described in Figure 7.

Parameters: A sender set size n_x and a receiver set size n_y , and value length ℓ .

Input: A sender with an input-value pair set $(X, V) = \{(x, v_x) : x \in X, v_x \in \{0, 1\}^{\ell}\}$ of size n_x , and a receiver with input Y of size n_y .

Functionality: The functionality samples a random function $F_k: \{0,1\}^* \to \{0,1\}^\ell$ conditioned on $F_k(x) = v_x$ for every (x,v_x) pair, and sends $F_k(Y) := \{F_k(y) : y \in Y\}$ to the receiver, and the key k to the sender.

Figure 7: Ideal Functionality \mathcal{F}_{OPPRF} of OPPRF

³Almost independent to the field size and the vector length.

2.5 Oblivious Key-Value Store

Oblivious Key-Value Store (OKVS) [13] is a useful concept that abstracts many recent constructions of O(P)PRF. OKVS is a pair of algorithms Encoding and Decoding, where Encoding on an input key-value pair (X,V) outputs some object (typically a vector) P, and Decoding on an input key y and the encoding result P outputs v_X if $y=x\in X$ and random value otherwise. For an input key size n, the state-of-the-art OKVS algorithm named by 3H-GCT [13] achieves $O(n\lambda)$ encoding time complexity, and $O(\lambda)$ decoding time complexity, and corresponding P consists of roughly 1.3n items having the same size with values $v\in V$.

Garimella et. al. [13] provides an abstract construction of OPRF on input Y, from O(n) VOLEs over some field $\mathbb F$ and OKVS Encoding with a key set Y and the value set $V=\{H(y):y\in Y\}$ with a random oracle $H:\{0,1\}^*\to\mathbb F$. Rindal and Schoppmann [29] show that OPPRF on (X,V) and Y can be obtained from OPRF on Y that determines a function F_k , and then OKVS Encoding with a key set X and the value set $Y':=\{v_x-F_k(x):x\in X\}$. In both constructions, the computation cost is dominated by OKVS Encoding and Decoding, and the communication cost is dominated by the size of P (thanks to the small communication cost of VOLE).

3 OUR SENDER CUCKOO PROTOCOL

In this section, we present our sender-side cuckoo hashing PSU protocol. Our construction starts from the framework of Garimella et. al. [12], which can be abstracted by following consecutive stages: (1) cuckoo / simple hashing to reduce the set size of private membership test (2) reduce each bin-wise private membership test into equality test (PEqT) instances using OPPRF (3) shuffle the order of PEqT instances using P&S (4) solve PEqT using OPRF to let the receiver know $\mathbf{1}(x_i \in Y)$ (5) obliviously sends X-Y using Boolean shares. Thanks to recent advances in submodules, one can expect a fast performance of this framework. However, the problem lies in the huge communication cost, which is mainly due to P&S on PEqT instances.

We briefly explain our idea that reduces that communication burden. We first observe that the core fact for security is that the receiver knows the membership indicator $\mathbf{1}(x \in Y)$ in shuffled order. Our main idea is to achieve the same goal by (3') solving PEqT in *secret-shared* manner, and then (4') shufling the results. This change greatly reduces the communication burden on P&S by changing the shuffling target item to Boolean. Moreover, secret-shared PEqT exactly corresponds to PESG of Figure 4, which can be efficiently executed from the GMW protocol. The details are presented in Figure 8.

A Tweak on Final OT. Denote the cuckoo table T_X as a vector $\mathbf{x}=(x_i)$. In Garimella et. al., the receiver knows $\mathbf{1}(x_{\pi(i)} \in Y)$ in consecutive order, so that the sender obliviously sends $m_0 = x_{\pi(i)}$ or $m_1 = \bot$. In our protocol of Figure 8, the receiver instead knows a Boolean share of $\mathbf{1}(x_{\pi(i)} \in Y)$ as c_i' , and the sender holds the other part say c_i . Instead of letting the sender send $\mathbf{c}=(c_i)$ to have an exactly same situation as Garimella et. al., we observe that two parties can directly engage the final OT with the sender's message $m_{c_i} = x_{\pi(i)}$ and $m_{1-c_i} = \bot$ and the receiver's choice c_i' . The tweak on the final OT phase decreases in one round interaction and tiny communication for sending \mathbf{c} .

Optimization on Shuffling. There is an optimization proposed in [12], that generalizes P&S to an injective function $\rho: [n] \to [\beta]$, instead of a permutation on $[\beta]$. Given a target vector $\mathbf{x} \in (\{0,1\}^\ell)^\beta$, it computes additive shares of $\rho(\mathbf{x}) := (x_{\rho(i)}) \in (\{0,1\}^\ell)^n$. The sender uses ρ to exclude dummy bins in the cuckoo table, and this provides a small benefit on procedures after P&S as it reduces target vector length from β to n. We can also apply this optimization, but we omit this in formal descriptions and performance evaluations for the sake of brevity.

Correctness and False Positives. For i-th cuckoo bin filled with $x \in X$, it holds that $t_{s,i} = t_{r,i}$ if $x \in Y$, thanks to the definition of $\mathcal{F}_{\text{OPPRF}}$. More precisely, we defined $t_{r,i} = F_k(x||j_x)$ where j_x is the the hash index such that $h_{j_x}(x) = i$. If $x = y \in Y$, it clearly holds that $F_k(x||j_x) = F_k(y||j_x)$ where the RHS maps to $t_{s,h_{j_x}(y)}$, thanks to the definition of OPPRF. Finally, we know $t_{s,h_{j_x}(y)} = t_{r,h_{j_x}(x)} = t_{r,i}$ from the definition of j_x .

As the next step, the private equality share generation \mathcal{F}_{PESG} ensures $b_{s,i} \oplus b_{r,i} = 1$ if and only if $t_{s,i} = t_{r,i}$. Then $\mathcal{F}_{P\&S}$ permutes them without correctness harm, and then $c_{s,\pi(i)} \oplus c_{r,\pi(i)} = 1$ if and only if $t_{s,i} = t_{r,i}$. Finally, \mathcal{F}_{SOT} sends x_i if and only if $c_{s,\pi(i)} \oplus c_{r,\pi(i)} = 0$, which holds when $x_i \notin Y$.

The only part that may cause a false positive is $x \in Y \Rightarrow t_{s,i} = t_{r,i}$, whose reverse is not always true. Since $t_{s,i}$ is uniformly chosen from $\{0,1\}^{\ell}$, it may happen that $t_{s,i} = t_{r,i}$ despite $x \notin Y$ with probability $1/2^{\ell}$. We set the OPPRF output length by $\ell \geq \lambda + \log \beta$ to avoid such event in every bin with probability $2^{-\lambda}$.

Security Proof. Theorem 3.1 shows the security of our construction against a semi-honest adversary.

Theorem 3.1. The protocol in Figure 8 realizes \mathcal{F}_{PSU} of Figure 3 in semi-honest setting, in \mathcal{F}_{OPPRF} , $\mathcal{F}_{P\&S}$, \mathcal{F}_{OT} -hybrid model.

