Scientific Reports

https://doi.org/10.1038/s41598-026-40531-5

Article in Press

# Towards an improved efficient leakage-resilient enhanced private set union

Received: 26 December 2025

Accepted: 13 February 2026

Published online: 21 February 2026

Cite this article as: Liu Q., Bae J. & Lee J. Towards an improved efficient leakage-resilient enhanced private set union. *Sci Rep* (2026). https://doi.org/10.1038/s41598-026-40531-5

Qiang Liu, JaeYoung Bae & Joon-Woo Lee

We are providing an unedited version of this manuscript to give early access to its findings. Before final publication, the manuscript will undergo further editing. Please note there may be errors present which affect the content, and all legal disclaimers apply.

If this paper is publishing under a Transparent Peer Review model then Peer Review reports will publish with the final article.

ARTICLE IN PRESS

© The Author(s) 2026. **Open Access** This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

ARTICLE IN PRESS

# Towards an Improved Efficient Leakage-resilient Enhanced Private Set Union

**Qiang Liu**[1], **JaeYoung Bae**[1], **and Joon-Woo Lee**[1,*]

[1]School of Computer Science and Engineering, Chung-Ang University, Seoul 06974, South Korea.
[*]jwlee2815@cau.ac.kr

## ABSTRACT

Private Set Union (PSU) enables two parties to compute the union of their input sets without revealing any additional information. Tu et al. (USENIX Security 2025) introduced the state-of-the-art enhanced PSU (ePSU) framework, which strengthens security by preventing during-execution leakage. However, we observe that directly applying hash-to-bin on input sets within their framework introduces subtle but non-negligible privacy risks. In this work, we address this issue by combining oblivious pseudorandom functions (OPRF) with randomized shuffling, which eliminates the privacy leakage caused by direct hash-to-bin usage. Building on the revised framework, we further optimize the ePSU construction by introducing a bidirectional oblivious key-value store (OKVS), significantly reducing both communication and computational overhead. Experimental results show that, compared with the revised ePSU of Tu et al., our protocol achieves a $1.089$–$3.049\times$ reduction in communication cost and a $1.027$–$1.744\times$ runtime speedup.

**Keywords** Enhanced private set union, During-execution leakage, Hash-based leakage, Bidirectional OKVS

## Introduction

As a cryptographic protocol, private set union (PSU) enables two parties, the sender holding set $\mathsf{X}$ and the receiver holding set $\mathsf{Y}$, to securely compute the union $\mathsf{X}\cup\mathsf{Y}$ without revealing anything else. PSU serves as an important privacy preserving tool for computing the union of sets, with significant applications in many areas such as security risk assessment[1–3], IP blacklist and vulnerability data aggregation[4–6], and private ID matching[7–9]. For example, in an application aggregating IP blacklists, two organizations maintaining blacklists wish to compute the union of their lists to minimize vulnerabilities in their infrastructures. Directly sharing the blacklists is insecure because each list is generated according to its owner's detection policy, which must remain confidential. A curious party could infer the other organization's policy from IP addresses in the intersection. Therefore hiding the intersection is essential, and PSU provides exactly this capacity.

In 2019, Kolesnikov et al.[10] proposed the first scalable PSU protocol. The core building block of their construction is the reverse private membership test (RPMT), which privately determines, for each item in $\mathsf{X}$, whether it belongs to $\mathsf{Y}$. The resulting indication bit is then used as the selection bit in an oblivious transfer (OT) protocol[11], such that the receiver obtains a dummy value when $x_i \in \mathscr{Y}$ and obtains $x_i$ otherwise. Finally, the receiver locally computes the union $\mathsf{X}\cup\mathsf{Y}$. Subsequent works[8,12–14] further improved the performance by optimizing the underlying RPMT construction.

However, in 2024, Jia et al.[15] showed that the RPMT+OT framework inherently suffers from during-execution leakage. Briefly, by invoking RPMT, it leaks the intersection-size of sets $\mathsf{X}$ and $\mathsf{Y}$ before protocol completion. In practice, this disclosure may lead to unintended leakage about the items in the intersection $\mathsf{X}\cap\mathsf{Y}$. For instance, a corrupted receiver could abort the execution after learning the intersection size (e.g., blaming a network failure) and request to rerun the protocol. By slightly modifying its input set across multiple executions and observing the resulting intersection sizes, the receiver can infer partial information about the intersection items even under standard rate-limiting mechanisms. This makes such leakage particularly insidious in practical deployments. To mitigate during-execution leakage, they designed a enhanced PSU (ePSU) protocol with enhanced security, but the complexity is superlinear. Recently, in 2025, Tu et al.[16] presented the first linear ePSU by introducing permuted non-membership conditional randomness generation (pnMCRG).

In addition, Chandran et al.[14] observed that directly applying the hash-to-bin on input sets to compute the union can lead to potential privacy leakage, which we call hash-based leakage. Briefly, once the receiver obtains $\mathsf{X}\setminus\mathsf{Y}$, it may infer information about the items that do not participate in the intersection by inspecting unmatched bins in its simple hash table. To mitigate this leakage, they proposed encrypting the input sets using an oblivious pseudorandom function (OPRF)[17], applying hash-to-bin on the encrypted items, and further employing a shuffling step to randomize the ordering. However, as their protocol fundamentally builds upon RPMT-based constructions, it remains vulnerable to during-execution leakage. Under this perspective, several prior works[8,10,12,16] remain vulnerable to hash-based leakage due to their reliance on hash-to-bin, whereas the construction of[13]

ARTICLE IN PRESS

inherently avoids this issue by not incorporating hash-to-bin in its design.

In light of the fact that most existing efficiently scalable PSU protocols suffer from privacy leakage arising either from during-execution leakage or hash-based leakage, we propose a leakage-resilient ePSU. The main contributions of this work can be summarized as follows:

- **Revisiting state-of-the-art ePSU.** We revisit the ePSU framework proposed in[16] and identify both security and performance limitations.
  - **Security.** In their framework, both the sender and the receiver directly apply hash-to-bin techniques to their input sets to construct a cuckoo hash table and a simple hash table, respectively. After obtaining $\mathsf{X} \setminus \mathsf{Y}$, the receiver can further leverage this information together with its simple hash table to infer which items in $\mathsf{Y}$ are not part of the intersection, resulting in hash-based leakage. We eliminate this leakage by incorporating oblivious pseudorandom functions (OPRFs) and a shuffling mechanism.
  - **Performance.** Their ePSU protocol performs computation and communication over a large number of dummy entries, introducing substantial redundant overhead.
- **A leakage-resilient ePSU construction.** We address the hash-based leakage identified in the framework of[16] and, based on a revised design, construct a more efficient and leakage-resilient ePSU protocol. Specifically, both parties first invoke an OPRF to encrypt their input sets, ensuring that identical items are mapped to the same ciphertext while distinct items yield different ciphertexts. The encrypted sets are then processed using hash-to-bin and bidirectional OKVS. The parties derive per-item equality conditional randomness, which coincides if and only if the item placed in the $i$-th cuckoo hash bin corresponds to the item in the $i$-th simple hash bin. To ensure uniformity, random values are generated for empty cuckoo hash bins. Importantly, in our design, the cuckoo hash table is not padded, and the OKVS is constructed only over occupied bins. As a result, no redundant entries are encoded or transmitted, significantly reducing both communication and computational costs. Finally, following the same procedure as in[16], the parties sequentially invoke permuted equality conditional randomness generation (pECRG), non-equality conditional randomness generation (nECRG), and one-time pad encryption, enabling the receiver to obtain the union set $\mathsf{X} \cup \mathsf{Y}$.
- **Evaluation.** We implement our ePSU protocol and compare it with state-of-the-art constructions. Our experimental results demonstrate that, compared with the revised ePSU in[16], our protocol achieves a 1.089–3.049× reduction in communication and a 1.027–1.744× speedup in runtime, depending on the set sizes and network conditions.

## Related Work

We review prior scalable PSU protocols in the presence of semi-honest adversaries and compare our proposed protocol with existing constructions, as summarized in Table 1.

In 2019, Kolesnikov et al.[10] proposed the first scalable PSU protocol. They introduced the RPMT and combined it with OT to compute the set union. RPMT privately determines, for each item $x_i$ ($i \in [n]$) from the sender's set $\mathsf{X}$, whether $x_i \in \mathsf{Y}$: if $x_i \in \mathsf{Y}$, the receiver obtains only a dummy value during the OT phase; otherwise, the receiver obtains the item $x_i$. After $n$ invocations of RPMT and OT, the receiver locally derives the union $\mathsf{X} \cup \mathsf{Y}$. By employing a bucketing technique to group items, their protocol achieves $O(n \log n)$ communication complexity and $O(n \log n \log\log n)$ computational complexity. In 2021, Garimella et al.[8] introduced a PSU protocol based on permuted characteristic (PC), achieving $O(n \log n)$ communication and computational complexities. Their PC protocol is built on Permuted+Share (PS)[22] and provides a batched RPMT functionality. Subsequently, Jia et al.[12] proposed a shuffle-based PSU protocol with performance comparable to that of[8], achieving the same asymptotic complexity. In 2023, Zhang et al.[13] introduced the first PSU protocol with linear complexity. They proposed a new primitive, called multi-query RPMT (mq-RPMT), to efficiently realize batched RPMT, resulting in $O(n)$ communication and computation. In 2024, Jia et al.[15] addressed the during-execution leakage present in prior works[8,10,12,13] by proposing an ePSU protocol with $O(n \log n)$ communication and computational complexities. Their construction relies on oblivious programmable PRFs (OPPRFs)[23], PS, and a newly introduced primitive called batched equality-conditional randomness generation (bECRG). While their protocol achieves runtime comparable to[13] in LAN settings, its communication overhead is approximately 5–6× higher. More recently, Tu et al.[16] proposed the first ePSU protocol with linear complexity, building on their pnMCRG primitive and achieving significant performance improvements over[15].

In addition, Chandran et al.[14] observed that directly applying hash-to-bin techniques to the input sets may lead to privacy leakage. Specifically, once the receiver obtains $\mathsf{X} \setminus \mathsf{Y}$, it may further infer which items in $\mathsf{Y}$ are not part of the intersection by examining unmatched hash bins. To mitigate this issue, they proposed combining OPRF with a combine-and-permute strategy that incorporates a shuffling step. However, their approach relies on realizing mq-RPMT via a circuit-based private

ARTICLE IN PRESS

set intersection protocol[18], which inherently incurs during-execution leakage, while its performance remains comparable to that of[13]. Notably, the hash-based leakage identified in their work also affects several earlier PSU protocols, including those in[8,10,12,16].

Finally, we note that there exists a variant of two-party PSU, commonly referred to as unbalanced PSU[16,19–21], in which the two parties hold asymmetric set sizes and one party's input set is much smaller, i.e., $m \ll n$. In this case, the main design goal is to achieve communication (and ideally computation) that scales linearly with the smaller set size $m$ while remaining essentially independent of the larger set size $n$. In the balanced PSU setting, where the two parties hold input sets of equal size, i.e., $|\mathsf{X}| = |\mathsf{Y}| = n$, and the protocol aims for communication and computation that scale linearly with $n$. Both settings are essential in practice: the balanced model captures the theoretical efficiency limits under symmetric interactions, while the unbalanced model reflects common real-world deployments with resource-constrained clients and powerful servers. Nevertheless, in this paper we primarily concentrate on the balanced setting with equal-size inputs.

| Protocol | Communication | Computation | DEL |
|---|---|---|---|
| Kolesnikov et al.[10]* | $O(n\log n)$ | $O(n\log n\log\log n)$ | Y |
| Garimella et al.[8]* | $O(n\log n)$ | $O(n\log n)$ | Y |
| Jia et al.[12]* | $O(n\log n)$ | $O(n\log n)$ | Y |
| Zhang et al.[13] | $O(n)$ | $O(n)$ | Y |
| Chandran et al.[14] | $O(n)$ | $O(n)$ | Y |
| Jia et al.[15]* | $O(n\log n)$ | $O(n\log n)$ | N |
| Tu et al.[16]* | $O(n)$ | $O(n)$ | N |
| Ours | $O(n)$ | $O(n)$ | N |

**Table 1.** Comparison of scalable PSU protocols in the semi-honest model. "DEL" means the during-execution leakage, "Y" indicates the existence of during-execution leakage, and "N" indicates no during-execution leakage. "*" denotes hash-based leakage caused by directly applying hash-to-bin on the input sets.

ARTICLE IN PRESS

## Preliminaries

### Notation

We let $\kappa$ and $\lambda$ denote the computational and statistical security parameters, respectively. The notation $1^\kappa$ denotes the string of $\kappa$ ones. For any $n \in \mathbb{N}$, we write $[n] = \{1,\ldots,n\}$. For a bit string $b \in \{0,1\}^\ell$, let $b_i$ denote its $i$-th bit. For a set $\mathsf{X} = \{x_1,\ldots,x_n\} \in (\{0,1\}^\ell)^n$, let $x_{i,j}$ denote the $j$-th bit of $x_i$. We use $\approx_c$ to denote computational indistinguishability, and PPT to denote probabilistic polynomial-time. For bit strings $x$ and $y$, $x \oplus y$ denotes the bitwise exclusive-or (XOR). The notation $r \xleftarrow{\$} \{0,1\}^\ell$ indicates that $r$ is sampled uniformly at random from $\{0,1\}^\ell$.

