Secure Join Operations in Multi-Identifier Databases: Performance and Practicality

Wen-jie Lu
TikTok
luwenjie@tiktok.com

Yongchuan Niu
TikTok
niuyongchuan.nyc@tiktok.com

Yongjun Zhao
TikTok
zhaoyongjun.280191@tiktok.com

Wei Dai
TikTok
weidai.d@tiktok.com

Donghang Lu
TikTok
donghang.lu@tiktok.com

Li Wang
TikTok
wangli.hz@tiktok.com

Qiang Yan
TikTok
yanqiang.mr@tiktok.com

ABSTRACT

In this work, we present an efficient and cryptographically secure protocol for multi-key inner-join computation that addresses the limitations of existing approaches. Our protocol leverages established Circuit Private Set Intersection (PSI) techniques to privately compute left-joins over individual key columns. These results are then securely aggregated into a final inner-join table using a novel private permutation protocol, which achieves a speedup of approximately 2× to 4× over prior methods.

To enhance utility without compromising privacy, we introduce a deduplication mechanism based on ordered left-joins, enabling first-key deduplication while revealing no sensitive matching information. We formally analyze the security of our construction in the semi-honest model. Furthermore, we optimize the equality testing subroutine, a core component of Circuit PSI, reducing its round complexity without an increase in computational overhead.

Empirically, our system demonstrates strong scalability, processing up to 1.8×10^4 records of 4 keys per second per CPU core. This represents a significant improvement over industry solutions such as Google's and Meta's, which are not only slower but also reveal more information about the input databases.

PVLDB Reference Format:
Wen-jie Lu, Yongchuan Niu, Yongjun Zhao, Wei Dai, Donghang Lu, Li Wang, and Qiang Yan. Secure Join Operations in Multi-Identifier Databases: Performance and Practicality. PVLDB, 19(9): 1854–1866, 2026.
doi:10.14778/3819518.3819519

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/tiktok-privacy-innovation/secure-multikey-join.

Proceedings of the VLDB Endowment, Vol. 19, No. 9 ISSN 2150-8097.
doi:10.14778/3819518.3819519

1 INTRODUCTION

Private set intersection (PSI) and its extensions have become foundational tools in privacy-preserving collaborative computation, especially in settings involving joint database analysis. A canonical use case involves two parties wishing to perform relational operations, such as inner-joins, on their respective datasets. Typically, this begins with identifying matching records through a PSI protocol on primary keys, followed by combining associated attributes from both databases. In the standard PSI framework, each party inputs a set X_i, and the protocol outputs their intersection $\cap_i X_i$ [30, 35]. Modern variants further support the evaluation of arbitrary public functions F over the intersection, yielding results such as $F(\cap_i X_i)$ without revealing the intersection itself [7, 19, 20, 22, 34, 36, 37].

However, most existing PSI protocols are designed around exact matching of *a single* primary key. This single-key assumption falls short in practical scenarios where entities are identified by multiple, heterogeneous attributes. Take cross-platform advertising attribution as an example: ad platforms and advertisers often need to track purchases by users who viewed ads, but face low matching rates due to inconsistent identifiers. A user might register with an email on an ad platform yet use a phone number on an advertiser's site, while others may provide both on the advertiser's side. To address this, a common strategy is to combine identifiers (e.g., email and phone number) to improve cross-platform user identification.

In such multi-identifier settings, naively executing independent PSI across the multiple identifier columns introduces significant risks. The recent works [18, 23] have shown that size-revealing PSI protocols (e.g., [22, 35]) are vulnerable to membership inference attacks. These vulnerabilities intensify in the multi-identifier setting, where attackers can exploit correlations between identifier sets to amplify attackers's power.

This motivates the need for principled, secure, and efficient protocols for identifing and combining matching rows across multiple identifier columns while preserving privacy. We propose such a solution, offering strong security guarantees, improved performance over the existing solutions within the PSI context.

1.1 Problem Definition

Our problem concerns computation over joint private databases. A key distinction from existing work is our handling of multiple matching keys. We denote the set of possible keys (i.e., identifiers) that either party may hold by $K = \otimes K_i$, where each set K_i corresponds to one column and contains a distinguished element $\perp \in K_i$ to represent missing entries.

To define a match between two rows of multiple keys S_1 and S_2, we define an equivalence relation $S_1 \sim S_2$ as follows: there exists an entry $s_{1,i} \neq \perp$ in S_1 and an entry $s_{2,j} \neq \perp$ in S_2 such that $s_{1,i} = s_{2,j}$. In other words, two rows are considered equivalent if they share at least one non-empty key in common.