PROOF. We first construct $\operatorname{Sim}_{\mathcal{S}}$ that simulates the views of corrupt \mathcal{S} of input X and no output. First, $\operatorname{Sim}_{\mathcal{S}}$ samples PRF key k uniformly at random, and simulates the views for $\mathcal{F}_{\operatorname{OPPRF}}$ by computing the output set $\{F_k(x||j_x):x\in X\}$. Then it samples a random Boolean vector $\mathbf{b}_s\in\{0,1\}^{\beta}$ and uses it as the output of $\mathcal{F}_{\operatorname{PESG}}$ to simulates the views. Similarly it samples another random Boolean vector $\mathbf{s}_s\in\{0,1\}^{\beta}$ to simulates the views for $\mathcal{F}_{\operatorname{P\&S}}$. The simulation for $\mathcal{F}_{\operatorname{OT}}$ can be done trivially, as it has no output. $\operatorname{Sim}_{\mathcal{S}}$ is trivially indistinguishable from the view of \mathcal{S} in the real execution, since each simulated output follows exactly the same distribution of the real execution with $\mathcal{F}_{\operatorname{OPPRF}}$, $\mathcal{F}_{\operatorname{PESG}}$ and $\mathcal{F}_{\operatorname{P\&S}}$.

We proceed to $\operatorname{Sim}_{\mathcal{R}}$ that simulates the views of corrupt \mathcal{R} of input Y and output $X \cup Y$ (and $|X \cap Y|$). $\operatorname{Sim}_{\mathcal{R}}$ samples a random PRF key k to simulate the views for $\mathcal{F}_{\operatorname{OPPRF}}$. Similar to the corrupt S case, it samples a random Boolean vector $\mathbf{b}_r \in \{0,1\}^{\beta}$ to simulates the views for $\mathcal{F}_{\operatorname{PESG}}$, and samples another random Boolean vector $\mathbf{c}_r \in \{0,1\}^{\beta}$ to simulates the views for $\mathcal{F}_{\operatorname{P\&S}}$. To simulate $\mathcal{F}_{\operatorname{OT}}$, $\operatorname{Sim}_{\mathcal{R}}$ first takes a random subset $U \subset [\beta]$ of size $|X \cup Y - Y|$. Then it simulates the views of i-th OT call by taking one item from $X \cup Y - Y$ in random order (as output), otherwise taking \bot .

We show the indistinguishability of $Sim_{\mathcal{R}}$ from the real execution via the sequence of the following hybrids:

Hyb 0. The real execution where \mathcal{R} runs honestly.

Input: A sender S with an input set $X \subset \{0,1\}^{\ell_0}$ of size n_X and a receiver R with an input set $Y \subset \{0,1\}^{\ell_0}$ of size n_Y .

Protocol:

- (1) S and R agree on hash table size β , random functions $h_1, \dots, h_{\gamma} : \{0, 1\}^* \to [\beta]$, and dummy element $\bot \in \{0, 1\}^{\ell_0}$.
- (2) S constructs cuckoo table CT_X from X using h_1, \dots, h_γ of size β , with empty bins filled by \bot . After cuckoo hash done denote $j_X \in [\gamma]$ be the hash index that x is finally stored in $h_{j_X}(x)$ -th bin.
- (3) S and R invoke \mathcal{F}_{OPPRF} :
 - Both parties agree on OPPRF output length ℓ .
 - *S* acts as a receiver with an input set $\{x | |j_x : x \in X\}$.
 - \mathcal{R} samples a random tag $\mathbf{t}_r \in (\{0,1\}^\ell)^\beta$, and acts as a sender with the following input-value set of size $\gamma \cdot n_{\psi}$:

$$\left\{ \left(y||j,t_{r,h_{j}}(y)\right):y\in Y,j\in\left[\gamma\right]\right\}$$

- \mathcal{R} receives a PRF key k, and \mathcal{S} receives $\{F_k(x||j_x): x \in X\}$.
- (4) *S* defines $\mathbf{t}_{S} = (t_{S,i}) \in (\{0,1\}^{\ell})^{\beta}$ where

$$t_{s,i} = \begin{cases} F_k(x||j_x) & \text{if } \mathsf{CT}_X[i] = x \in X \\ r & \text{otherwise,} \end{cases}$$

for some random value $r \in \{0, 1\}^{\ell}$.

- (5) For each $i \in [\beta]$, S and R invoke \mathcal{F}_{PESG} :
 - Each party feeds $t_{s,i}$ and $t_{r,i}$, respectively.
 - Each party receives $b_{s,i} \in \{0,1\}$ and $b_{r,i} \in \{0,1\}$, respectively.
- (6) S and R invoke $\mathcal{F}_{P\&S}$:
 - S picks a random permutation π on $[\beta]$, and acts as a sender with input π .
 - \mathcal{R} acts as a receiver with input $\mathbf{b}_r = (b_{r,i}) \in \{0,1\}^{\beta}$.
 - S receives $s \in \{0,1\}^{\beta}$, and R receives $\mathbf{c}_r := \pi(\mathbf{b}_r) \oplus \mathbf{s} \in \{0,1\}^{\beta}$.
- (7) S defines $\mathbf{c}_s = \pi(\mathbf{b}_s) \oplus \mathbf{s}$, where $\mathbf{b}_s = (b_{s,i})$
- (8) \mathcal{R} initializes $Z = \emptyset$.
- (9) For each $i \in [\beta]$,
 - (a) S and R invoke SOT_{ℓ_0} :
 - S acts as a sender with input $m_{c_{s,i}} = \mathsf{CT}_X[\pi(i)]$ and $m_{1-c_{s,i}} = \bot$
 - \mathcal{R} acts as a receiver with choice index $c_{r,i}$.
 - \mathcal{R} receives z_i .
 - (b) If $z_i \neq \bot$, \mathcal{R} adds z_i to Z.
- (10) \mathcal{R} outputs $Y \cup Z$.

Figure 8: A Full Description of Our Sender-Cuckoo Shuffle-based PSU Protocol

- Hyb 1. Change PRF key to another one k_{sim} sampled randomly from PRF key space, and Boolean vectors $\mathbf{b}_{r,sim} \in \{0,1\}^{\beta}$ and $\mathbf{s}_{r,sim} \in \{0,1\}^{\beta}$ sampled uniformly at random. This change is indistinguishable from Hyb 0, since the random choice of key in $\mathcal{F}_{\text{OPPRF}}$, and uniform randomness of outputs of $\mathcal{F}_{\text{PESG}}$ and $\mathcal{F}_{\text{P\&S}}$.
- Hyb 2. Sample a random subset U of β of size $|X \cap Y Y|$. Then uniformly assign elements of $X \cap Y Y$ for i-th OT for $i \in U$, and \bot for $i \notin U$. This is perfectly indistinguishable from Hyb 1, thanks to the random choice of permutation π .

Note that Hyb 2 is same with the views output by $Sim_{\mathcal{R}}$.

Cost Analysis. We decompose our protocols into 4 phases to represent the costs in a convenient form for comparison: (1) OPPRF on the simple table of size γn_u and the cuckoo table of size β

 $O(n_x)$. More precisely, it first invokes OPRF on X that requires VOLE of size εn_X over a field size 2^{κ} , and OKVS encoding on X with values of bit-length κ . Then the receiver decodes the received OKVS on each simple table item of size γn_y , and sends back an OKVS encoding on another γn_y size inputs with values of bit-length ℓ . Finally, the sender decodes the received OKVS n_X times. (2) $\mathcal{F}_{\text{PESG}}$ of $\beta \ell$ AND gate evaluation. It requires $2\beta \ell$ times of COT₁ using GMW protocol. (3) $\mathcal{F}_{\text{P\&S}}$ of $\beta \log \beta$ times of COT₂ using a protocol [23]. (4) The final OT that consists of β times of SOT $_{\ell_0}$.