**Parameters:**

- The functionality interacts with two parties, the sender $\mathscr{S}$ with set size $n_1$ and the receiver $\mathscr{R}$ with set size $n_2$, and the simulator *Sim*.

**Functionalities:**

1. Initialize an ideal state $\mathsf{state}_{\mathscr{U}} = \emptyset$ for party $\mathscr{U} \in \{\mathscr{S},\mathscr{R}\}$; if $\mathscr{U}$ is corrupt, *Sim* is allowed to access $\mathsf{state}_{\mathscr{U}}$.
2. Wait for input $\mathsf{X} = \{x_1,\ldots,x_{n_1}\}$ from $\mathscr{S}$, abort if $|\mathsf{X}| \neq n_1$; otherwise, update state $\mathsf{state}_{\mathscr{S}} = \langle \mathsf{X} \rangle$, and send $\langle \mathsf{Request}, \mathscr{S} \rangle$ to *Sim*.
3. Wait for input $\mathsf{Y} = \{y_1,\ldots,y_{n_2}\}$ from $\mathscr{R}$, abort if $|\mathsf{Y}| \neq n_2$; otherwise, update state $\mathsf{state}_{\mathscr{R}} = \langle \mathsf{Y} \rangle$, and send $\langle \mathsf{Request}, \mathscr{R} \rangle$ to *Sim*.
4. Upon obtaining $\langle \mathsf{Request}, \mathsf{OK} \rangle$ from *Sim*, compute $\mathsf{Z} = \mathsf{X} \cup \mathsf{Y}$, and add $\langle \mathsf{Finished} \rangle$ to $\mathscr{S}$'s state $\mathsf{state}_{\mathscr{S}}$ and $\langle \mathsf{Z} \rangle$ to $\mathscr{R}$'s state $\mathsf{state}_{\mathscr{R}}$.
5. Output $\mathsf{Z}$ to $\mathscr{R}$ and $\langle \mathsf{Finished} \rangle$ to $\mathscr{S}$.

**Figure 1.** Enhanced private set union functionality $\mathscr{F}_{\mathrm{ePSU}}^{n_1,n_2}$

ARTICLE IN PRESS

## Building Blocks

**Enhanced Private Set Union.** A PSU protocol that incurs no during execution leakage is regarded as having an enhanced security, and such protocols are referred as enhanced PSU (ePSU). The functionality is shown in Figure 1.

**Oblivious Pseudorandom Function.** An oblivious pseudorandom function (OPRF)[17,24] allows the receiver to input $\{x_1,\ldots,x_n\}$ and obtains PRF values $\{F(key,x_1),\ldots,F(key,x_n)\}$, and the sender obtains a PRF key *key*. The definition of OPRF functionality is shown in shown in Figure 2..

**Parameters:**
- the sender $\mathscr{S}$ and the receiver $\mathscr{R}$.
- A PRF $F:\{0,1\}^{\ell}\times\{0,1\}^{\ell}\to\{0,1\}^{\kappa}$, where $\ell$ is the input length.

**Functionalities:**
1. Wait for input $\{x_1,\ldots,x_n\}$ from $\mathscr{R}$.
2. Sample a random PRF key *key* and compute $\{F(key,x_1),\ldots,F(key,x_n)\}$.
3. Send $\langle\mathsf{Request}\rangle$ to the simulator *Sim*.
4. Upon obtaining $\langle\mathsf{Request},\mathsf{OK}\rangle$ from *Sim*, send the key *key* to $\mathscr{S}$ and $\{F(key,x_1),\ldots,F(key,x_n)\}$ to $\mathscr{R}$.

**Figure 2.** Oblivious pseudorandom function functionality $\mathscr{F}_{\text{OPRF}}$.

**Oblivious Key-Value Store.** A key-value store[25–27] is a data structure that maps a set of keys to their corresponding values, defined as follows:

*Definition 1.* A key-value store (KVS) is parameterized by a set $\mathscr{K}$ of keys and a set $\mathscr{V}$ of values. It consists of two algorithms:

- $\mathsf{Encode}(\{(k_1,v_1),\ldots,(k_n,v_n)\})$: on input a set of $(k_i,v_i)$ key-value pairs, and outputs an object $\mathsf{D}=(d_1,\ldots,d_m)$ (or, with statistically samll probability, an error indicator $\perp$).
- $\mathsf{Decode}(\mathsf{D},k)$: on input an object $\mathsf{D}=(d_1,\ldots,d_m)$, a key $k$, and outputs a value $v$.

*Correctness.* A KVS is correct if, for all $\mathsf{A}\subseteq\mathscr{K}\times\mathscr{V}$ with distinct keys, $(k,v)\in\mathsf{A}$ and $\perp\neq\mathsf{D}\leftarrow\mathsf{Encode}(\mathsf{A})\Longrightarrow\mathsf{Decode}(\mathsf{D},k)=v$.

*Obliviousness.* A KVS is an oblivious KVS (OKVS) if, for all distinct $\{k_1^0,\ldots,k_n^0\}$ and all distinct $\{k_1^1,\ldots,k_n^1\}$, if Encode does not output $\perp$ for $\{k_1^0,\ldots,k_n^0\}$ or $\{k_1^1,\ldots,k_n^1\}$, then $\{\mathsf{D}|v_i\leftarrow\mathscr{V},i\in[n],\mathsf{Encode}_{\mathsf{H}}(\{(k_1^0,v_1),\ldots,(k_n^0,v_n)\})\}\approx_{\mathscr{C}}\{\mathsf{D}|v_i\leftarrow\mathscr{V},i\in[n],\mathsf{Encode}_{\mathsf{H}}(\{(k_1^1,v_1),\ldots,(k_n^1,v_n)\})\}$.

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

**Parameters:**
- Two parties: the sender $\mathscr{S}$ and the receiver $\mathscr{R}$.

**Functionalities:**
1. Initialize an ideal state $\mathsf{state}_{\mathscr{U}}=\emptyset$ for party $\mathscr{U}\in\{\mathscr{S},\mathscr{R}\}$; if $\mathscr{U}$ is corrupt, *Sim* is allowed to access $\mathsf{state}_{\mathscr{U}}$.
2. Wait for input $\{c_1,\ldots,c_n\}$ and a permutation $\pi:[n]\to[n]$ from $\mathscr{S}$ and update state $\mathsf{state}_{\mathscr{S}}=\langle\{c_i\}_{i\in[n]},\pi\rangle$, and send $\langle\mathsf{Request},\mathscr{S}\rangle$ to *Sim*.
3. Wait for input $\{c_1^*,\ldots,c_n^*\}$ from $\mathscr{R}$ and update state $\mathsf{state}_{\mathscr{R}}=\langle\{c_i^*\}_{i\in[n]}\rangle$, and send $\langle\mathsf{Request},\mathscr{R}\rangle$ to *Sim*.
4. Upon obtaining $\langle\mathsf{Request},\mathsf{OK}\rangle$ from *Sim*, generate two random sets $\{u_1,\ldots,u_n\}$ and $\{u_1^*,\ldots,u_n^*\}$, where $u_i=u_i^*$ if $c_{\pi^{-1}(i)}=c^*_{\pi^{-1}(i)}$; otherwise, $u_i\neq u_i^*$.
5. Add $\langle\{u_i\}_{i\in[n]}\rangle$ to state $\mathsf{state}_{\mathscr{S}}$ and $\langle\{u_i^*\}_{i\in[n]}\rangle$ to state $\mathsf{state}_{\mathscr{R}}$.
6. Output $\langle\{u_i\}_{i\in[n]}\rangle$ to $\mathscr{S}$ and $\langle\{u_i^*\}_{i\in[n]}\rangle$ to $\mathscr{R}$.

**Figure 3.** Permuted equality conditional randomness generation functionality $\mathscr{F}_{\text{pECRG}}$

**Permuted Equality Conditional Randomness Generation.** A permuted equality conditional randomness generation (pECRG)[16] enables the sender inputs a permutation $\pi:[n]\to[n]$ and $\{c_1,\ldots,c_n\}$, and the receiver inputs $\{c_1^*,\ldots,c_n^*\}$. Finally,

ARTICLE IN PRESS

the functionality returns the $\{u_1,\ldots,u_n\}$ and $\{u_1^*,\ldots,u_n^*\}$ to the sender and the receiver, respectively, where for $i\in[n]$, $u_i=u_i^*$ if $c_{\pi^{-1}(i)}=c^*_{\pi^{-1}(i)}$; otherwise $u_i\neq u_i^*$. The definition of pECRG functionality is shown in Figure 3.

**Non-Equality Conditional Randomness Generation.** A non-equality conditional randomness generation (nECRG)[16] enables the sender and the receiver input $\{t_1,\ldots,t_n\}$ and $\{\tilde{t}_1,\ldots,\tilde{t}_n\}$, respectively. Finally, they obtain $\{r_1,\ldots,r_n\}$ and $\{\tilde{r}_1,\ldots,\tilde{r}_n\}$, respectively, where $r_i=\tilde{r}_i$ if $t_i\neq\tilde{t}_i$; otherwise, $r_i\neq\tilde{r}_i$. The definition of nECRG functionality is shown in Figure 4.

**Parameters:**

- the sender $\mathscr{S}$, the receiver $\mathscr{R}$, and the simulator *Sim*.

**Functionalities:**

1. Initialize an ideal state $\mathsf{state}_{\mathscr{U}}=\emptyset$ for party $\mathscr{U}\in\{\mathscr{S},\mathscr{R}\}$; if $\mathscr{U}$ is corrupt, *Sim* is allowed to access $\mathsf{state}_{\mathscr{U}}$.
2. Wait for input $\{t_1,\ldots,t_n\}$ from $\mathscr{S}$ and update state $\mathsf{state}_{\mathscr{S}}=\langle\{t_i\}_{i\in[n]}\rangle$, and send $\langle\mathsf{Request},\mathscr{S}\rangle$ to *Sim*.
3. Wait for input $\{\tilde{t}_1,\ldots,\tilde{t}_n\}$ from $\mathscr{R}$ and update state $\mathsf{state}_{\mathscr{R}}=\langle\{\tilde{t}_i\}_{i\in[n]}\rangle$, and send $\langle\mathsf{Request},\mathscr{R}\rangle$ to *Sim*.
4. Upon obtaining $\langle\mathsf{Request},\mathsf{OK}\rangle$ from *Sim*, generate two random sets $\{r_1,\ldots,r_n\}$ and $\{\tilde{r}_1,\ldots,\tilde{r}_n\}$, where $r_i=\tilde{r}_i$ if $t_i\neq\tilde{t}_i$; otherwise, $r_i\neq\tilde{r}_i$.
5. Add $\langle\{r_i\}_{i\in[n]}\rangle$ to state $\mathsf{state}_{\mathscr{S}}$ and $\langle\{\tilde{r}_i\}_{i\in[n]}\rangle$ to state $\mathsf{state}_{\mathscr{R}}$.
6. Output $\langle\{r_i\}_{i\in[n]}\rangle$ to $\mathscr{S}$ and $\langle\{\tilde{r}_i\}_{i\in[n]}\rangle$ to $\mathscr{R}$.

**Figure 4.** Non-equality conditional randomness generation functionality $\mathscr{F}_{\text{nECRG}}$.

**Hash-to-bin.** *Cuckoo hashing* scheme[29] uses $\gamma$ hash functions $h_1,\ldots,h_\gamma:\{0,1\}^*\to[\beta]$ to map n items to bins in a hash table of size $\beta=\varepsilon n$, and each bin contains only one item. When an item $x$ is inserted, $x||k$ (where $k$ denotes the corresponding hash label) is initially hashed into one of the bins $B_{h_1(x)},\ldots,B_{h_\gamma(x)}$. If all designated bins are occupied, then randomly selects one of the occupied bins, evicts the item currently in that bin, and re-inserts it into another bin determined by a different hash function. This process is repeated iteratively until either a vacant bin is found, or the number of evictions reaches a predefined threshold, indicating that insertion into the regular hash table is infeasible. In such cases, the item is placed into a small auxiliary space called the "stash," reserved for overflow items. According to the empirical analysis in[28], by appropriately tuning the parameters $\gamma$ and $\varepsilon$, the stash size can be minimized to zero in most practical scenarios while maintaining a failure probability as low as $2^{-\lambda}$. We summarize the parameter choices from[28] for the case of stash$=0$ as below.

| Hash-failure parameter $\lambda$ | #hash functions $\gamma$ | Table size $\beta$ |
|---|---|---|
| 30 | 2 | $2.4n$ |
| 30 | 3 | $1.2n$ |
| 30 | 4 | $1.07n$ |
| 30 | 5 | $1.04n$ |
| 40 | 2 | $2.4n$ |
| 40 | 3 | $1.27n$ |
| 40 | 4 | $1.09n$ |
| 40 | 5 | $1.05n$ |

*Simple hashing* scheme[29] enables $\gamma$ hash functions $h_1,\cdots,h_\gamma:\{0,1\}^*\to[\beta]$ to map $n$ items into $\beta$ bins $B_1,\cdots,B_\beta$. Each item $x_i||j$ ($i\in[n]$) is added to bins $B_{h_j(x_i)}$, where $j\in[\gamma]$.

## Analyzing and Mitigating Leakage in the Framework of[16]

In this section, we analyze the hash-based leakage in the framework of[16] that arises from directly applying hash-to-bin to the input sets, and we propose a correction to address this issue. Before presenting our analysis, we briefly review the framework of[16], summarizing the relevant details below.