Table 1 summarizes the results. Note that we represent OKVS cost by adding all input set sizes required during protocol executions. Although OKVS Encoding and Decoding are actually done separately on different input sets during protocol execution, our representation is reasonable as the complexity is linear on the input set size.

Table 1: The costs of sender-cuckoo based protocols for a sender set size n_x and a receiver set size n_y . OKVS(a) represents the computation for OKVS Encoding on a size set, and OKVS Decoding on a size set. γ is the number of hash functions for cuckoo hashing, and $\beta = O(n_x)$ is the cuckoo hash table size. ε is the OKVS encoding size expansion factor. ℓ_0 denotes the original item length, and $\ell \approx \lambda + \log \beta$ is OPPRF output length. κ is a computational security parameter, and λ is a statistical security parameter.

Ours	OPPRF	PESG	P&S	Final OT
Comp.	$OKVS(n_x + \gamma n_y)$	$COT_1^{2eta\ell}$	$COT_2^{\beta \log \beta}$	$SOT^eta_{\ell_0}$
Comm.	$\varepsilon(\kappa n_x + \gamma \ell n_y)$	$2\ell\beta$	$2\beta \log \beta$	$2\ell_0\beta$
[12]	OPPRF	P&S	PEqT	Final OT
Comp.	$OKVS(n_x + \gamma n_y)$	$COT_{2\ell}^{\beta \log \beta}$	$OKVS(\beta)$	$SOT^eta_{\ell_0}$
Comm.	$\varepsilon(\kappa n_x + \gamma \ell n_y)$	$2\ell\beta\log\beta$	$\varepsilon(\kappa + \ell)\beta$	$2\ell_0\beta$

Comparison with Garimella et. al. With respect to the view of computation cost, the only difference of [12] with our proposal is whether to perform PEqT on β strings (of length ℓ) or PESG of β strings. PEqT can be efficiently done by executing OPRF on the receiver's inputs and then the sender sends its PRF evaluation values. Thus PEqT computation is dominated by OPRF, which consists of OKVS Encoding and Decoding on β inputs and VOLE of size $\varepsilon\beta$, instead of $2\beta\ell$ times of COT₁.

4 OUR RECEIVER CUCKOO PROTOCOL

In this section, we present our receiver-side cuckoo hashing PSU protocol. This shares the framework of Jia et. al.[17], which consists of the following consecutive stages: (1) the receiver builds a cuckoo table to reduce the set size of the corresponding membership problem (2) generate additive shares of the shuffled cuckoo table using P&S (3) the receiver obtains $\mathbf{1}(x \in Y)$ for each $x \in X$ using OPRF (4) obliviously sends X - Y. This shows considerably fast computational cost, but also suffers from huge communication costs due to P&S on the cuckoo table.

Our first idea to reduce communication is an application of permutation-based hashing (phashing) on the cuckoo hash part. It readily reduces the communication cost burden of P&S, but significantly harms correctness, due to a larger false positive probability. Our second idea is to modify OPRF stage to normalize the correctness, which is based on a simple but clever idea that reflects the property of phashing. As a consequence, we become able to enjoy the communication benefit of phashing without correctness harm. The formal description is presented in Figure 9, and below we present the detailed explanation.

Correctness and False Positives. Pick an item $x \in X$. Suppose that $x = y \in Y$ and y' := H(y) is stored by k-th index function h'_k . It can be easily checked that for x' = H(x)

$$h'_k(x') = h'_k(y') \wedge x'_R = y'_R,$$
 (4)

thanks to the permutation-based hashing technique. This is equivalent to

$$\pi \circ h_k(x')||x_R' \oplus s_{\pi \circ h_k(x')} = \pi \circ h_k(y')||y_R' \oplus s_{\pi \circ h_k(y')},$$

where the LHS is an element in I_x , and the RHS is an element in \mathbf{p}' . Thus $F_k(I_x)$ and $F_k(\mathbf{p}')$ both contains the element, so that the receiver concludes $b_x=1$.

There could be false positive cases where the receiver concludes $b_X = 1$ despite $x \notin Y$. First, it may happens that H(x) = H(y) for some $y \in Y$. To prevent this collision by preprocessing hash, we use the birthday bound to have

$$\lambda + \log n_x n_y \le \ell_1$$
.

Second, the set I_x and $\mathbf{p'}$ may have false intersections. In other words, eq. (4) falsely holds for some index $k \in [\gamma]$, precisely

$$h_k(x') \oplus h_k(y') = x_L \oplus y_L \wedge x_R = y_R$$

for some $k \in [\gamma]$. For each k, the probability of the former part is $2^{-\log \beta}$, and latter part is $2^{-\ell_2}$. Then it can happen with probability $2^{-\ell_2 - \log(\gamma \beta)}$ for some $k \in [\gamma]$. To prevent this event for every $x \in X$ with probability $2^{-\lambda}$, we set $\lambda + \log(\gamma n_x) \le \ell_2$, equivalently

$$\lambda + \log(\gamma \beta n_x) \le \ell_1.$$

Finally, the set $F_k(I_x)$ and $F_k(\mathbf{p}')$ may have false intersection, despite $I_x \cap \mathbf{p}' = \emptyset$. This can happen since $F_k(I_x)$ consists of γ random elements in $\{0,1\}^\ell$ for OPRF output length ℓ . Each element of $F_k(I_x)$ can falsely intersects with $F_k(\mathbf{p}')$ with probability $< 2^{-\ell + \log \beta}$. The probability of false intersection for each $F_k(I_x)$ is $< 2^{-\ell + \log (\gamma \beta)}$. In order to upper bound the false intersection for every $x \in X$ by we set

$$\lambda + \log(\gamma \beta n_x) \le \ell$$
.

To summarize, it is sufficient to take both ℓ_1 and ℓ by

$$\lambda + \log(\gamma \beta n_x) \approx \lambda + \log \gamma + \log(n_x n_y)$$

to ensure false positive $2^{-\lambda}$.

Table 2: The costs of receiver-cuckoo based protocols for a sender set size n_x and a receiver set size n_y . OKVS(a,b) represents the computation for OKVS Encoding on a size set, and OKVS Decoding on b size set. γ is the number of hash functions for cuckoo hashing, and $\beta = O(n_y)$ is the cuckoo hash table size. ε is the OKVS encoding size expansion factor. ℓ_0 denotes the original item length, $\ell_1 \approx \lambda + \log \gamma + 2 \log n$ is both OPRF output length and initial hash compression length, and $\ell_2 \approx \min(\ell_0, \ell_1) - \log n$ is phashing length. κ is a computational security parameter, and λ is a statistical security parameter.