1. The sender uses $\gamma=3$ hash functions to construct cuckoo hash table $X_C=\{X_C[i]\}_{i\in[\beta]}$ and fills empty bins with the dummy value $\perp$, where each bin $X_C[i]$ contains only one item and there is a index $k_{x_i}\in[\gamma]$ such that $X_C[h_{k_{x_i}}(x_i)]=x_i||k_{x_i}$ for each $x_i$;

ARTICLE IN PRESS

2. The receiver uses the same $\gamma = 3$ hash functions to construct simple hash table. For each $k_i \in [\gamma]$, the value $y_i||k_i$ is placed into the bin $Y_{h_{k_i}(y_i)}$. Finally, there are $\beta$ bins $Y_1, \ldots, Y_\beta$, and each bin $Y_i$ contains $|Y_i|$ items.

3. The sender and receiver invoke pnMCRG, where the sender inputs $X_C$ and a permutation $\pi : [\beta] \to [\beta]$ and the receiver inputs $Y_1, \ldots, Y_\beta$. After execution, the sender and receiver obtain $\{u_1, \ldots, u_\beta\}$ and $\{v_1, \ldots, v_\beta\}$, respectively, where if $X_C[\pi^{-1}(i)] \notin Y_{\pi^{-1}(i)}$, then $u_i = v_i$; otherwise $u_i \neq v_i$.

4. The sender computes $\{c_i\}_{i\in[\beta]}$ and sends it to receiver, where $c_i = u_i \oplus (X_C[\pi^{-1}[i]||\mathsf{H}(X_C[\pi^{-1}[i])$, and $\mathsf{H}$ is pre-agreed hash function.

5. The receiver sets $Z = \{\mathsf{Y}\}$. For each $i \in [\beta]$, the receiver computes $m_i||m_i' = v_i \oplus c_i$ and sets $Z = Z \cup m_i$ if $m_i' = \mathsf{H}(m_i)$ and $m_i' \neq \perp$. Finally, the receiver obtains the union set $Z$.

According to step 5, the receiver computes $m_i||m_i'$ and checks whether $m_i = h(m_i')$ and $m_i \neq \perp$. All $m_i$ satisfying these conditions are collected to form the set $\mathsf{X} \setminus \mathsf{Y}$, and the union is then obtained as $Z = (\mathsf{X} \setminus \mathsf{Y}) \cup \mathsf{Y}$. However, the receiver can infer the additional information from $\mathsf{X} \setminus \mathsf{Y}$, thereby causing hash-based leakage. Specifically, a corrupted receiver can exploit this leakage as follows: for each $y \in \mathsf{Y}$, the receiver can locally compute its candidate bin positions $\mathsf{Pos}(y) = \{h_1(y), \ldots, h_\gamma(y)\}$. Upon obtaining $\mathsf{X}\backslash\mathsf{Y}$, it can use hash functions to inspect the bins in which the items of $\mathsf{X}\backslash\mathsf{Y}$ reside. If all of $y$'s candidate bins are already occupied by distinct items from $\mathsf{X}\backslash\mathsf{Y}$, the receiver concludes that $y \notin \mathsf{X}$. Hence, even though $\mathsf{X}\backslash\mathsf{Y}$ is necessary for correctness, its hash structure reveals negative membership information about the sender's set. This attack requires no access to $X_C$ itself, only to the public hash functions and the observable distribution of $\mathsf{X}\backslash\mathsf{Y}$ across bins.

Formally, let $B_i$ denote the set of items mapped to bin $i$ by the sender, where $B_i = \{x \in \mathsf{X} : \exists j \in [\gamma], h_j(x) = i\}$. After learning $\mathsf{X}\backslash\mathsf{Y}$, the receiver knows the occupancy set $\mathsf{Occ}_X = \{i : \exists x \in \mathsf{X}\backslash\mathsf{Y}, h_j(x) = i\}$. Then for any $y \in \mathsf{Y}$,

$$\forall j,\ h_j(y) \in \mathsf{Occ}_X \Longrightarrow y \notin \mathsf{X}.$$

Thus the receiver learns a leakage function

$$\mathscr{L}_{\mathscr{R}}^{\mathsf{ePSU}}(\mathsf{X},\mathsf{Y}) = \{y \in \mathsf{Y} : \forall j,\ h_j(y) \in \mathsf{Occ}_X\},$$

which equals the subset of $\mathsf{Y}$ that can be certified as not in $\mathsf{X}$. In addition,[14] shows that if the receiver knows $|\mathsf{X} \cap \mathsf{Y}|$, $|\mathsf{X}\backslash\mathsf{Y}|$ and $|\mathsf{Y}|$, the probability that it can compute some $y \notin \mathsf{X} \cap \mathsf{Y}$ is $\mathsf{pr} = 1 - \frac{|\mathsf{X}\cap\mathsf{Y}|}{|\mathsf{Y}|} \cdot (\frac{2\beta}{3(\beta-1)})^{|\mathsf{X}\backslash\mathsf{Y}|}$. Then, we can know $\Pr[\mathscr{L}_{\mathscr{R}}^{\mathsf{ePSU}}(\mathsf{X},\mathsf{Y})] = \mathsf{pr}$. Evidently, $|\mathsf{X} \cap \mathsf{Y}|$ equals the number of items satisfying $m_i \neq h(m_i')$ minus $(\beta - |\mathsf{X}|)$, where $(\beta - |\mathsf{X}|)$ denotes the number of dummy values $\perp$ in the cuckoo hash table, and $|\mathsf{X} \setminus Y|$ corresponds to the number of items satisfying $m_i = h(m_i')$ and $m_i \neq \perp$. Consequently, after completing step 5, the receiver can directly derive these quantities.

To address this issue, Chandran et al.[14] proposed encrypting the input sets using an OPRF before applying the hash-to-bin technique, thereby replacing the direct hash-to-bin operation on plaintext inputs. This approach mitigates the hash-based leakage that arises when the receiver exploits $\mathsf{X} \setminus \mathsf{Y}$ to infer items $y \notin \mathsf{X}$. In addition, they incorporate a shuffling mechanism to randomize the protocol output, preventing the receiver from inferring whether the sender's $i$-th item corresponds to the $i$-th bin in its simple hash table. We adopt the suggestion of[14] and revise the framework of[16] accordingly, as described below.

(1) the sender and the receiver invoke OPRF, where the receiver inputs $\mathsf{Y} = \{y_1, \ldots, y_n\}$ and obtains $\{F(key, x_1), \ldots, F(key, x_n)\}$, while the sender obtains a PRF key $key$.

(2) The sender uses $\gamma = 3$ hash functions to construct cuckoo hash tables $(X_C = \{X_C[i]\}_{i\in[\beta]}, \tilde{X}_C = \{\tilde{X}_C[i]\}_{i\in[\beta]})$ and fills empty bins with the dummy values $(\perp, \perp')$, where each cuckoo hash bin contains only one item and there is a index $k_{x_i} \in [\gamma]$ such that $X_C[h_{k_{x_i}}(F(key, x_i))] = x_i||k_{x_i}$ and $\tilde{X}_C[h_{k_{x_i}}(F(key, x_i))] = F(key, x_i)||k_{x_i}$ for each $x_i$;

(3) The receiver uses the same $\gamma = 3$ hash functions to construct simple hash table. For each $k_i \in [\gamma]$, the value $F(key, y_i)||k_i$ is placed into the bin $Y_{h_{k_i}(F(key,y_i))}$. Finally, there are $\beta$ bins $Y_1, \ldots, Y_\beta$, and each bin $Y_i$ contains $|Y_i|$ items.

(4) The sender and receiver invoke pnMCRG, where the sender inputs $\tilde{X}_C$ and a permutation $\pi : [\beta] \to [\beta]$ and the receiver inputs $Y_1, \ldots, Y_\beta$. After execution, the sender and receiver obtain $\{u_1, \ldots, u_\beta\}$ and $\{v_1, \ldots, v_\beta\}$, respectively, where if $\tilde{X}_C[\pi^{-1}(i)] \notin Y_{\pi^{-1}(i)}$, then $u_i = v_i$; otherwise $u_i \neq v_i$.

(5) The sender computes $\{c_i\}_{i\in[\beta]}$ and sends it to receiver, where $c_i = u_i \oplus (X_C[\pi^{-1}[i]||\mathsf{H}(X_C[\pi^{-1}[i])$, and $\mathsf{H}$ is pre-agreed hash function.

ARTICLE IN PRESS

(6) The receiver sets $Z = \{\mathsf{Y}\}$. For each $i \in [\beta]$, the receiver computes $m_i || m_i' = v_i \oplus c_i$ and sets $Z = Z \cup m_i$ if $m_i' = \mathsf{H}(m_i)$ and $m_i' \neq \perp$. Finally, the receiver obtains the union set $Z$.

Note that in[16], pnMCRG is defined as follows: the sender inputs a permutation $\pi : [n] \to [n]$ and a set $\{x_1, \ldots, x_n\}$, while the receiver inputs $n$ sets $Y_1, \ldots, Y_n$. After execution, the sender and the receiver obtain $\{u_1, \ldots, u_n\}$ and $\{v_1, \ldots, v_n\}$, respectively, such that for each $i \in [n]$, if $x_{\pi^{-1}(i)} \notin Y_{\pi^{-1}(i)}$ then $u_i = v_i$, and otherwise $u_i \neq v_i$. In particular, the permutation $\pi$ induces a random re-indexing of the sender's items/bins before any equality test results become observable to the receiver: the receiver only sees outputs in the permuted order and thus cannot correlate an output position $i$ with the original index of $x_j$ (or the corresponding hash bin) in a predictable way. Hence, pnMCRG already provides the required permutation-based shuffling that we rely on to break index-level linkability. In our revised framework, for each $y \in \mathsf{Y}$, the receiver's candidate bin positions $\mathsf{Pos}(y) = \{h_1(F(key, y)), \ldots, h_\gamma(F(key, y))\}$. Each bin $B_i$ is defined as $B_i = \{x \in \mathsf{X} : \exists j \in [\gamma], h_j(F(key, x)) = i\}$, and the corresponding occupancy set is defined as $\mathsf{Occ}_X = \{i : \exists x \in \mathsf{X} \backslash \mathsf{Y}, h_j(F(key, x)) = i\}$. Since the OPRF ensures that the receiver has no knowledge of the PRF key $key$, it cannot compute $F(key, x)$ for any $x \in \mathsf{X} \setminus \mathsf{Y}$. Consequently, the receiver is unable to construct or evaluate $\mathsf{Occ}_X$, i.e., it cannot determine whether $h_j(F(key, y)) \in \mathsf{Occ}_x$. As a result, the receiver cannot infer which items belong to $\mathsf{Y}$ but not to $\mathsf{X}$.

Following, a potential concern is whether a receiver could accumulate information across multiple protocol runs (possibly with adaptively chosen inputs) and thereby recover patterns about $\mathsf{Occ}_X$. Our revised framework mitigates this as well because the OPRF key $key$ is freshly sampled in each execution. Consequently, for any fixed item $z$, the encodings $F(key, z)$ in different runs are computationally unlinkable from the receiver's perspective. Hence, the hash-to-bin placements derived from $h_j(F(key, z))$ are also re-randomized across runs, and no stable "bin-occupancy pattern" persists that could be correlated across executions. In particular, even if the receiver repeats the protocol and adaptively modifies $\mathsf{Y}$, it still cannot compute $F(key, x)$ for $x \in \mathsf{X} \setminus \mathsf{Y}$ in any run, nor can it link the (unknown) placements of $\mathsf{X} \setminus \mathsf{Y}$ across runs due to the fresh OPRF re-randomization. Therefore, repeated executions do not enable additional hash-based inference beyond what is captured by our semi-honest security model.

Additionally, in the revised ePSU framework, besides the invocation of pnMCRG and the sender's transmission of ciphertexts $\{c_i\}_{i \in [\beta]}$, an additional OPRF invocation is introduced. The security of the OPRF ensures that this interaction is computationally indistinguishable for any corrupted party, while the other two phases have already been proven computationally indistinguishable in[16]. Hence, the revised protocol is secure in the semi-honest model.

## Our Improved ePSU based on Bidirectional OKVS

After addressing the potential hash-based leakage in the framework of[16], we further investigate the detailed protocol design of their ePSU construction under the revised framework. In this section, we introduce an optimization based on a bidirectional OKVS, which significantly improves the efficiency of their protocol. We then formally present our optimized ePSU protocol, followed by a detailed security analysis and a complexity analysis.

### Technical Overview

In their framework, hash-to-bin and pnMCRG serve as the core components. The pnMCRG can be further decomposed into three building blocks: membership conditional randomness generation (MCRG), pECRG, and nECRG. The MCRG is defined as follows: the sender inputs $\{x_1, \ldots, x_n\}$, while the receiver inputs $n$ sets $Y_1, \ldots, Y_n$. After execution, they obtain $\{m_1, \ldots, m_n\}$ and $\{\alpha_1, \ldots, \alpha_n\}$, respectively, such that for each $i \in [n]$, if $x_i \in Y_i$, then $m_i = \alpha_i$; otherwise, $m_i \neq \alpha_i$. Our optimization mainly focuses on improving the hash-to-bin and MCRG components.

As described by our revised framework, the sender and receiver construct the cuckoo hash tables $(X_C, \tilde{X}_C)$ and simple hash table $Y_1, \ldots, Y_\beta$ in steps (1–3), respectively. In step (4), they invoke pnMCRG, where the sender provides $\pi$ and $\tilde{X}_C$, while the receiver provides $Y_1, \ldots, Y_\beta$. Internally, pnMCRG first executes MCRG and then sequentially runs pECRG and nECRG before producing its outputs. Since our improvements primarily focus on the MCRG component of their pnMCRG framework, we begin with a brief description of their MCRG protocol. Figure 5 illustrates its workflow within the ePSU protocol.