Ours	P&S	OPRF	Final OT
Comp.	$COT_{2\ell_2}^{\beta\log\beta}$	$OKVS(\beta, \gamma n_x)$	$SOT^{n_{\scriptscriptstyle \mathcal{X}}}_{\ell_0}$
Comm.	$2\ell_2\beta\log\beta$	$\varepsilon\kappa\beta+\gamma\ell_1n_x$	$2\ell_0 n_X$
[17]	P&S	OPRF	Final OT
Comp.	$COT_{2\ell_0}^{\beta\log\beta}$	$OKVS(\beta, \gamma n_x)$	$SOT^{n_{\scriptscriptstyle \mathcal{X}}}_{\ell_0}$
Comm.	$2\ell_0\beta\log\beta$	$\varepsilon\kappa\beta + \gamma\ell_1n_x$	$2\ell_0 n_x$

Input: A sender S with an input set $X \subset \{0, 1\}^{\ell_0}$ of size n_X and a receiver R with an input set $Y \subset \{0, 1\}^{\ell_0}$ of size n_Y . **Protocol:**

- (1) S and \mathcal{R} agree on initial hash length ℓ_1 and hash function $H: \{0,1\}^{\ell_0} \to \{0,1\}^{\ell_1}$.
- (2) If $\ell_0 > \ell_1$, the receiver computes Y' = H(Y). Otherwise Y' = Y.
- (3) S and R agree on hash table size β and cuckoo hash functions $h_1, \dots, h_{\gamma} : \{0, 1\}^* \to [\beta]$.
- (4) \mathcal{R} computes cuckoo table $\mathsf{CT}_Y \in (\{0,1\})^{\ell_2}$ of Y' using permutation-based hashing routine with h_1, \cdots, h_{γ} . Denote the index function used for permutation-based hashing by $h'_i : x \mapsto x_L \oplus h_i(x)$ where $x = x_L ||x_R|$ with $|x_L| = \log \beta$. Note that $\mathsf{CT}_Y[i]$ has length $\ell_2 := \ell_1 \log \beta$.
- (5) S and R invoke $\mathcal{F}_{P\&S}$:
 - S samples a random permutation π on $[\beta]$, and acts as a sender with input π .
 - \mathcal{R} acts as a receiver with input CT_{Y} .
 - S receives s, and R receives $p := \pi(CT_Y) \oplus s$.
- (6) S and R invoke \mathcal{F}_{OPRF} :
 - Both parties agree on OPRF output length ℓ .
 - \mathcal{R} acts as a receiver with input $\mathbf{p}' = (p_i||i)_{i \in [\beta]}$.
 - S receives a PRF key k, and $\mathcal R$ receives $F_k(\mathbf p') \coloneqq \left(F_k(p_i')\right)_{i\in [\beta]} \in (\{0,1\}^\ell)^\beta$
- (7) \mathcal{R} initialize $Z = \emptyset$.
- (8) For each *x* in *X*:
 - (a) S defines

$$I_{x} := \left\{ (x_{R}' \oplus s_{id_{i}} || id_{i}) : id_{i} = \pi \circ h'_{i}(x'), i \in [\gamma] \right\},$$
 where $x' = H(x)$ if $\ell_{0} > \ell_{1}$, otherwise $x' = x$. Then S sends to R the set $F_{k}(I_{x})$.

- (b) \mathcal{R} checks $F_k(I_x) \cap F_k(\mathbf{p'}) = \emptyset$. If so, \mathcal{R} sets $b_x = 0$, otherwise sets $b_x = 1$.
- (c) S and R invoke \mathcal{F}_{OT} :
 - S acts as a sender with input $m_0 = x$ and $m_1 = \bot$.
 - \mathcal{R} acts as a receiver with choice index b_x .
 - \mathcal{R} receives z_x .
- (d) If $z_i \neq \bot$, \mathcal{R} adds z_i to Z.
- (9) \mathcal{R} outputs $Y \cup Z$.

Figure 9: A Full Description of Our Receiver Cuckoo Shuffle-based PSU Protocol

Security Proof. Theorem 4.1 shows the semi-honest security of our construction. The simulator construction and proof are almost the same with [17], and hence we only describe simulators with some intuitions in this paper.

Theorem 4.1. The protocol in Figure 9 realizes \mathcal{F}_{PSU} of Figure 3 in semi-honest setting, in $\mathcal{F}_{P\&S}$, \mathcal{F}_{OPRF} , \mathcal{F}_{OT} -hybrid model.

PROOF. We first construct $\operatorname{Sim}_{\mathcal{S}}$ that simulates the views of corrupt \mathcal{S} of input X and no output. $\operatorname{Sim}_{\mathcal{S}}$ samples a random string $\mathbf{s} \in (\{0,1\}^{\ell_2})^{\beta}$, and simulates the views for $\mathcal{F}_{P\&S}$ using $\mathcal{F}_{P\&S}$ -simulator with output \mathbf{s}_s . Similarly, it samples a random key k from the PRF key space to simulate the views for \mathcal{F}_{OPRF} with output k. The final simulation for \mathcal{F}_{OT} views is immediate, as it has no output. Each simulation outputs exactly the same distribution with $\mathcal{F}_{P\&S}$, \mathcal{F}_{OPRF} and \mathcal{F}_{OT} , and hence this $\operatorname{Sim}_{\mathcal{S}}$ is indistinguishable from the view of \mathcal{S} in the real execution.

We proceed to $\operatorname{Sim}_{\mathcal{R}}$ that simulates the views of corrupt \mathcal{R} of input Y and output $X \cup Y$ and $|X \cap Y|$. It samples a random string $\mathbf{p} \in (\{0,1\}^{\ell_2})^{\beta}$ to simulate the views for $\mathcal{F}_{P\&S}$, similarly to the corrupt \mathcal{S} case. To simulate \mathcal{F}_{OPRF} , it samples a random key k and

use $F_k(\mathbf{p'}) := (F_k(p_i||i))$ as an output of \mathcal{F}_{OPRF} . The simulation of the views of the sets $F_k(I_X)$ for $x \in X$ is as follows. For the sake of convenience, we represent $X = \{x_1, \cdots, x_{n_X}\}$. Since $Sim_{\mathcal{R}}$ knows $|X \cap Y|$, it can takes a random subset $U \subset [n_X]$ of size $|X \cap Y|$, which would be the bin index where the RPMT results is 1. If $i \in U$, $Sim_{\mathcal{R}}$ defines $F_k(I_{x_i})$ by a random entry in $F_k(\mathbf{p'})$ with $\gamma - 1$ random elements other than $F_k(\mathbf{p'})$. If $i \notin U$, $Sim_{\mathcal{R}}$ defines $F_k(I_{x_i})$ by γ random elements other than $F_k(\mathbf{p'})$. Finally, it remains to simulate \mathcal{F}_{OT} views. For an input $b_X = 0$, $Sim_{\mathcal{R}}$ takes one item from $X \cup Y - Y$ in random order as output, otherwise use \bot .

Cost Analysis. We divide our proposal into 3 phases: (1) P&S phase with $\mathcal{F}_{P\&S}$ call that includes $\beta \log \beta$ times of $COT_{2\ell_2}$ where $\beta = O(n_y)$, (2) OPRF phase \mathcal{F}_{OPRF} call on a vector of size β , which requires $\varepsilon\beta$ VOLE and OKVS on β size set with values of size κ , and the sender sends γn_x PRF evaluations of each length ℓ_1 . (3) Final OT phase that consists of n_x times of SOT_{ℓ_0} .

Table 2 shows the results with respect to submodules. We again represent OKVS cost by adding all input set sizes required during protocol executions, as the cost is linear in the size of inputs.

In terms of cost, the only difference from [17] is the input item bit-length for P&S. It can be easily confirmed that the computational cost of our protocol is almost the same as [17], as our protocol has only an additional computational burden on the preprocessing hash and permutation-based hash that require negligible time.

5 PERFORMANCE EVALUATION

In this section, we evaluate the performances of our proposals and provide comparisons with other protocols. Throughout this section, we assume the computational security parameter $\kappa=128$ and the statistical security parameter $\lambda=40$ for concrete evaluations.

Submodule Instantiations. We assume that cuckoo hash is done with $\gamma=3$ hash functions with corresponding table size expansion factor $\varepsilon=1.3$, which is empirically considered to have $2^{-\lambda}$ failure probability. We consider 3H-GCT construction [13] for OKVS that has encoding expansion factor $\varepsilon=1.3$, which also have $2^{-\lambda}$ failure probability. For VOLE and OT extension, we assume Silver [10]. For P&S, we consider the switching network based construction [23].

To instantiate [31] protocols that are our main comparison targets, we basically follow those of the original work; we consider ECC ElGamal encryption scheme with the curve SecP256K1 for [31]-PK, and LowMC [1] for [31]-SK where the block size and the key size are both 128-bit, the number of Sboxes m = 10, and the number of rounds are taken r = 20.4

5.1 Theoretical Comparisons

5.1.1 Communication Costs. Table 3 shows asymptotic communication costs represented with dominant terms, and some concrete numbers for various input sizes. The concrete numbers for ours, [12], and [17] can be reproduced from Table 1 and Table 2. The numbers for [31] protocols are computed from the formulas in the original paper.

As we stressed several times, the largest bottleneck of communication costs of previous shuffle-based PSU [12, 17] comes from the dependency on $\ell n \log n$, for some length parameter $\ell \geq O(\lambda + \log n)$. Our sender-cuckoo based protocol (Our-SC) splits this term into $\ell n + n \log n$, and this shows considerable decreases in communication costs. Moreover, this shows the lowest concrete cost among all known PSU protocols, including [31].

Our receiver-cuckoo protocol (Our-RC) still has a term of the form $\ell n \log n$, and only optimizes by reducing ℓ by $\log n$. Therefore, it still requires somewhat heavy communication compared to [31]. However, we would like to stress that it has a special advantage when the input length ℓ_0 is short. In that case, Our-RC has parameter $\ell = \ell_0 - \log n$, and this significantly reduces communication cost. For example, when $\ell_0 = 32$, Our-RC has even better concrete cost than [31] as shown in Table 3. Note that [12] cannot enjoy this, as it always has a dependency with λ regardless of the input length ℓ_0 .

5.1.2 Computation Costs. Table 4 represents required computations with respect to common submodules, say OKVS Encoding and Decoding, and OT and VOLE. Regarding OT, note that we exploit the OT extension framework that consists of an offline ROT phase, and an online phase that interacts with actual messages. Since the

Table 3: Communication costs of PSU protocols, where ℓ_0 is the input item length. For each column, we mark the best value by blue, and the next by grey. The constant t is the number of AND gates of LowMC [1] decryption circuit, which is ≈ 600 for actual instantiation.

	Asymptotic	$\ell_0 =$	128	$\ell_0 = 32$	
	Tisymptotic	2^{16}	2^{20}	2^{16}	2^{20}
[12]	$O(\lambda n \log n)$	3826n	4406n	3576n	4157n
[17]	$O(\ell_0 n \log n)$	6082n	7395 <i>n</i>	1896n	2211n
[31]-PK	$O(\kappa n)$	1433n	1433n	1241n	1241n
[31]-SK	O(tn)	3016n	3058n	2569n	2569n
Our-SC	$O(\lambda n + n \log n)$	1168n	1136n	918n	886n
Our-RC	$O(\lambda n \log n)^5$	3419n	4483n	1231n	1379n

online phase is computationally cheaper than the offline ROT phase, we count all required OT calls without distinguishing COT and SOT, and represent them by the column 'ROT'.

Although our experimental results are based on specific constructions of submodules at the time of writing, the abstraction of Table 4 will remain useful even with further improvements on each submodule, and help future works to compare with the protocols that we cover.

Table 4: Computation costs of PSU protocols with respect to submodules. The constant t is the number of AND gates of LowMC [1] decryption circuit, which is ≈ 600 for actual instantiation.

	OKVS		ROT	VOLE	Other
	Enc	Dec			
[12]	5.3 <i>n</i> (1+3+1.3)	5.3 <i>n</i> (3+1.3+1.3)	$1.3n\log n$	3.4n	-
[17]	1.3 <i>n</i>	3 <i>n</i>	$1.3n\log n$	1.7 <i>n</i>	-
[31]-PK	n	n	-	-	n PK
[31]-SK	n	n	$\begin{array}{c} 2tn \\ (\approx 1200n) \end{array}$	-	-
Our-SC	4n (1+3)	4 <i>n</i> (3 + 1)	$2.6\lambda n + 3.9n \log n$	1.7 <i>n</i>	-
Our-RC	1.3 <i>n</i>	3 <i>n</i>	$1.3n\log n$	1.7 <i>n</i>	-

5.2 Experimental Comparisons

5.2.1 Overview of Our Implementation. We found that some previous PSU works publicized their implementations [12, 17]. However, they used rather old instantiation for some submodules, e.g. OPRF or OT extension, whose performances have been rapidly improved over the past few years. To capture the latest performance state of each PSU protocol, we have to replace them into state-of-the-art constructions. As a consequence, we end with recent PSU protocols implementation in C++ with updated submodule instantiations,

⁴Measured by the attack estimation code in [2]

 $^{^5}O(\ell_0 n \log n)$ when $\ell_0 < \lambda + 2 \log n$

which covers original shuffle-based one [12, 17], the linear complexity one [31], and our optimized shuffle-based one. We are planning to publicize our implementation.

Below are some further information of each submodule implementation. We self-implement 3H-GCT OKVS by following the description in [13], and rely on libOTe library [28] for Silver VOLE and OTe, and Microsoft Kuku library [21] for a cuckoo hashing. For P&S, we adapted the implementation of [12, 17], while changing the underlying OTs from IKNP [16] to Silver [10]. For [31]-PK, we rely on MCL library [22] that provides an ECC ElGamal encryption implementation, which is also used as a base library in [31] implementation. For [31]-SK, we adapt the publicly available implementation of LowMC [2], but with some simple speed-ups on implementation. Note that the decryption phase should be executed in 2PC manner in [31]-SK, and we use GMW protocol for that.

All our experiments are conducted on a machine equipped with Intel Xeon Silver 4208 CPU of 2.10GHz clock, and 128GB RAM. The network environment is simulated using linux tc command.

5.2.2 Experimental Results. We provide experimental results over various networks by Table 5. Taking a look at the exact amount of improvement, our optimizations have almost no improvement for extremely high bandwidth (like > 10Gbps), where a decrease in communication cost has negligible impact. Our-SC is even slower than the original sender-cuckoo [12], and this would arguably be due to the change from PEqT (based on OPRF) to PESG (based on OT). Apart from the advantage of our optimizations, we would like to remark that shuffle-based PSUs, especially receiver-cuckoo version [17], have really small computational cost, considering state-of-the-art instantiations of OPRF [13] and OT extension [10].