As shown in Figure 5, the sender and receiver first invoke a batch OPRF (bOPRF)[30]. The sender inputs $\{\tilde{X}_C[i]\}_{i \in [\beta]}$, after which the receiver obtains $\beta$ PRF keys $\{k_i\}_{i \in [\beta]}$ and the sender obtains $\{w_i\}_{i \in [\beta]}$, where $w_i = F(k_i, \tilde{X}_C[i])$. Next, the receiver constructs an OKVS $D = \mathsf{Encode}(\{P_i\}_{i \in [\beta]})$, where each $P_i = \{(Y_i[j], \tilde{w}_j^i \oplus \alpha_i)\}_{j \in |Y_i|}$ and $\alpha_i$ is a random value, $\tilde{w}_j^i = F(k_i, Y_i[j])$. The receiver then sends $D$ to the sender. For each $i \in [\beta]$, the sender computes $m_i = \mathsf{Decode}(D, \tilde{X}_C[i]) \oplus w_i$. Finally, the sender and receiver obtain their respective MCRG outputs: $\{m_i\}_{i \in [\beta]}$ and $\{\alpha_i\}_{i \in [\beta]}$. By construction, if $\tilde{X}_C[i] \in Y_i$, then $m_i = \mathsf{Decode}(D, \tilde{X}_C[i]) \oplus w_i = w_i \oplus \alpha_i \oplus w_i = \alpha_i$; otherwise, $\mathsf{Decode}(D, \tilde{X}_C[i])$ produces a random value, making $m_i$ a random, meaningless output.

Since the hash-to-bin is employed, the sender's cuckoo hash tables inevitably contain $\beta - n$ padded dummy slots. In the MCRG protocol illustrated in Figure 5, the bOPRF is invoked over all $\beta$ slots, causing these dummy positions to incur

ARTICLE IN PRESS

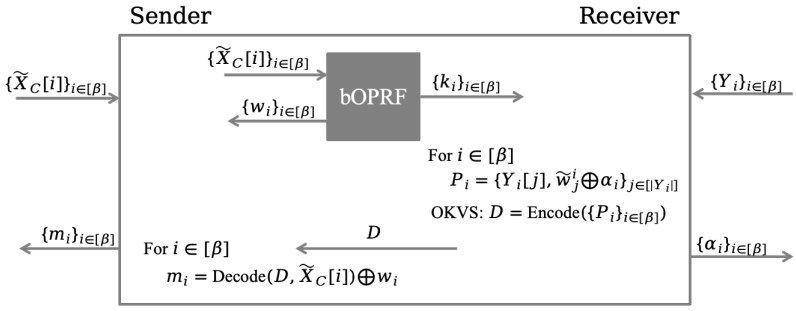



**Figure 5.** The protocol workflow of MCRG of ePSU in[16]. Note: $w_i = F(k_i, \tilde{X}_C[i])$; $\tilde{w}^i_j = F(k_i, Y_i[j])$.

additional computational and communication overhead. According to the bOPRF protocol described in[30], which represents the most widely used implementation, processing $\beta$ inputs of $\kappa$ bits each results in approximately $2\kappa\beta$ bits of communication (excluding the base OT cost), of which $2\kappa(\beta - n)$ bits are redundant. Our optimization aims to exclude these dummy slots from the active execution path while still ensuring the correctness of the outputs $\{m_i\}_{i\in[\beta]}$ and $\{\alpha_i\}_{i\in[\beta]}$. The dummy slots are instead treated as local placeholders, thereby avoiding unnecessary cost; in the final output, the corresponding $m_i$ values can simply be replaced with random values.

Building upon this optimization insight, we observe that the bOPRF in Figure 5 is in fact unnecessary. If both parties attach identical random tags to the same keys, then locally XORing the returned values cancels out the matching tags, yielding a fixed, known value; for non-matching items, the results remain pseudorandom. This observation motivates our bidirectional OKVS-based approach: the sender writes an individual key–value pair $(\tilde{X}_C[i], r_i)$ for each real item $\tilde{X}_C[i] \neq \perp'$ to construct an OKVS object D. The receiver then decodes its candidate keys $F(k_i, y_i)$ via $\text{Decode}(\mathsf{D}, F(k_i, y_i))$, XORs the outputs with local randomness, re-encodes the results into another OKVS object $\mathsf{D}^*$, and sends it back. Finally, the sender decodes using its own $\tilde{X}_C[i]$ (if $\tilde{X}_C[i] \neq \perp'$) and XORs the corresponding $r_i$ to determine the outcome. This design eliminates the need for bOPRF entirely, relying only on the obliviousness and randomness of OKVS. The workflow of our optimized scheme is illustrated in Figure 6.

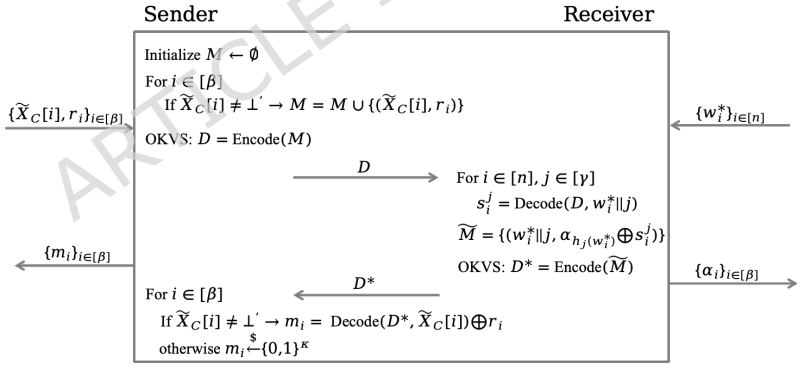



**Figure 6.** Our improved workflow based on MCRG

As shown in Figure 6, the sender constructs an OKVS data structure $\mathsf{D} = \text{Encode}(M)$, where $M = \{(\tilde{X}_C[i], r_i) \mid \tilde{X}_C[i] \neq \perp'\}$ is the set of key–value pairs corresponding to all non-dummy entries. The sender then sends D to the receiver. For each $i \in [n]$ and $j \in [\gamma]$, the receiver computes $s^j_i = \text{Decode}(\mathsf{D}, w^*_i \| j)$, where $w^*_i = F(\text{key}, y_i)$. If $w^*_i \| j$ equals a key contained in $M$, then $s^j_i$ equals the corresponding value in $M$; otherwise, $s^j_i$ is a random value. The receiver then constructs a new set of key–value pairs $\tilde{M} = \{(w^*_i \| j, \alpha_{h_j(w^*_i)} \oplus s^j_i)\}_{i\in[n], j\in[\gamma]}$, generates an OKVS object $\mathsf{D}^* = \text{Encode}(\tilde{M})$, and sends it to the sender. For each $i \in [\beta]$, if $\tilde{X}_C[i] \neq \perp'$, the sender computes $m_i = \text{Decode}(\mathsf{D}^*, \tilde{X}_C[i]) \oplus r_i$; otherwise, $m_i \xleftarrow{\$} \{0,1\}^\kappa$. If $\tilde{X}_C[i]$ equals a key in $\tilde{M}$, for instance $\tilde{X}_C[i] = w^*_k \| k'$, then $s^{k'}_k = r_i$, which implies $\text{Decode}(\mathsf{D}^*, \tilde{X}_C[i]) = \alpha_{h_{k'}(w^*_k)} \oplus s^{k'}_k = \alpha_i \oplus r_i \Longrightarrow m_i = \alpha_i$. Otherwise, if $\tilde{X}_C[i]$ does not match any key in $\tilde{M}$, then $\text{Decode}(\mathsf{D}^*, \tilde{X}_C[i])$ yields a random value, and consequently $m_i$ is also random. Therefore, our improved scheme correctly outputs $\{m_i\}_{i\in[\beta]}$ and $\{\alpha_i\}_{i\in[\beta]}$. Moreover, unlike the original construction, our scheme does not require the receiver to explicitly build a simple hash table. Instead, during the generation of $\tilde{M}$, the receiver simply uses the hash functions $h_j$ ($j \in [\gamma]$) to compute the index $h_j(w^*_i)$ for each random value $\alpha_{h_j(w^*_i)}$, thereby determining the

ARTICLE IN PRESS

corresponding position efficiently.

**Parameters:**

- Two parties: the sender $\mathscr{S}$, the receiver $\mathscr{R}$.
- An OKVS scheme (Encode, Decode).
- Hash functions $h_1,\ldots,h_\gamma : \{0,1\}^\ell \to [\beta]$.
- A cuckoo hash table based on $h_1,\ldots,h_\gamma$, with $\beta = \varepsilon \cdot n$ bins, stash size is 0.
- A collision-resistant hash function: $\mathsf{H} : \{0,1\}^* \to \{0,1\}^{\ell_1}$, where $\ell_1 \geq \lambda + 2\log \varepsilon n$.

**Inputs:**

- The sender $\mathscr{S}$: $\mathsf{X} = \{x_1,\ldots,x_n\} \in (\{0,1\}^\ell)^n$.
- The receiver $\mathscr{R}$: $\mathsf{Y} = \{y_1,\ldots,y_n\} \in (\{0,1\}^\ell)^n$.

**Protocol:**

1. $\mathscr{S}$ and $\mathscr{R}$ invoke $\mathscr{F}_{\text{OPRF}}$:
   - $\mathscr{R}$ inputs $\mathsf{Y} = \{y_1,\ldots,y_n\}$.
   - $\mathscr{S}$ and $\mathscr{R}$ obtain key $key$ and $\{w_1^*,\ldots,w_n^*\}$, respectively, where $w_i^* = F(key, y_i)$.
2. The sender uses $\gamma = 3$ hash functions to construct cuckoo hash tables $X_C = \{X_C[i]\}_{i\in[\beta]}$ and $\tilde{X}_C = \{\tilde{X}_C[i]\}_{i\in[\beta]}$), and fills empty bins with the dummy values $\perp$ and $\perp'$, respectively, where each cuckoo hash bin contains only one item and there is a index $k_{x_i} \in [\gamma]$ such that $X_C[h_{k_{x_i}}(F(key,x_i))] = x_i||k_{x_i}$ and $\tilde{X}_C[h_{k_{x_i}}(F(key,x_i))] = F(key,x_i)||k_{x_i}$ for each $x_i$;
3. $\mathscr{S}$ chooses $\beta$ random values $\{r_1,\ldots,r_\beta\} \xleftarrow{\$} (\{0,1\}^\kappa)^\beta$. $\mathscr{S}$ sets $M = \{(\tilde{X}_C[i], r_i) \mid \tilde{X}_C[i] \neq \perp'\}$ and computes an OKVS structure $\mathsf{D} = \mathsf{Encode}(M)$. Then, $\mathscr{S}$ sends $\mathsf{D}$ to $\mathscr{R}$.
4. $\mathscr{R}$ randomly chooses $\beta$ values $\{\alpha_1,\ldots,\alpha_\beta\} \xleftarrow{\$} (\{0,1\}^\kappa)^\beta$ and computes the set $\{s_1^j,\ldots,s_n^j\}_{j\in[\gamma]}$, where $s_i^j = \mathsf{Decode}(\mathsf{D}, w_i^*||j)$ for $i \in [n]$. $\mathscr{R}$ sets $\tilde{M} = \{(w_i^*||j,\ \alpha_{h_j(w_i^*)} \oplus s_i^j)\}_{i\in[n],j\in[\gamma]}$ and computes an OKVS structure $\mathsf{D}^* = \mathsf{Encode}(\tilde{M})$. Then, $\mathscr{R}$ sends $\mathsf{D}^*$ to $\mathscr{S}$.
5. $\mathscr{S}$ computes the set $\{m_1,\ldots,m_\beta\}$, where for $i \in [\beta]$, $m_i = \mathsf{Decode}(\mathsf{D}^*, \mathsf{X_C}[i]) \oplus r_i$ if $\mathsf{X_C}[i] \neq \perp$; otherwise $m_i \xleftarrow{\$} \{0,1\}^\kappa$.
6. $\mathscr{S}$ chooses a permutation function $\pi : [\beta] \to [\beta]$. $\mathscr{S}$ and $\mathscr{R}$ invoke $\mathscr{F}_{\text{pECRG}}$
   - $\mathscr{S}$ inputs $\{m_1,\ldots,m_\beta\}$ and $\pi$, and $\mathscr{R}$ inputs $\{\alpha_1,\ldots,\alpha_\beta\}$.
   - $\mathscr{S}$ and $\mathscr{R}$ obtain $\{c_1,\ldots,c_\beta\}$ and $\{c_1^*,\ldots,c_\beta^*\}$, respectively, where $c_i = c_i^*$ if $m_{\pi^{-1}(i)} = \alpha_{\pi^{-1}(i)}$; otherwise, $c_i \neq c_i^*$.
7. $\mathscr{S}$ and $\mathscr{R}$ invoke $\mathscr{F}_{\text{nECRG}}$:
   - $\mathscr{S}$ inputs $\{c_1,\ldots,c_\beta\}$, and $\mathscr{R}$ inputs $\{c_1^*,\ldots,c_\beta^*\}$.
   - $\mathscr{S}$ and $\mathscr{R}$ obtain $\{u_1,\ldots,v_\beta\}$, $\{v_1,\ldots,v_\beta\}$, respectively, where $u_i = v_i^*$ if $c_i \neq c_i^*$; otherwise, $u_i \neq v_i$.
8. $\mathscr{S}$ computes and sends $\{ct_i\}_{i\in[\beta]} = \{u_i \oplus (X_C[\pi^{-1}(i)]||\mathsf{H}(X_C[\pi^{-1}(i)]))\}_{i\in[\beta]}$ to $\mathscr{R}$.
9. $\mathscr{R}$ sets $\mathsf{Z} \leftarrow \{\mathsf{Y}\}$. For $i \in [\beta]$, $\mathscr{R}$ computes $m_i||m^* = ct_i \oplus v_i$, $\mathscr{R}$ sets $\mathsf{Z} = \mathsf{Z} \cup \{m_i\}$ if $m_i^* = \mathsf{H}(m_i)$.
10. $\mathscr{R}$ outputs $\mathsf{Z} = \mathsf{Y} \cup \mathsf{Z}$ and $\mathscr{S}$ outputs $\langle \mathsf{Finished} \rangle$.