The value of our optimization comes out for slower bandwidth networks. For a medium network environment (100Mbps) where computation and communication costs both affect running time, Our-SC and Our-RC perform best. Our optimization brings about 1.36-1.46x improvement over the original shuffle-based PSUs for $n=2^{16}-2^{20}$, and 3.21-4.15x over [31] protocols. For an extremely slow network environment (10Mbps) where communication cost is dominant for running time, Our-SC is the best and [31]-PK becomes a runner-up instead of Our-RC. The ratio of improvement is about 1.8-2.18x for $n=2^{16}-2^{20}$.

Superiority of Our Proposals. As for Our-RC, we can argue that it is strictly superior than [17] and [12] from theoretical comparisons of Table 3 and Table 4, even without any experimental results. Our-RC has almost the same computation cost but strictly smaller communication cost than [17], and similar communication (even slightly smaller) communication cost and smaller input sizes for every submodule than [12]. Note that this superiority holds regardless of not only network environments and but instantiations of submodules.

We then justify that Our-SC is superior to both versions of [31]. Note that both protocols of [31] require smaller OKVS input size, but significantly larger number of OTs or heavy unit operation from public key primitives. This leads to large (local) computation time, and this is really the main reason for the large running time of [31] protocols. As Our-SC requires the smallest communication costs among comparison targets, we can deduce that Our-SC is better for any network bandwidth than [31] protocols.

5.2.3 Differences with [31] Reports. We found that our reports in Table 5 shows some differences with [31, Table 3]. As a general factor, the implementation of [31] is based on a different programming language Java, which may bring some difference in experiments.

Still, we have some points to explain with more specific argument. As for [31]-SK timing, we found that the authors of [31] excludes ROT timing to generate Beaver triples in the timing reports of the main body, while assuming this process is done in advance. Although this part can be done in offline, we believe such offline cost should be specified for a fair comparison. Indeed, their comparison target protocols also had some parts (including ROTs) can be done offline, but report the total timing without separating or excluding such offline timings. To clarify things, we report the offline time including ROT time in Table 5. Note that [31] needs to spend somewhat large offline time for ROT, and Our-SC follows the next. This can be expected from the 'ROT' column of Table 4, as ROT time is linear to the number of required OTs.

As another one, [31, Table, 3] also reported quite difference performance for previous shuffle-based PSUs [12] and [17], compared to our reports. We explain two reasons for this difference. The first one comes from the difference of implementation language, where Java incurs overhead for type conversion [31, Section 6.3]. As the second one, we found that [31] used rather old OPRF due to [9] and OPPRF due to [24] to implement [12] and [17]. Our implementation exploits more efficient OKVS-based OPRF and OPPRF constructions [13, 29] that are recently proposed. Thus we believe our timing more exactly shows the current state of [12] and [17], as well as our proposed protocols.

Lastly, we found that the implementation [31] becomes available⁶, and tested it with our machine. Our machine shows a bit slower performance than originally reported numbers⁷; for example, [31]-PK takes about 67.5s and [31]-SK takes 25.5s for online phase when $n=2^{18}$, whereas the original paper reports 41.5s and 10.8s for each case. Our self-implementation of [31] protocol shows even closer performance than their implementation executed on our machine, and hence we decide to report the numbers obtained from our implementation in Table 5.

6 DISCUSSIONS

We provide some further discussions that might lead to future work or some considerations related to other recent works.

Asymptotically better Receiver-Cuckoo. As the secret-shared equality check idea leads to asymptotic improvement for sender-cuckoo protocols from $O(\ln \log n)$ to $O(\ln n + n \log n)$, one may wonder whether the same idea can apply to receiver-cuckoo protocols. However, it is non-trivial because the fundamental logic for the membership check is quite different from the sender-cuckoo case. More precisely, in the sender-cuckoo case, each RPMT problem $1(x \in Y)$ is reduced to an equality check instance $1(t_{s,i} = t_{r,i})$, and this can be easily converted into an equality share generation instance. However, in the receiver-cuckoo case, each RPMT problem $1(x \in Y)$ is reduced to a problem of checking whether $I_X \cap p' = \emptyset$,

⁶https://github.com/alibaba-edu/mpc4j

⁷Conducted on Intel Core i9-9900K with 3.6GHz and 128GB RAM. We had some personal conversation with the authors of [31], and conclude this comes from machine spec difference.

Table 5: Comparison of timings and communications. In all experiments, we assume $\ell_0 = 128$. Offline phase includes base OT, ROT extension, VOLE, and pre-computation for ECC, where ROT extension usually dominates. For each online and total timings, we mark the best value by blue, and the next by grey.

$\ell_0 = 128$		2 ¹⁶			Set Si	Set Size $n = n_x = n_y$ 2^{18}			2^{20}		
		Offline	Online	Total	Offline	Online	Total	Offline	Online	Total	
	[12]	0.22	1.87	2.09	0.63	7.32	7.95	2.34	31.09	33.42	
	[17]	0.25	1.06	1.31	0.63	4.01	4.65	2.36	18.02	20.39	
10Gbps	[31]-PK	0.19	12.88	13.07	0.22	51.30	51.52	0.20	204.8	205.0	
0.2ms	[31]-SK	6.62	3.77	10.39	27.3	15.66	43.00	109.9	62.30	172.2	
	Our-SC	1.21	1.72	2.93	4.02	6.71	10.73	15.18	27.43	42.62	
	Our-RC	0.23	1.06	1.29	0.63	4.02	4.64	2.36	18.22	20.55	
	[12]	0.27	3.29	3.57	0.69	10.40	11.08	2.40	40.57	42.97	
	[17]	0.27	2.50	2.78	0.69	6.90	8.59	2.36	31.69	34.05	
1Gbps	[31]-PK	0.20	14.86	15.06	0.20	59.02	59.22	0.22	234.0	234.2	
40ms	[31]-SK	6.68	9.99	16.66	27.6	23.04	50.61	111.3	75.48	186.8	
	Our-SC	1.31	3.08	4.39	4.06	8.56	12.62	15.4	30.74	46.16	
	Our-RC	0.28	2.16	2.44	0.67	6.42	7.09	2.37	26.35	28.72	
	[12]	0.30	6.65	6.95	0.72	21.15	21.87	2.42	83.20	85.62	
	[17]	0.32	6.96	7.25	0.74	23.91	24.65	2.37	103.7	106.2	
100Mbps	[31]-PK	0.22	16.23	16.45	0.22	60.21	60.43	0.20	242.4	242.6	
80ms	[31]-SK	6.85	17.32	24.17	28.1	36.04	64.17	111.6	111.7	223.4	
	Our-SC	1.37	4.92	6.29	4.13	12.68	16.81	15.3	43.19	58.49	
	Our-RC	0.31	4.80	5.11	0.72	16.41	17.13	2.42	65.95	68.37	
	[12]	0.35	29.91	30.26	0.84	121.0	121.8	2.43	513.1	515.6	
	[17]	0.35	44.71	45.06	0.77	190.2	191.0	2.45	838.6	841.0	
10Mbps	[31]-PK	0.20	23.70	23.90	0.20	91.17	91.37	0.21	362.2	362.4	
80ms	[31]-SK	6.72	34.77	41.49	27.8	106.2	134.0	111.6	395.4	507.0	
	Our-SC	1.45	11.69	13.14	4.26	39.02	43.28	15.5	150.0	165.5	
	Our-RC	0.34	23.70	24.04	0.74	108.0	108.7	2.47	465.6	468.1	
	[12]	0.10	31.27	31.37	0.12	133.2	133.3	0.13	572.2	572.3	
	[17]	0.10	50.38	50.48	0.12	221.1	221.2	0.12	969.6	969.7	
Comm.	[31]-PK	≈ 0	11.58	11.58	≈ 0	46.08	46.08	≈ 0	184.0	184.0	
(MB)	[31]-SK	0.15	23.30	23.45	0.43	94.03	94.45	1.5	374.9	376.4	
	Our-SC	0.22	9.38	9.59	0.25	36.43	36.68	0.26	145.4	145.6	
	Our-RC	0.10	25.52	25.61	0.12	122.3	122.4	0.12	532.8	532.9	

which is difficult to convert into a secret-share manner. Resolving this technical difficulty would be an interesting topic.