**Figure 7.** Our enhanced private set union protocol $\Pi_{\text{ePSU}}$

From a performance perspective, the original protocol shown in Figure 5 incurs $2\beta\kappa$ bits of communication from the bOPRF and $(1+\sigma)n\gamma\kappa$ bits from the receiver's transmission of the OKVS data structure D. Hence, the total communication cost is $2\beta\kappa + (1+\sigma)n\gamma\kappa$ bits. Note that for the encoding of key-value pairs, the OKVS in[27] achieves an optimal value of $\sigma = 0.03$. In contrast, our improved scheme requires only two OKVS transmissions, resulting in a total communication of $(\gamma+1)(1+\sigma)n\kappa$ bits. This yields a communication reduction of $2\beta\kappa - (1+\sigma)n\kappa$ bits. For example, setting $\sigma = 0.05$, $\gamma = 3$, and $\beta = 1.27n$, our optimization achieves a reduction of approximately $1.51n\kappa$ bits.

After the sender and the receiver obtain $\{m_i\}_{i\in[\beta]}$ and $\{\alpha_i\}_{i\in[\beta]}$, we simply follow the subsequent steps of our revised ePSU framework to complete the remaining procedure. Specifically, the sender and receiver first run pECRG with inputs a permutation $\pi$, $\{m_1,\ldots,m_\beta\}$ and $\{\alpha_1,\ldots,\alpha_\beta\}$. They obtain outputs $\{c_1,\ldots,c_\beta\}$ and $\{c_1^*,\ldots,c_\beta^*\}$, respectively, where $c_i = c_i^*$ if $m_{\pi^{-1}(i)} = \alpha_{\pi^{-1}(i)}$, and $c_i \neq c_i^*$ otherwise. They then run nECRG on inputs $\{c_1,\ldots,c_\beta\}$ and $\{c_1^*,\ldots,c_\beta^*\}$ to obtain $\{u_1,\ldots,u_\beta\}$ and $\{v_1,\ldots,v_\beta\}$, respectively, where $u_i \neq v_i$ if $c_i = c_i^*$, and $u_i = v_i$ otherwise. Sender then sends $\{ct_i\}_{i\in[\beta]}$ to receiver, where

ARTICLE IN PRESS

$ct_i = u_i \oplus (\mathsf{X}_C[\pi^{-1}(i)] || \mathsf{H}(\mathsf{X}_C[\pi^{-1}(i)]))$. Receiver computes $m_i || m_i^* = ct_i \oplus v_i$, and sets $\mathsf{Z} = \mathsf{Z} \cup \{m_i\}$ if $m_i^* = \mathsf{H}(m_i)$; otherwise, nothing. Finally, the receiver outputs the union $\mathsf{Z} \cup \mathsf{Y}$.

## Protocol Details of Our ePSU

We give our improved formal ePSU protocol that realizes the functionality $\mathscr{F}_{\text{ePSU}}^{n_1,n_2}$ in Figure 7.

**Security.** We state the security of our $\Pi_{\text{ePSU}}$ in Theorem 1, and the security proof is shown as follows.

**Theorem 1.** *The protocol $\Pi_{ePSU}$ shown in Figure 7 securely computes $\mathscr{F}_{ePSU}^{n_1,n_2}$ functionality shown in Figure 1 in the $\{\mathscr{F}_{nECRG}, \mathscr{F}_{pECRG}\}$-hybrid model, against semi-honest adversaries, provided a secure OKVS scheme.*

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

*Corrupted sender: Sim* simulates a real execution in which the sender is corrupted. Since $\mathscr{A}$ is semi-honest, *Sim* can obtain the input $\mathsf{X}$ of the sender directly, and externally send $\mathsf{X}$ to $\mathscr{F}_{\text{ePSU}}^{n_1,n_2}$, and then receive $\langle \mathsf{Request}, \mathscr{S} \rangle$. *Sim* randomly selects a key *key* to simulate the execution of $\Pi_{\text{OPRF}}$. *Sim* computes a random OKVS $\mathsf{D}^*$ by selecting random key-value pairs. When receiving a set $\{m_i\}_{i\in[\beta]}$ and permutation $\pi$, the input of $\Pi_{\text{pECRG}}$, from $\mathscr{A}$, *Sim* checks if it is a permutation of $\beta$ items. If so, *Sim* randomly selects $\{c_i\}_{i\in[\beta]}$ and simulates the execution of $\Pi_{\text{pECRG}}$. After receiving $\{c_i\}_{i\in[\beta]}$, the input of $\Pi_{\text{nECRG}}$, from $\mathscr{A}$, *Sim* randomly selects $\{u_i\}_{i\in[\beta]}$ and simulates the execution of $\Pi_{\text{nECRG}}$. After obtaining $\{ct_i\}_{i\in[\beta]}$, *Sim* sends $\langle \mathsf{Request}, \mathsf{OK} \rangle$ to $\mathscr{F}_{\text{ePSU}}^{n_1,n_2}$.

We argue that the output of *Sim* are indistinguishable from the real view of $\mathscr{S}$ by the following hybrids:

$\mathsf{Hybrid}_0$: The view of the sender in the real protocol.

$\mathsf{Hybrid}_1$: Same as $\mathsf{Hybrid}_0$, except that the output of $\Pi_{\text{OPRF}}$ is replaced by *key* chosen by *Sim*, and runs the $\Pi_{\text{OPRF}}$ simulator to produce the simulated view for $\mathscr{S}$. The security of protocol $\Pi_{\text{OPRF}}$ guarantees that the view in simulation is computationally indistinguishable from the view in $\mathsf{Hybrid}_0$.

$\mathsf{Hybrid}_2$: Same as $\mathsf{Hybrid}_1$, except for computing a random OKVS $\mathsf{D}^*$ by *Sim*. Briefly, this simulation is indistinguishable for the following reasons: The randomness of $\{\alpha_{h_j(w_i^*)} \oplus s_i^j\}_{i\in[n],\, j\in[\gamma]}$ is indistinguishable from random, and the obliviousness and randomness of OKVS guarantee that the $\mathsf{D}^*$ is distributed uniformly.

In $\mathsf{Hybrid}_1$, the sender first samples independent random labels $\{r_i\}_{i\in[\beta]} \xleftarrow{\$} \{0,1\}^\kappa$ and sends $\mathsf{D} = \mathsf{Encode}(M)$ where $M = \{(\tilde{X}_C[i], r_i)\}_{\tilde{X}_C[i]\neq\perp'}$. Upon receiving $\mathsf{D}$, the receiver samples independent masks $\{\alpha_t\}_{t\in[\beta]} \xleftarrow{\$} \{0,1\}^\kappa$, and for each $i \in [n]$ and $j \in [\gamma]$ computes $s_i^j \leftarrow \mathsf{Decode}(\mathsf{D}, w_i^* \| j)$, and then sets the encoded value $\alpha_{h_j(w_i^*)} \oplus s_i^j$. Finally, the receiver returns $\mathsf{D}^* = \mathsf{Encode}(\tilde{M})$ for $\tilde{M} = \{(w_i^* \| j,\, \alpha_{h_j(w_i^*)} \oplus s_i^j)\}_{i\in[n],j\in[\gamma]}$. We argue that $\{\alpha_{h_j(w_i^*)} \oplus s_i^j\}_{i,j}$ is computationally indistinguishable from uniform from the sender's view and is independent of $\mathsf{Y}$. Fix any $(i,j)$. There are two cases:

(i) If $w_i^* \| j$ equals some key $\tilde{X}_C[q]$, then by OKVS correctness $s_i^j = r_q$. Since $\alpha_{h_j(w_i^*)}$ is uniform and independent of $r_q$, the value $v_i^j = \alpha_{h_j(w_i^*)} \oplus r_q$ is uniformly random (a one-time pad masking of $r_q$).

(ii) Otherwise, $w_i^* \| j$ is not in the key set of $M$. By the randomness of OKVS, the decoding output $s_i^j = \mathsf{Decode}(\mathsf{D}, w_i^* \| j)$ is computationally indistinguishable from a uniform random value from the sender's perspective. XORing with an independent uniform mask $\alpha_{h_j(w_i^*)}$ preserves randomness, hence $\alpha_{h_j(w_i^*)} \oplus s_i^j$ is computationally indistinguishable from uniform.

Moreover, this holds jointly: for any fixed index $t \in [\beta]$, consider all pairs $(i,j)$ such that $h_j(w_i^*) = t$. Their values satisfy $\alpha_{h_j(w_i^*)} \oplus s_i^j = \alpha_t \oplus s_i^j$ with a common mask $\alpha_t$. Since $\alpha_t$ is uniform and independent, and each $s_i^j$ is either a fresh uniform label $r_q$ or random value, the joint distribution of $\{\alpha_{h_j(w_i^*)} \oplus s_i^j\}_{h_j(w_i^*)=t}$ is indistinguishable from independent uniform values. Consequently, the entire $\{\alpha_{h_j(w_i^*)} \oplus s_i^j\}_{i\in[n],j\in[\gamma]}$ is indistinguishable from uniform and does not reveal information about $\mathsf{Y}$.

$\mathsf{Hybrid}_3$: Same as $\mathsf{Hybrid}_2$, except that the output of $\Pi_{\text{pECRG}}$ is replaced by $\{c_i\}_{i\in[\beta]}$ chosen by *Sim*, and runs the $\Pi_{\text{pECRG}}$ simulator to produce the simulated view for $\mathscr{S}$. The security of protocol $\Pi_{\text{pECRG}}$ guarantees that the view in simulation is computationally indistinguishable from the view in $\mathsf{Hybrid}_2$. The hybrid is the view output by *Sim*.

$\mathsf{Hybrid}_4$: Same as $\mathsf{Hybrid}_3$ except that the output of $\Pi_{\text{nECRG}}$ is replaced by $\{u_i\}_{i\in[\beta]}$ chosen by *Sim*, and runs the $\Pi_{\text{nECRG}}$ simulator to produce the simulated view for $\mathscr{S}$. The security of protocol $\Pi_{\text{nECRG}}$ guarantees that the view in simulation is computationally indistinguishable from the view in $\mathsf{Hybrid}_3$.

*Corrupted receiver: Sim* simulates a real execution in which the receiver is corrupted. Since $\mathscr{A}$ is semi-honest, *Sim* can obtain the input $\mathsf{Y}$ of the receiver directly, externally send $\mathsf{Y}$ to $\mathscr{F}_{\text{ePSU}}^{n_1,n_2}$ and then receive $\langle \mathsf{Request}, \mathscr{R} \rangle$. Upon receiving $\mathsf{Y}$ from $\mathscr{A}$, *Sim* randomly selects $\{w_i^*\}_{i\in[n]}$ and simulates the execution of $\Pi_{\text{OPRF}}$. *Sim* computes a random OKVS $\mathsf{D}$ by selecting random key-value pairs. When receiving $\{\alpha_i\}_{i\in[\beta]}$, the input of $\Pi_{\text{pECRG}}$, from $\mathscr{A}$, *Sim* randomly selects $\{c_i^*\}_{i\in[\beta]}$ and simulates the execution of $\Pi_{\text{pECRG}}$. After receiving $\{c_i^*\}_{i\in[\beta]}$, the input of $\Pi_{\text{nECRG}}$, from $\mathscr{A}$, *Sim* randomly selects $\{v_i\}_{i\in[\beta]}$

ARTICLE IN PRESS

and simulates the execution of $\Pi_{\text{nECRG}}$. *Sim* sends $\langle \mathsf{Request}, \mathsf{OK} \rangle$ to $\mathscr{F}_{\text{ePSU}}^{n_1,n_2}$. After receiving $\mathsf{Z} = \mathsf{X} \cup \mathsf{Y}$ from $\mathscr{F}_{\text{ePSU}}^{n_1,n_2}$, *Sim* computes $\tilde{\mathsf{X}} = \mathsf{X} \backslash \mathsf{Y} = \mathsf{Z} \backslash \mathsf{Y}$ and randomly selects a subset $\tilde{V}$ of $\{v_i\}_{i\in[\beta]}$ where $|\tilde{V}| = |\tilde{\mathsf{X}}|$. For each item $\tilde{v}_i \in \tilde{V}$, *Sim* sets $ct_i = \tilde{v}_i \oplus (\tilde{x}_i || \mathsf{H}(\tilde{x}_i))$. Finally, *Sim* chooses random values for $i \in \{|\tilde{\mathsf{X}}|+1, \ldots, \beta\}$ and sends all shuffled ciphertexts to $\mathscr{A}$.

We argue that the outputs of *Sim* are indistinguishable from the real view of $\mathscr{R}$ by the following hybrids:

$\mathsf{Hybrid}_0$: The view of the receiver in the real protocol.

$\mathsf{Hybrid}_1$: Same as $\mathsf{Hybrid}_0$ except that the output of $\Pi_{\text{OPRF}}$ is replaced by $\{w_i^*\}_{i\in[\beta]}$ chosen by *Sim*, and runs the $\Pi_{\text{OPRF}}$ simulator to produce the simulated view for $\mathscr{R}$. The security of protocol $\Pi_{\text{OPRF}}$ guarantees that the view in simulation is computationally indistinguishable from the view in $\mathsf{Hybrid}_0$.

$\mathsf{Hybrid}_2$: Same as $\mathsf{Hybrid}_1$ except for computing a random OKVS D by *Sim*. Briefly, this simulation is indistinguishable for the following reasons: The randomness of $\{r_i\}_{i\in[\beta]}$ is indistinguishable from random, and the obliviousness and randomness of OKVS guarantee that the D is distributed uniformly.

$\mathsf{Hybrid}_3$: Same as $\mathsf{Hybrid}_2$ except that the output of $\Pi_{\text{pECRG}}$ is replaced by $\{c_i\}_{i\in[\beta]}$ chosen by *Sim*, and runs the $\Pi_{\text{pECRG}}$ simulator to produce the simulated view for $\mathscr{R}$. The security of protocol $\Pi_{\text{pECRG}}$ guarantees that the view in simulation is computationally indistinguishable from the view in $\mathsf{Hybrid}_2$.

$\mathsf{Hybrid}_4$: Same as $\mathsf{Hybrid}_3$ except that the output of $\Pi_{\text{nECRG}}$ is replaced by $\{v_i\}_{i\in[\beta]}$ chosen by *Sim*, and runs the $\Pi_{\text{nECRG}}$ simulator to produce the simulated view for $\mathscr{R}$. The corresponding $\{ct_i\}_{i\in[\beta]}$ are also changed according to $\{v_1, \ldots, v_\beta\}$. The security of protocol $\Pi_{\text{nECRG}}$ guarantees that the view in simulation is computationally indistinguishable from the view in $\mathsf{Hybrid}_3$. The hybrid is the view output by *Sim*. □

**Complexity.** The theoretical complexities of our $\Pi_{\text{ePSU}}$ is shown in Table 2.

| | $\Pi_{\text{OPRF}}$ | Bi. OKVS | $\Pi_{\text{nECRG}}$ | $\Pi_{\text{pECRG}}$ | Ciphertexts |
|---|---|---|---|---|---|
| Comp. | $O(n)$ | $O(\gamma \cdot n)$ | $O(\varepsilon \cdot n)$ | $O(\varepsilon \cdot n)$ | $O(\varepsilon \cdot n)$ |
| Comm. | $O(n)$ | $O(\gamma \cdot n)$ | $O(\varepsilon \cdot n)$ | $O(\varepsilon \cdot n)$ | $O(\varepsilon \cdot n)$ |

**Table 2.** The theoretical complexities of $\Pi_{\text{ePSU}}$. "Bi. OKVS" means "Bidirectional OKVS".

ARTICLE IN PRESS

## Implementation and Performance

Our evaluation proceeds in three stages.

- First, we benchmark the core components in the framework of Tu et al.[16], including the hash-structure construction and the MCRG module, and compare them against our optimized counterparts.
- Second, under the revised framework that eliminates hash-based leakage via introducing an OPRF, we further evaluate the end-to-end cost of OPRF + hash-structure construction + MCRG and compare it with our improved design under the same security setting.
- Finally, we conduct an end-to-end evaluation of our ePSU protocol. Specifically, we measure both (i) a variant without OPRF (which is vulnerable to hash-based leakage) and (ii) the leakage-resilient variant with OPRF, and compare them with state-of-the-art baselines, including PSU protocols with during-execution leakage[13,14] and the ePSU protocol of[16]. In addition, we implement and evaluate our revised version of[16] to ensure a fair comparison under aligned security goals.

**Benchmarking Environment.** We implement our protocols in C++ on a MacBook Pro running a macOS 14.0, equipped with an Apple M3 Pro chip and 36GB RAM. We evaluate the protocols in two network settings: LAN (10Gbps bandwidth, 0.2ms RTT) and WAN (100Mbps, 10Mbps bandwidth, 80ms RTT).

**Implementation Details.** Following most PSU[12–16], We set the computational security parameter $\kappa = 128$, the statistical security parameter $\lambda = 40$, and use $\gamma = 3$ hash functions to insert set $\mathsf{X}$ into the cuckoo hash table with bin size $\beta = \varepsilon \cdot n = 1.27n$ (stash=0) in our setting. We set each item in the set is represented as a 128-bit value. We implement OPRF following the construction in[17] (code: `https://github.com/encryptogroup/circuitPSU.git`), OKVS based on[27] (code: `https://github.com/google/private-membership.git`) and nECRG/pECRG based on[16] (code: `https://zenodo.org/records/14725816`). Additionally, we use the cryptoTools library (code: `https://github.com/ladnir/cryptoTools`) and coproto library (code: `https://github.com/Visa-Research/coproto.git`) as the general framework to compute the hash functions, PRNG calls, creating channels, and so on. Moreover, since our protocol is an improvement over the construction of[16], our implementation is built upon the codebase (`https://zenodo.org/records/14725816`) released by[16].

ARTICLE IN PRESS

**Impact of item length.** Our implementation represents each element as an `osuCrypto::block` (128-bit). For longer real-world identifiers, there are two standard ways to handle item length.

*(Fixed-length encoding).* One may first map an arbitrary-length identifier $id$ to a fixed-length representation using a collision-resistant hash (e.g., $\mathsf{enc}(id) = H(id) \in \{0,1\}^{\ell'}$ with $\ell' \geq \lambda + 2\log n$), and then run the protocol over these encodings. In this common setting, the dominant costs of the protocol (OKVS/OPRF/OT-based sub-protocols) remain essentially unchanged; only a local hashing cost is added.

*(Multi-block representation).* Alternatively, if one wishes to operate on the raw values directly, an $L$-bit element can be represented using $b = \lceil L/128 \rceil$ blocks. In this case, both communication and computation scale approximately linearly with $b$ for the components that process elements block-wise, implying an $\approx 2\times$ overhead when moving from 128-bit to 256-bit items (and $\approx 4\times$ for 512-bit). Minor deviations from exact linearity may occur due to fixed protocol overheads independent of $L$.

## Performance Evaluation

### *Evaluation on Hash-structure construction + MCRG.*

As described in our protocol construction, our work optimizes the ePSU protocol of Tu et al., with the main improvements concentrated in the hash-structure construction + MCRG component. Table 3 reports the performance of this component in[16] and in our optimized design. The results show that our approach substantially improves efficiency. When the set sizes range from $|X| = |Y| = 2^{10}$ to $2^{24}$, our protocol achieves a 1.413–17.431× reduction in communication compared to[16], and a 1.375–18.444× runtime speedup across different network bandwidth settings.

| Size($\|X\| = \|Y\|$) | Protocols | Comm.(MB) | Total running time (s) with single thread | | |
|---|---|---|---|---|---|
| | | | 10 Gbps | 100 Mbps | 10 Mbps |
| $2^{10}$ | Tu et al.[16] hash+MCRG | 1.255 | 0.166 | 2.299 | 3.868 |
| | Our improved hash+MCRG | 0.072 | 0.009 | 0.194 | 0.272 |
| $2^{12}$ | Tu et al.[16] hash+MCRG | 1.617 | 0.278 | 3.073 | 4.278 |
| | Our improved hash+MCRG | 0.279 | 0.036 | 0.249 | 0.746 |
| $2^{14}$ | Tu et al.[16] hash+MCRG | 2.854 | 0.450 | 4.315 | 6.629 |
| | Our improved hash+MCRG | 1.104 | 0.098 | 0.838 | 1.617 |
| $2^{16}$ | Tu et al.[16] hash+MCRG | 7.689 | 1.230 | 6.978 | 11.940 |
| | Our improved hash+MCRG | 4.493 | 0.394 | 2.168 | 5.867 |
| $2^{18}$ | Tu et al.[16] hash+MCRG | 26.796 | 3.792 | 16.524 | 34.420 |
| | Our improved hash+MCRG | 18.016 | 1.638 | 4.942 | 21.596 |
| $2^{20}$ | Tu et al.[16] hash+MCRG | 103.082 | 13.379 | 34.610 | 117.461 |
| | Our improved hash+MCRG | 72.123 | 6.771 | 17.970 | 78.523 |
| $2^{22}$ | Tu et al.[16] hash+MCRG | 408.392 | 55.218 | 124.048 | 453.130 |
| | Our improved hash+MCRG | 288.623 | 26.767 | 63.440 | 329.635 |
| $2^{24}$ | Tu et al.[16] hash+MCRG | 1631.097 | 225.649 | 454.554 | 2230.786 |
| | Our improved hash+MCRG | 1154.553 | 129.539 | 236.405 | 981.384 |

**Table 3.** Comparisons of communication and runtime between our optimized hash-structure construction + MCRG and that of[16] in single thread. 10Gbps bandwidth, 0.2ms RTT; 100Mbps and 10Mbps bandwidth, 80ms RTT; The best results are marked in pink.

### *Evaluation on OPRF + hash-structure construction + MCRG.*

To eliminate hash-based leakage, we incorporate an OPRF into the framework of Tu et al., which yields a revised baseline with the desired leakage-resilience, and we further optimize the resulting construction. Enabling leakage-resilience via introducing OPRF and improving the efficiency of the hash-structure construction + MCRG component constitute the main contributions of this work. Table 4 reports the performance of OPRF + hash-structure construction + MCRG in the revised baseline of[16] and in our optimized design.

The results show that our approach provides substantial efficiency gains. When the set sizes range from $|X| = |Y| = 2^{10}$ to $2^{24}$, our approach achieves a 1.197–6.975× reduction in communication compared to the revised baseline, and a 1.187–5.618× runtime speedup across different network bandwidth settings.

Compared with the hash-structure construction + MCRG evaluation (Table 3), the relative performance gap becomes smaller in the OPRF-augmented setting (Table 4). This is expected, since introducing OPRF adds a common overhead to both the revised baseline and our optimized design. In other words, the overall cost in the leakage-resilient variant can be decomposed into (i) an OPRF-dependent part and (ii) the remaining pipeline dominated by hash-structure construction and MCRG, where our optimizations apply. When the OPRF-dependent part constitutes a non-trivial fraction of the end-to-end

ARTICLE IN PRESS

runtime and communication, improvements in the non-OPRF part contribute a smaller fraction of the total, thereby reducing the observed relative speedups and communication reductions.

This observation also implies that adopting a more efficient OPRF instantiation could further amplify the benefit of our optimized design: as the OPRF overhead decreases, the end-to-end cost becomes increasingly dominated by the hash-structure construction and MCRG components, where our bidirectional-OKVS optimization yields substantial savings. We note that OPRF constructions (e.g., circuit-based OPRF[24]) report improved efficiency under comparable security settings. For fairness and reproducibility, we keep the OPRF implementation consistent with the released codebases of our baselines. In particular, the public PSU implementation of Chandran et al. adopts the same OPRF instantiation as ours. Consequently, replacing it with a different OPRF construction would change the baseline performance and hinder an apples-to-apples comparison.

| Size($\|X\| = \|Y\|$) | Protocols | Comm.(MB) | Total running time (s) with single thread | | |
|---|---|---|---|---|---|
| | | | 10 Gbps | 100 Mbps | 10 Mbps |
| $2^{10}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 1.381 | 0.191 | 2.985 | 4.445 |
| | Our improved OPRF+hash+MCRG | 0.198 | 0.034 | 0.905 | 1.567 |
| $2^{12}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 1.963 | 0.325 | 3.965 | 5.838 |
| | Our improved OPRF+hash+MCRG | 0.625 | 0.083 | 1.141 | 2.306 |
| $2^{14}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 4.086 | 0.577 | 5.466 | 9.563 |
| | Our improved OPRF+hash+MCRG | 2.336 | 0.225 | 1.989 | 4.551 |
| $2^{16}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 12.501 | 1.652 | 9.013 | 20.820 |
| | Our improved OPRF+hash+MCRG | 9.305 | 0.816 | 4.203 | 14.747 |
| $2^{18}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 46.069 | 5.891 | 22.120 | 57.617 |
| | Our improved OPRF+hash+MCRG | 37.289 | 3.737 | 10.538 | 44.793 |
| $2^{20}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 180.762 | 21.999 | 54.607 | 196.944 |
| | Our improved OPRF+hash+MCRG | 149.803 | 15.289 | 37.865 | 157.995 |
| $2^{22}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 721.947 | 91.168 | 193.605 | 786.892 |
| | Our improved OPRF+hash+MCRG | 602.178 | 62.094 | 132.366 | 662.756 |
| $2^{24}$ | Revised Tu et al.[16] OPRF+hash+MCRG | 2897.152 | 369.357 | 696.914 | 3229.653 |
| | Our improved OPRF+hash+MCRG | 2420.608 | 272.923 | 478.434 | 1979.911 |

**Table 4.** Comparisons of communication and runtime between our optimized OPRF + hash-structure construction + MCRG and the revised baseline of[16] in single thread. 10Gbps bandwidth, 0.2ms RTT; 100Mbps and 10Mbps bandwidth, 80ms RTT; The best results are marked in pink.