Unbalanced Input Sets. Although we mostly consider $n=n_x=n_y$ case in our experiments, sender-cuckoo and receiver-cuckoo have quite different performances when the input set size n_x and n_y are unbalanced. This is because the two protocols have a heavy dependence on the number of hash table bins β , which is $O(n_x)$ in sender-cuckoo versions, and $O(n_y)$ in receiver-cuckoo versions. Therefore, sender-cuckoo versions are better for $n_y \geq n_x$ case, and receiver-cuckoo versions are better for $n_x \geq n_y$.

Round Complexity. Although our main body has almost no interest on round complexity, it is definitely one of important features of cryptographic protocols, and could be another interest. We briefly discuss the round complexities of our interest PSU protocols. First, our optimization for the receiver-cuckoo PSU [17] incurs no additional interaction. However, Our-SC requires more interactions due to the change from PEqT to PESG based on GMW protocol, compared to the original sender-cuckoo PSU [12]. Precisely, PEqT based on OPRF usually requires some constant rounds, but PESG for ℓ -bit strings based on GMW protocol requires $\log(\ell)$ rounds. [31]-SK is even worse in round complexity view, because it evaluates LowMC decryption circuit and comparison circuit using GMW protocol, which is about $20 + \log(\ell)$. Lastly, [31]-PK is the best in

the round complexity view, as it requires only two rounds to obtain RPMT result $\mathbf{1}(x \in Y)$. It could be an interesting direction to lower round complexity, while maintaining low computation and communication cost.

Better OKVS. Recently, another construction of OKVS [27] has been proposed. Although the size of OKVS encoding remains almost similar (about 1.3n), it shows about one (two, resp) order of magnitude improvement on encoding (decoding, resp) time than our exploited 3H-GCT construction [13], according to reported performances. We remark that our proposals still be the best one even considering this advanced OKVS. First, it is already discussed the advantage of Our-RC over [17] and [12] is independent of the instantiation of OKVS. Second, Our-SC would likely still be better than [31], recalling that the computation bottleneck of [31] was not OKVS operations. However, it is true that the OKVS construction of [27] will make not a few changes on the concrete timings in Table 5. We leave additional consideration of [27] as an immediate future work.

REFERENCES

- Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In EUROCRYPT 2015. Springer, 430–454
- [2] Martin R Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2017. LowMC implementation. https://github.com/LowMC/ lowmc.

- [3] Yuriy Arbitman, Moni Naor, and Gil Segev. 2010. Backyard cuckoo hashing: Constant worst-case operations with a succinct representation. In FOCS 2010. IEEE, 787–796.
- [4] Gilad Asharov, Yehuda Lindell, Thomas Schneider, and Michael Zohner. 2013. More Efficient Oblivious Transfer and Extensions for Faster Secure Computation. In CCS 2013. ACM, New York, NY, USA, 535–548.
- [5] Donald Beaver. 1991. Efficient Multiparty Protocols Using Circuit Randomization. In CRYPTO 1991. Springer, 420–432.
- [6] Elette Boyle, Geoffroy Couteau, Niv Gilboa, Yuval Ishai, Lisa Kohl, Peter Rindal, and Peter Scholl. 2019. Efficient Two-Round OT Extension and Silent Non-Interactive Secure Computation. In CCS 2019. 291–308.
- [7] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2014. (Leveled) fully homomorphic encryption without bootstrapping. TOCT 6, 3 (2014), 1–36.
- [8] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-Shared Shuffle. In ASIACRYPT 2020. Springer, Cham, 342–372.
- [9] Melissa Chase and Peihan Miao. 2020. Private Set Intersection in the Internet Setting From Lightweight Oblivious PRF. In CRYPTO 2020. Springer, 34–63.
- [10] Geoffroy Couteau, Peter Rindal, and Srinivasan Raghuraman. 2021. Silver: Silent VOLE and Oblivious Transfer from Hardness of Decoding Structured LDPC Codes. In CRYPTO 2021. Springer, Cham, 502–534.
- [11] Keith Frikken. 2007. Privacy-preserving set union. In ACNS 2007. Springer, 237–252.
- [12] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. 2021. Private Set Operations from Oblivious Switching. In PKC 2021. Springer.
- [13] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2021. Oblivious key-value stores and amplification for private set intersection. In CRYPTO 2021. Springer, 395–425.
- [14] O. Goldreich, S. Micali, and A. Wigderson. 1987. How to Play ANY Mental Game. In STOC 1987. 218–229.
- [15] Dov Gordon, Carmit Hazay, Phi Hung Le, and Mingyu Liang. 2022. More Efficient (Reusable) Private Set Union. Cryptology ePrint Archive, Paper 2022/713. https://eprint.iacr.org/2022/713 https://eprint.iacr.org/2022/713.
- [16] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. 2003. Extending Oblivious Transfers Efficiently. In CRYPTO 2003. Springer, 145–161.
- [17] Yanxue Jia, Shi-Feng Sun, Hong-Sheng Zhou, Jiajun Du, and Dawu Gu. 2022. Shuffle-based Private Set Union: Faster and More Secure. In USENIX 2022. USENIX Association.
- [18] Lea Kissner and Dawn Song. 2005. Privacy-Preserving Set Operations. In CRYPTO 2005. Springer, 241–257.
- [19] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. 2017. Practical Multi-party Private Set Intersection from Symmetric-key Techniques. In CCS 2017. 1257–1272.
- [20] Vladimir Kolesnikov, Mike Rosulek, Ni Trieu, and Xiao Wang. 2019. Scalable private set union from symmetric-key techniques. In ASIACRYPT 2019. Springer, 636–666.
- [21] Kuku 2021. Microsoft Kuku. https://github.com/microsoft/Kuku. Microsoft Research, Redmond, WA...
- [22] Shigeo Mitsunari. 2022. MCL: A portable and fast pairing-based cryptography library. https://github.com/herumi/mcl.
- [23] Payman Mohassel and Saeed Sadeghian. 2013. How to hide circuits in MPC an efficient framework for private function evaluation. In EUROCRYPT 2013. Springer, Cham, 557–574.
- [24] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2019. Spot-light: Lightweight Private Set Intersection from Sparse OT Extension. In CRYPTO 2019. Springer, 401–431.
- [25] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In USENIX 2015. USENIX Association.
- [26] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient Circuit-based PSI with Linear Communication. In EUROCRYPT 2019. Springer, Cham, 122–153.
- [27] Srinivasan Raghuraman and Peter Rindal. 2023. Blazing Fast PSI from Improved OKVS and Subfield VOLE. In CCS 2023.
- [28] Peter Rindal. 2022. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe
- [29] Peter Rindal and Phillipp Schoppmann. 2021. VOLE-PSI: Fast OPRF and Circuit-PSI from Vector-OLE. In EUROCRYPT 2021. Springer, Cham, 901–930.
- [30] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast extension for correlated OT with small communication. In CCS 2020. ACM, New York, NY, USA, 1607–1626.
- [31] Cong Zhang, Yu Chen, Weiran Liu, Min Zhang, and Dongdai Lin. 2023. Optimal Private Set Union from Multi-Query Reverse Private Membership Test. In USENIX 2023. USENIX Association.