ARTICLE IN PRESS

***Evaluation on PSU.***

Table 5 and Figure 8 report the end-to-end communication and runtime of our ePSU against state-of-the-art baselines, including PSU protocols with during-execution leakage (Zhang et al.[13] and Chandran et al.[14]) and the ePSU protocol of Tu et al.[16]. We also evaluate (i) a revised implementation of Tu et al.'s framework in which hash-based leakage is mitigated, and (ii) our ePSU* variant that still exhibits hash-based leakage, obtained by replacing Tu et al.'s hash-structure construction + MCRG component with our optimized counterpart while keeping the rest of the framework unchanged. Additionally, we ran the public implementation of the PSU protocol released by Chandran et al. However, their implementation repeatedly crashed with a segmentation fault when $|X| = |Y| \geq 2^{18}$. Therefore, we report and compare their results only for set sizes $|X| = |Y| \in \{2^{10}, 2^{12}, 2^{14}, 2^{16}\}$.

**Communication analysis.** Table 5 and Figure 8 (the first panel) report the communication costs of the evaluated protocols and their growth trends. We analyze the communication results along two axes: (i) efficiency-oriented PSU baselines that incurs during-execution leakage, and (ii) leakage-resilient ePSU baselines that aim to prevent both during-execution leakage and hash-based leakage.

- *PSU baselines with during-execution leakage.* Zhang et al. and Chandran et al. incur the lowest communication among all evaluated schemes. For instance, at $|X| = |Y| = 2^{20}$, Zhang et al. communicates 179.061 MB, whereas our leakage-resilient ePSU communicates 325.453 MB. Overall, compared to Zhang et al., our ePSU incurs a 1.797–2.612× higher communication for set sizes ranging from $2^{10}$ to $2^{24}$; compared to Chandran et al., it incurs a 1.299–2.097× higher communication for set sizes ranging from $2^{10}$ to $2^{16}$. This gap is expected, as ePSU additionally pays for during-execution leakage-mitigation components that are absent from standard PSU.

- *Original vs. revised Tu et al.* To quantify the communication overhead introduced purely by mitigating hash-based leakage, we evaluate Tu et al.'s original ePSU and its revised variant, as well as our leakage-resilient ePSU and its counterpart Our ePSU* that still exhibits hash-based leakage. We test balanced set sizes ranging from $2^{10}$ to $2^{24}$. The results show that revised Tu et al.'s framework increases communication by 1.081–1.279× compared to their original

ARTICLE IN PRESS

| Size($\|X\| = \|Y\|$) | Protocols | Comm.(MB) | Total running time (s) with single thread | | |
|---|---|---|---|---|---|
| | | | 10 Gbps | 100 Mbps | 10 Mbps |
| $2^{10}$ | Zhang et al.[13] PSU | 0.188 | 2.314 | 3.489 | 4.477 |
| | Chandran et al.[14] PSU | 0.378 | 0.359 | 5.216 | 5.728 |
| | Tu et al.[16]* ePSU | 1.548 | 0.343 | 4.573 | 7.018 |
| | Our ePSU* | 0.365 | 0.186 | 2.468 | 3.422 |
| | Revised Tu et al.[16] ePSU | 1.674 | 0.368 | 5.259 | 7.595 |
| | Our ePSU | 0.491 | 0.211 | 3.179 | 4.717 |
| $2^{12}$ | Zhang et al.[13] PSU | 0.700 | 2.731 | 3.799 | 5.182 |
| | Chandran et al.[14] PSU | 0.798 | 0.501 | 5.931 | 6.433 |
| | Tu et al.[16]* ePSU | 2.346 | 0.796 | 5.882 | 9.141 |
| | Our ePSU* | 1.008 | 0.554 | 3.058 | 5.609 |
| | Revised Tu et al.[16] ePSU | 2.692 | 0.843 | 6.774 | 10.701 |
| | Our ePSU | 1.354 | 0.601 | 3.950 | 7.169 |
| $2^{14}$ | Zhang et al.[13] PSU | 2.726 | 5.255 | 6.760 | 8.995 |
| | Chandran et al.[14] PSU | 2.507 | 1.059 | 6.947 | 9.309 |
| | Tu et al.[16]* ePSU | 5.570 | 2.441 | 9.628 | 14.676 |
| | Our ePSU* | 3.820 | 2.089 | 6.151 | 9.664 |
| | Revised Tu et al.[16] ePSU | 6.802 | 2.568 | 10.779 | 17.610 |
| | Our ePSU | 5.052 | 2.216 | 7.302 | 12.598 |
| $2^{16}$ | Zhang et al.[13] PSU | 11.143 | 15.921 | 18.772 | 22.804 |
| | Chandran et al.[14] PSU | 9.499 | 3.844 | 10.417 | 18.326 |
| | Tu et al.[16]* ePSU | 18.306 | 9.105 | 21.670 | 33.216 |
| | Our ePSU* | 15.110 | 8.269 | 16.860 | 27.143 |
| | Revised Tu et al.[16] ePSU | 23.118 | 9.527 | 23.705 | 42.096 |
| | Our ePSU | 19.922 | 8.691 | 18.895 | 36.023 |
| $2^{18}$ | Zhang et al.[13] PSU | 44.712 | 50.930 | 57.447 | 73.581 |
| | Chandran et al.[14] PSU | – | – | – | – |
| | Tu et al.[16]* ePSU | 69.872 | 36.100 | 60.083 | 105.410 |
| | Our ePSU* | 61.092 | 33.946 | 48.501 | 92.586 |
| | Revised Tu et al.[16] ePSU | 89.145 | 38.199 | 65.679 | 128.607 |
| | Our ePSU | 80.365 | 36.045 | 54.097 | 115.783 |
| $2^{20}$ | Zhang et al.[13] PSU | 179.061 | 183.825 | 205.714 | 272.943 |
| | Chandran et al.[14] PSU | – | – | – | – |
| | Tu et al.[16]* ePSU | 278.732 | 156.835 | 203.108 | 401.533 |
| | Our ePSU* | 247.773 | 150.227 | 186.468 | 362.595 |
| | Revised Tu et al.[16] ePSU | 356.412 | 165.455 | 223.105 | 481.016 |
| | Our ePSU | 325.453 | 158.745 | 206.363 | 442.067 |
| $2^{22}$ | Zhang et al.[13] PSU | 716.875 | 768.948 | 829.887 | 3814.863 |
| | Chandran et al.[14] PSU | – | – | – | – |
| | Tu et al.[16]* ePSU | 1123.657 | 596.138 | 794.473 | 5578.385 |
| | Our ePSU* | 1003.888 | 567.687 | 733.865 | 5454.890 |
| | Revised Tu et al.[16] ePSU | 1437.212 | 632.088 | 864.030 | 5912.147 |
| | Our ePSU | 1317.443 | 603.014 | 802.791 | 5788.011 |
| $2^{24}$ | Zhang et al.[13] PSU | 2869.866 | 2951.187 | 3182.994 | 18566.191 |
| | Chandran et al.[14] PSU | – | – | – | – |
| | Tu et al.[16]* ePSU | 4545.158 | 2690.400 | 3047.165 | 27148.909 |
| | Our ePSU* | 4068.614 | 2594.290 | 2829.016 | 25899.504 |
| | Revised Tu et al.[16] ePSU | 5811.213 | 2834.108 | 3289.525 | 28147.776 |
| | Our ePSU | 5334.669 | 2737.674 | 3071.045 | 26898.031 |

**Table 5.** Comparisons of communication and runtime between our ePSU and (PSU with during-execution leakage[13,14], ePSU[16], revised ePSU in[16]) in single thread. 10Gbps bandwidth, 0.2ms RTT; 100Mbps and 10Mbps bandwidth, 80ms RTT; The best and the second results are marked in pink and blue, respectively. The best results among protocols that resist both during-execution leakage and hash-based leakage are marked in green. "*" denotes the privacy leakage caused by directly applying hash-to-bin on sets. "–" indicates that the implementation crashed with a segmentation fault during the evaluation. Our ePSU* denotes our implementation without OPRF, and is vulnerable to hash-based leakage. Our ePSU denotes the hash-based leakage-resilient variant with OPRF.

framework, while Our ePSU incurs a 1.311–1.345× increase compared to Our ePSU*. In both cases, the additional communication is primarily due to the extra OPRF invocation. The slightly larger multiplicative increase in our case is mainly because our bidirectional-OKVS optimization reduces the cost of the remaining pipeline, making the fixed OPRF overhead account for a larger fraction of the total communication.

ARTICLE IN PRESS

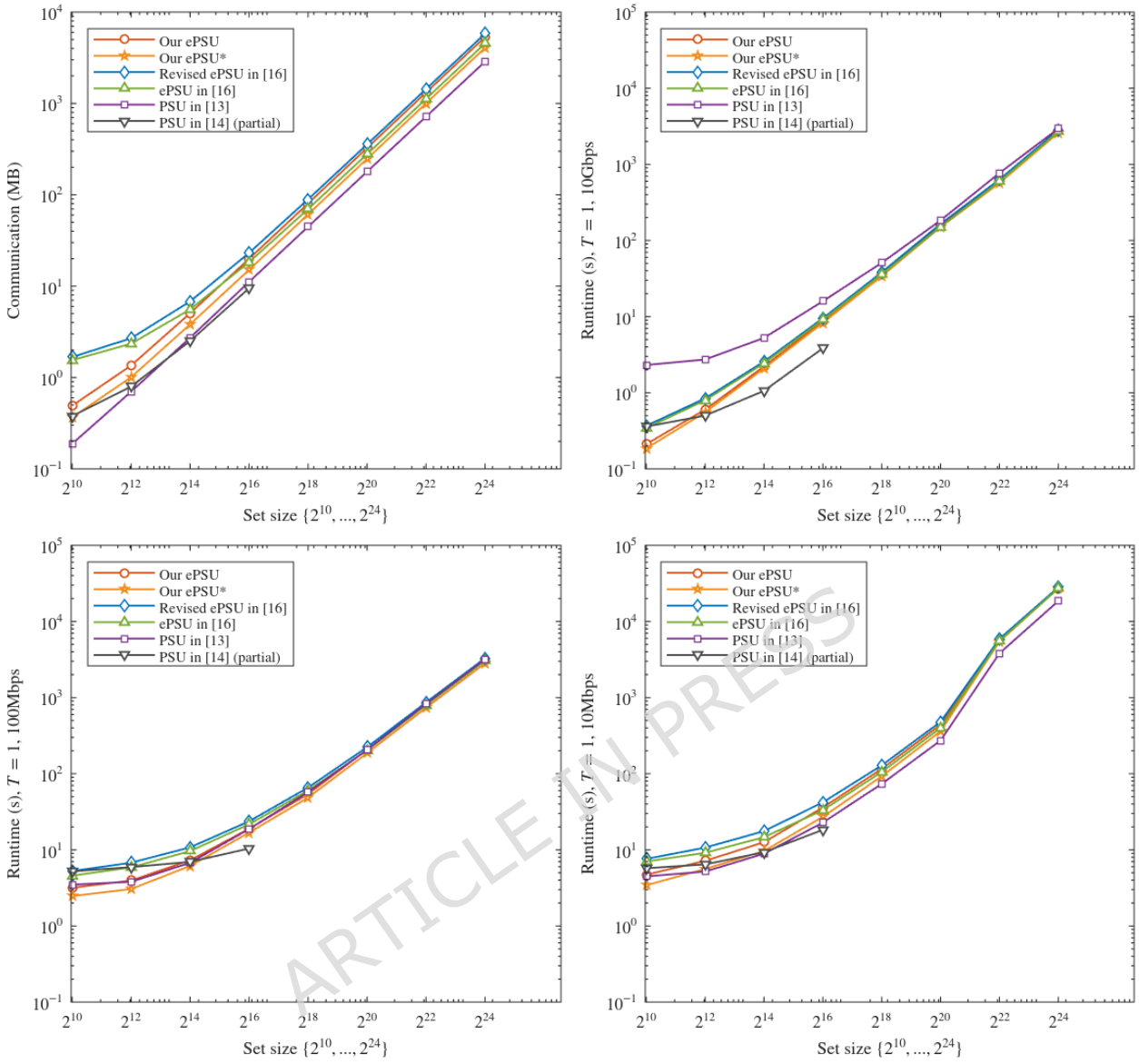



**Figure 8.** Comparisons of communication and runtime between our ePSU and (PSU with during-execution leakage[13,14], ePSU[16], revised ePSU[16]). Both $x$ and $y$-axis are in log scale. The first figure shows the communication cost increases as the set size increases. The last three figures show the runtime increases as the set size increases in different bandwidth.