A ANOTHER P&S PROTOCOL

There is another P&S proposal [8]. However, this protocol originally aims to improve P&S when the permutation target vector item is quite long. To be precise, when assuming $\ell \geq \kappa$, [8] has a communication cost about

$$(2\ell/\log T + 4\kappa)N\log N$$

where T is an optimization parameter for time-communication trade-off. Clearly, this provides almost $1/\log T$ reduction on communication cost than the protocol of [23] when $\ell \gg \kappa$, but it is rather worse for our interest item lengths; less than κ bits. For more detailed formula, we refer [8, Section 7.2].

B MICRO-BENCHMARKS

Table 6 present micro-benchmarks for our experiments for input set size $n = 2^{20}$. For a detailed procedure for each step, see descriptions in the main body; see Table 1 and Table 2.

Table 6: Micro-benchmarks for our implementations for input set size $n=2^{20}$ and item length $\ell_0=128$. Timings are in second measured in LAN (10Gbps) environment, and communications are in MB.

[12]	Time	Comm		Our-SC	Time	Comm
Offline	2.34	0.12		Offline	15.18	0.12
OPPRF	20.1	52.4		OPPRF	20.48	52.4
P&S	3.57	440.1		PESG	2.14	41.0
PEqT	7.17	37.8		P&S	4.46	10.2
Final OT	0.27	41.9	_	Final OT	0.31	41.9
[17]	Time	Comm		Our-RC	Time	Comm
Offline	2.36	0.12		Offline	2.33	0.12
P&S	5.75	876.9		P&S	3.47	440.1
OPRF	4.46	60.4		OPRF	14.49	60.4
Final OT	0.20	32.2		Final OT	0.27	32.2
[31]-PK	Time	Comm		[31]-SK	Time	Comm
Offline	0.2	0.12		Offline	109.9	1.51
Enc	24.8	-		Enc	25.2	-
Ctxt-OKVS	14.1	83.2		Ctxt-OKVS	4.64	13.0
ReRand & Dec	165.3	64		2PC-Dec	32.2	329.6
Final OT	0.27	32.0		Final OT	0.21	32.1

C DISCUSSIONS ABOUT [15]

We first detail the item length $\ell_0=64$ issue. Assuming $\ell_0=64$, the authors took the RLWE plaintext modulus by $\approx 64-\log n$ -bit assuming permutation-based hashing. However, other protocols assumed $\ell_0=128$, and this is unfair since the RLWE ciphertext modulus linearly depends on the length of RLWE plaintext modulus, and the RLWE ciphertext modulus linearly affects both computation and communication costs. As a rough estimation, it seems that [15] can also use preprocess hashing to compress the initial item length $\ell_0=128$ to $\lambda+2\log n$, and reduce it to $\lambda+\log n$ using permutation-based hashing. However, for example with $n=2^{20}$ and $\lambda=40$, the plaintext bit-length for $\ell_0=128$ increases to 60, which was about 44 for $\ell_0=64$. Then the ciphertext modulus should be about 1.3x times bigger, which also results in the similar overhead on both computation and communication costs.

We proceed to the security issue. For a security proof, the decryption of underlying AHE should not reveal any information about

the homomorphic operations giving the ciphertext. In particular, the returning ciphertext of P(x) should reveal only the inner plaintext P(x), and no more information about x (especially important when P(x)=0). The description of [11] already cared this by letting the sender re-randomize the returning ciphertext, whose procedure is a homomorphic addition of encryption of zero. However, such a simple remedy only works for ElGamal or Paillier AHE. In fact, RLWE-AHE requires slightly more, since it additionally reveals the noise term that could retain some information of x. We found no detailed argument about this in [15], and we believe it should be addressed properly. One folklore solution is a *noise flooding* that covers the noise term by another huge noise, but it requires an additional ciphertext modulus margin of about λ bits.

Finally, we remark that the remedy of aforementioned issues will be likely to require larger ciphertext modulus q than the currently used one. Then the RLWE packing parameter (4096 in the original text) should also increase to maintain the same security level (κ = 128). This results in a further negative effect on performance, as the basic RLWE-related operation performances are super-linear to the RLWE packing parameter.

We concede that those issues do not have a devastating impact on the performance of [15], and the asymptotic behavior will remain the same. Meanwhile, we also believe that those issues are likely to incur a non-negligible difference on concrete performance.

C.1 Comparison with As-Is Reports

At least, we can compare with the current reports of [15] and discuss some implications. Table 7 takes the reported costs for [15] which assumes $\ell_0 = 64$, and compares them with the numbers obtained from our implementation with $\ell_0 = 64$. As [15] provided no setup or offline separation, we also only compare with the total protocol running time.

Table 7: Comparison with as-is [15].

$\ell_0 = 64$		2^{16}	2^{18}	2^{20}
	Our-SC	2.94	10.7	42.5
10Gbps	Our-RC	1.31	4.66	20.1
rodops	[15]-A	3.65	21.1	73.8
	[15]-B	3.65	15.7	64.2
	Our-SC	12.1	38.9	148
10Mbps	Our-RC	20.8	84.1	364
Tolvibps	[15]-A	13.7	40.6	182
	[15]-B	13.7	54.6	221
	Our-SC	7.83	30.4	121
Comm.	Our-RC	20.5	88.8	388
Commi.	[15]-A	12.3	29.4	152
	[15]-B	12.3	49.8	203

First of all, we observe that [15] shows worse performance than shuffle-based PSUs, especially than Our-RC (or [17] also) in a high bandwidth network. This is because [15] is also based on public key operations like [31]-PK. However, it has quite a low communication cost comparable to Our-SC, and hence becomes efficient for a low bandwidth network. Note that, although [15] and [31]-PK

both are based on public key operations, the computational cost is cheaper than [15] because a difference in unit operation is cheaper than [15] (RLWE vs ECC). Therefore, we conclude that [15] could be a promising PSU protocol for low bandwidth, and we hope to provide a complete comparison after the aforementioned issues are addressed.

We also leave some discussion about other item lengths than $\ell_0=64$. First note that [15] has a strong linear dependency on the input length ℓ_0 for both computation and communication, as mentioned when we discuss the input length issue. Meanwhile, Our-SC has a dependency with ℓ_0 only in the final OT phase, and hence the input item length ℓ_0 has a small impact on the communication cost of Our-SC. This concludes that Our-SC would be better than [15] for longer item lengths, and [15] would be better than Our-SC for shorter item lengths. However, recall that Our-RC has a further stronger linear dependency on ℓ_0 , and it requires quite a small communication cost for short item lengths (See Table 3. Thus Our-RC could be a competitive protocol when the item length is short.