- *Effect of our bidirectional-OKVS optimization.* Our ePSU$^*$ replaces Tu et al.'s hash-structure construction + MCRG component with our optimized module while keeping the rest of the framework unchanged. This isolates the benefit of our bidirectional-OKVS optimization: compared to Tu et al.'s original ePSU, our ePSU$^*$ achieves a 1.117–4.241$\times$ reduction in communication for set sizes ranging from $2^{10}$ to $2^{24}$. In addition, compared with Tu et al.'s original protocol, our ePSU has a clear communication advantage for smaller sizes: for set sizes ranging from $2^{10}$ to $2^{14}$, it achieves a 1.103–3.153$\times$ reduction in communication. As the set size grows, Tu et al.'s original protocol gradually becomes more communication efficient, and for set sizes ranging from $2^{16}$ to $2^{24}$ our ePSU incurs a 1.088–1.174$\times$ communication overhead. This trend is consistent with our component-level analysis in the OPRF + hash-structure construction + MCRG evaluation: the additional OPRF cost, which is required to eliminate hash-based leakage, becomes increasingly visible at larger scales. Adopting a more efficient OPRF implementation would reduce this gap and could potentially make our protocol superior even in the large-scale regime. Finally, under aligned leakage-resilience goals, our ePSU consistently improves over the revised Tu et al. baseline. Specifically, compared to the revised Tu et al. protocol, our ePSU achieves a

ARTICLE IN PRESS

1.089–3.409× reduction in communication.

**Runtime analysis under different bandwidths.** Table 5 and Figure 8 (last three panels) report the single-thread runtime of the evaluated protocols under different network bandwidth settings. We first emphasize that, among the leakage-resilient protocols (i.e., our ePSU and the revised Tu et al. baseline), our construction consistently achieves the best runtime. Across bandwidth settings and for set sizes ranging from $2^{10}$ to $2^{24}$, our ePSU obtains a 1.021–1.744× runtime speedup over the revised Tu et al. protocol. Following, we provide a comprehensive comparison of the evaluated protocols under each bandwidth setting.

**LAN (10 Gbps, RTT 0.2 ms).** Under high bandwidth and low latency, runtime is dominated by computation and the core pipeline cost. At the smallest tested size $2^{10}$, both of our ePSU* and ePSU are faster than all other evaluated protocols. For medium sizes ($2^{12}$ to $2^{16}$), our ePSU* and ePSU are slightly slower than the PSU implementation of Chandran et al., incurring a 1.106–2.151× and 1.200–2.261× runtime overhead, respectively. For set sizes ranging from $2^{10}$ to $2^{24}$,, our ePSU* consistently achieves the best runtime among all protocols we can evaluate end-to-end: it yields a 1.224–12.441× speedup over the PSU protocol of Zhang et al., a 1.037–1.844× speedup over Tu et al.'s original ePSU, and a 1.092–1.978× speedup over the revised Tu et al. baseline. Our leakage-resilient ePSU also provides substantial speedups, achieving a 1.078–10.967× improvement over Zhang et al. and a 1.035–1.744× improvement over the revised Tu et al. baseline. Compared with Tu et al.'s original ePSU, our ePSU is faster for set sizes ranging from $2^{10}$ to $2^{18}$, with a 1.002–1.626× speedup. For larger sizes ($2^{20}$ to $2^{24}$), it becomes slightly slower (1.012–1.018× overhead), which is attributable to the additional OPRF cost required to eliminate hash-based leakage.

**WAN (100 Mbps, RTT 80 ms).** When bandwidth and latency are constrained, runtime increasingly reflects both data transfer and round-trip latency. Compared with the PSU protocol of Chandran et al., our ePSU* is faster for set sizes ranging from $2^{10}$ to $2^{14}$, achieving a 1.129–2.113× speedup, but becomes slower at $2^{16}$ with a 1.619× runtime overhead. Our ePSU achieves a 1.502–1.641× speedup over Chandran et al. for $2^{10}$–$2^{12}$, while incurring a 1.051–1.814× overhead for $2^{14}$–$2^{16}$. Compared with other baselines, our ePSU* consistently attains the best runtime among the evaluated protocols in this setting, yielding a 1.099–1.414× speedup over Zhang et al.'s PSU, a 1.077–1.923× speedup over Tu et al.'s original ePSU, and a 1.163–2.215× speedup over the revised Tu et al. baseline. Our ePSU achieves runtime comparable to Zhang et al.'s PSU, and it provides a 1.111–1.439× speedup over Tu et al.'s ePSU for set sizes ranging from $2^{10}$ to $2^{18}$. For larger sizes ($2^{20}$–$2^{24}$), it incurs a small overhead of 1.008–1.016× relative to Tu et al.'s ePSU, which is again attributable to the additional OPRF cost required to eliminate hash-based leakage. Under aligned leakage-resilience goals, our ePSU consistently outperforms the revised Tu et al. baseline, achieving a 1.071–1.654× runtime speedup for set sizes ranging from $2^{10}$ to $2^{24}$.

**WAN (10 Mbps, RTT 80 ms).** At 10 Mbps, runtime is overwhelmingly dominated by the total communication volume. Accordingly, Zhang et al.'s PSU consistently outperforms both our ePSU* and ePSU, achieving a 1.082–1.395× and 1.383–1.620× speedup, respectively. This is mainly because Zhang et al.'s protocol consists of only two bulk transmissions—one sending an OKVS encoding of $n$ key–value pairs and the other sending $n$ values—together with $n$ instances of 1-out-of-2 OT, whereas our construction invokes multiple OT-based sub-protocols and therefore transmits more data, making it more sensitive to bandwidth in this regime. Additionally, Compared with the PSU protocol of Chandran et al., our ePSU* is faster for set sizes ranging from $2^{10}$ to $2^{12}$, achieving a 1.147–1.674× speedup, but becomes slower for $2^{14}$–$2^{16}$ with a 1.038–1.481× overhead. Our ePSU achieves a 1.214× speedup at $2^{10}$, while incurring a 1.114–1.966× overhead for $2^{12}$–$2^{16}$. Compared with ePSU baselines, our ePSU* consistently improves over Tu et al.'s original ePSU by 1.023–2.051× and over the revised Tu et al. baseline by 1.084–2.219×. Our ePSU achieves runtime comparable to Tu et al.'s original ePSU, and it consistently outperforms the revised Tu et al. baseline, obtaining a 1.021–1.610× speedup for set sizes ranging from $2^{10}$ to $2^{24}$.

## Practical composition considerations

Our analysis is in the hybrid model and assumes idealized sub-functionalities. When instantiated with concrete sub-protocols, care is needed to avoid implementation-level leakage: protocol control flow and transcript lengths should be input-independent (or padded), cryptographic kernels should be constant-time, and aborts should be treated as DoS events rather than information-bearing signals. These are standard assumptions in practical secure computation deployments and are orthogonal to the protocol-level leakage mechanisms studied in this paper.

## Conclusion

In this work, we revisit the state-of-the-art enhanced private set union framework proposed by Tu et al. (USENIX Security 2025) and identify a subtle but critical hash-based leakage in their efficient construction. To address this problem, we propose a leakage-resilient ePSU framework that combines oblivious pseudorandom function with randomized shuffling, effectively eliminating the privacy risks introduced by directly applying hash-to-bin. Building upon the revised framework, we further

ARTICLE IN PRESS

present an optimized ePSU protocol based on a bidirectional oblivious key-value store, which significantly improves both communication and computational efficiency while preserving strong security guarantees. We formally prove the security of our construction in the semi-honest model and provide a complexity analysis. Extensive experimental results demonstrate that, compared to the revised baseline, our protocol consistently achieves substantial improvements in both communication cost and runtime. Our protocol is restricted to the semi-honest model. While this is the standard starting point for efficient PSU constructions and is consistent with the security model of the most directly comparable baselines, extending leakage resilience to the malicious setting would require additional consistency checks and/or zero-knowledge techniques, which may introduce non-trivial overhead. An interesting direction for future work is to extend our framework to stronger adversarial models, such as the malicious setting, further optimize performance under diverse network and system conditions, and generalize our approach to support a broader class of privacy-preserving set operations and secure multi-party computation tasks.

## Data availability

The datasets used and analyzed during the present study are available from the corresponding author upon reasonable request.

## References


**1.** Lenstra, A. & Voss, T. Information Security Risk Assessment, Aggregation, and Mitigation. *ACISP'04*, 391-401 (2004).

**2.** Hogan, K. et al. Secure Multiparty Computation for Cooperative Cyber Risk Assessment. *SecDev'16*, 75-76 (2016).

**3.** Chung, H., Kim, M. & Yang, P.C. Multiparty Delegated Private Set Union With Efficient Updates on Outsourced Datasets. *TDSC*, 903-918 (2025).

**4.** Burkhart, M., Strasser, M., Many, D. & Dimitropoulos, X. SEPIA: Privacy-Preserving Aggregation of Multi-Domain Network Events and Statistics. *USENIX Security'10* (2010).

**5.** Freudiger, J., Cristofaro, E.D. & Brito, A.E. Controlled Data Sharing for Collaborative Predictive Blacklisting. *DIMVA'15*, 327-349 (2015).

**6.** Ramanathan, S., Mirkovic, J. & Yu, M. BLAG: improving the accuracy of blacklists. *NDSS'20* (2020).

**7.** Buddhavarapu, P. et al. Private Matching for Compute. Preprint at https://eprint.iacr.org/2020/599 (2020).

**8.** Garimella, G. et al. Private Set Operations from Oblivious Switching. *PKC'21*, 591-617 (2021).

**9.** Yan, J., Wei, L., Qian, X. & Zhang, L. IDPriU: A two-party ID-private data union protocol for privacy-preserving machine learning. *J. Inf. Secur. Appl* (2025).

**10.** Kolesnikov, V., Rosulek, M., Trieu, N. & Wang, X. Scalable Private Set Union from Symmetric-Key Techniques. *ASIACRYPT'19*, 636-666 (2019).

**11.** Rabin, M.O. How to exchange secrets with oblivious transfer. Preprint at https://eprint.iacr.org/2005/187.pdf (2005).

**12.** Jia, Y. et al. Shuffle-based Private Set Union: Faster and More Secure. *USENIX Security'22*, 2947-2964 (2022).

**13.** Zhang, C. et al. Linear Private Set Union from Multi-Query Reverse Private Membership Test. *USENIX Security'23*, 337-354 (2023).

**14.** Chandran, G.R., Schneider, T., Stillger, M., & Weinert, C. Concretely Efficient Private Set Union via Circuit-Based PSI. *ASIACCS'25*, 149-162 (2025).

**15.** Jia, Y., Sun, S., Zhou, H & Gu, D. Scalable Private Set Union, with Stronger Security. *USENIX Security'24*, 6471-6488 (2024).

**16.** Tu, B. et al. Fast Enhanced Private Set Union in the Balanced and Unbalanced Scenarios. *USENIX Security'25*, 3437-2456 (2025).

**17.** Chase, M. & Miao, P. Private Set Intersection in the Internet Setting from Lightweight Oblivious PRF. *CRYPTO'20*, 34-63 (2020).

**18.** Raghuraman, S & Rindal, P. Blazing Fast PSI from Improved OKVS and Subfield VOLE. *CCS'22*, 2505-2517 (2022).

**19.** Tu, B., Chen, Y., Liu, Q., & Zhang, C. Fast Unbalanced Private Set Union from Fully Homomorphic Encryption. *ACM CCS'23*, 2959-2973 (2023).

**20.** Zhang, C. et al. Unbalanced Private Set Union with Reduced Computation and Communication. *ACM CCS'24*, 1434-1477 (2024).

**21.** Dumas, J.G. et al. Optimal Communication Unbalanced Private Set Union. *ACNS'25*, 107-135 (2025).

**22.** Mohassel, P. & Sadeghian, S.S. How to hide circuits in MPC an efficient framework for private function evaluation. *EUROCRYPT'13*, 557-574 (2013).

**23.** Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M. & Trieu, N. Practical multi-party private set intersection from symmetric-key techniques. *ACM CCS'17*, 1257-1272 (2017).

**24.** Rindal, P. & Schoppmann, P. VOLE-PSI: fast OPRF and circuit-PSI from vector-OLE. *EUROCRYPT'21*, 901-930 (2021).

**25.** Garimella, G. et al. Oblivious Key-Value Stores and Amplification for Private Set Intersection. *CRYPTO'21*, 395-425 (2021).


ARTICLE IN PRESS

ARTICLE IN PRESS


**26.** Raghuraman, S. & Rindal, P. Blazing Fast PSI from Improved OKVS and Subfield VOLE. *ACM CCS'22*, 2505-2517 (2022).

**27.** Bienstock, A., Patel, S., Seo, J.Y., & Yeo, K. NearOptimal Oblivious Key-Value Stores for Efficient PSI, PSU and Volume-Hiding Multi-Maps. *USENIX Security'23*, 301-318 (2023).

**28.** Pinkas, B., Schneider, T., & Zohner, M. Scalable Private Set Intersection Based on OT Extension. *ACM Trans. Priv. Secur*, 1-35 (2018).

**29.** Pinkas, B., Schneider, T., Segev, G. & Zohner, M. Phasing: Private Set Intersection Using Permutation-based Hashing. *USENIX Security'25*, 515-530 (2015).

**30.** Kolesnikov, V., Kumaresan, R., Rosulek, M. & Trieu, N. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. *CCS'16*, 818-829 (2016)


## Funding

This work was supported by the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00399491, Development of Privacy-Preserving Multiparty Computation Techniques for Secure Multiparty Data Integration)

## Author contributions

Qiang Liu conceived and designed the ePSU scheme, completed the security proofs, set up the experimental environment and conducted the simulations, and wrote the manuscript. JaeYoung Bae was responsible for reviewing and revising the manuscript, validating the experimental results, and checking the security proofs. Joon-Woo Lee supervised the entire study and was responsible for finalizing and submitting the manuscript. All authors have read and approved the final version of the manuscript.

## Declarations

## Competing interests

The authors declare no competing interests.

## Additional information

**Correspondence** and requests for materials should be addressed to J.-W.L.

ARTICLE IN PRESS