The inner join of two databases is then defined as $\mathcal{J} = \mathcal{DB}_0 \bowtie \mathcal{DB}_1$, where each row $(A\|B) \in \mathcal{J}$ satisfies $A \sim B$, with $A \in \mathcal{DB}_0$ and $B \in \mathcal{DB}_1$. This is analogous to the SQL script: `SELECT * FROM` $\mathcal{DB}_0$ `INNER JOIN` $\mathcal{DB}_1$ `ON` $\mathcal{DB}_0.K_0$ `=` $\mathcal{DB}_1.K_0$ `UNION ALL SELECT * FROM` $\mathcal{DB}_0$ `INNER JOIN` $\mathcal{DB}_1$ `ON` $\mathcal{DB}_0.K_1$ `=` $\mathcal{DB}_1.K_1$. The `UNION ALL` operation can result in duplicate rows. In certain applications, deduplication is necessary to avoid redundancy in the resulting joined dataset. For example, counting unique users, a common metric known as *reach* in the advertising industry, requires removing duplicates to ensure accurate results. In this work, we consider *first-key deduplication*, which retains only the first matching key column according to a predefined order, which is also known in the literature as *waterfall matching* [4]. We write $\overset{\text{dd}}{\bowtie}$ to denote the deduplicated inner join.

Our goal is to design a private joint-database query functionality, denoted $\mathcal{F}_{\texttt{Query}}$ in Figure 1, which allows two parties to evaluate a public function F (e.g., sum or average) over their joint database $\mathcal{DB}_0 \overset{\text{dd}}{\bowtie} \mathcal{DB}_1$. $\mathcal{F}_{\texttt{Query}}$ captures a strict privacy requirement: beyond the computation results, only the database size is revealed to the other party. We highlight three privacy leakages that our protocol successfully avoids:

- **Leaking Intersection.** For the most efficient PSI, such as [12, 35, 36], they are inherently designed to compute the intersection. When using these PSI solutions to implement $\mathcal{F}_{\texttt{Query}}$, the matched keys within the the joint table $\mathcal{DB}_0 \bowtie \mathcal{DB}_1$ end up being disclosed to either party as unintended extra information.
- **Leaking Intersection Size.** Some specialized single-key PSI solutions [22, 31] enable private computation (e.g., summation over the joint table) without revealing the intersection itself. However, these approaches disclose the intersection size, which has been shown vulnerable to membership inference attacks like those studied in [18, 23].
- **Leaking One-to-Many Connections.** The current PSI solution [4] reveals one-to-many connections: for example, the 3rd row of the left table matches both the 3rd and 4th rows of the right table. However we consider such information should be kept private, as their exposure poses privacy risks. For example, aggregate patterns of one-to-many connections (e.g., "most left records match 1–2 right records, but 5% match ≥ 5") can leak sensitive statistics about the right table. This includes the density of entries per key, frequency of missing data, or the prevalence of high-cardinality groups (e.g., "a small subset of users generates 80% of transactions"). Such leaks compromise the privacy of the right table's schema or data distribution.

Private Query Functionality $\mathcal{F}_{\texttt{Query}}$

Parameters Party P_σ ($\sigma = 0, 1$) with a multi-key database $\mathcal{DB}_\sigma$, and a public function $F = (F_0, F_1)$ that describes a two-party functionality over the two databases.
Functionality: Upon receiving $\mathcal{DB}_0$ and $\mathcal{DB}_1$, send

$$\{F_0(\mathcal{DB}_0 \overset{\text{dd}}{\bowtie} \mathcal{DB}_1), |\mathcal{DB}_1|\} \text{ and } \{F_1(\mathcal{DB}_0 \overset{\text{dd}}{\bowtie} \mathcal{DB}_1), |\mathcal{DB}_0|\}$$

to P_0 and P_1, respectively.

Figure 1: Query functionality $\mathcal{F}_{\texttt{Query}}$ that responds to queries over the joint database.

We also highlight the distinction between this work and prior research on the *private record linkage* problem, which allows for fuzzy matching between records [12–14, 39]. In private record linkage, two rows from different parties' databases may be considered a match if they satisfy a certain closeness relation. For example, a record containing the email `abc@gmail.com` might be regarded as matching another record with `abe@gmail.com`. However, such fuzzy matching techniques may not be appropriate for applications that require exact matches, such as ad conversion tracking or other domains where precise identifier alignment is critical.

1.2 Our Contributions

In this work, we address two key challenges in secure inner-join computation over relational databases with multiple joining keys:

- **Privacy-Preserving Querying.** Ensuring correctness and privacy when querying the result of an inner-join operation, such that neither party learns more than the final output.
- **Efficiency in Multi-Key Matching.** Overcoming performance bottlenecks in exact matching protocols when applied to multiple identifiers, particularly in terms of communication round complexity and runtime scalability.

We tackle both challenges in a two-stage approach. First, leveraging existing PSI techniques, we introduce a single-key *left-join* protocol to compute the left-join table between two private databases. Particularly, this single-key left-join protocol will not revealing either the intersection or the intersection size. By executing this left-join protocol for multiple times, once per identifier, we obtain a set of left-join tables. Next, we construct the final private inner-join table from the computed left-join tables using a novel private permutation protocol.

In terms of efficiency, we present a new two-party (semi-honest) private permutation protocol that achieves approximately 2× – 4× speedup over the state-of-the-art permutation protocols [15, 32]. Additionally, we introduce an optimized protocol to perform deduplication within the resulting $\bowtie$ table, further enhancing both the performance and practicality of our solution.

2 PROTOCOL OVERVIEW

We consider a two-party setting with parties P_0 and P_1, referred to as C and S in client/server contexts. Our goal is to compute a secretly shared representation of the deduplicated inner-join table

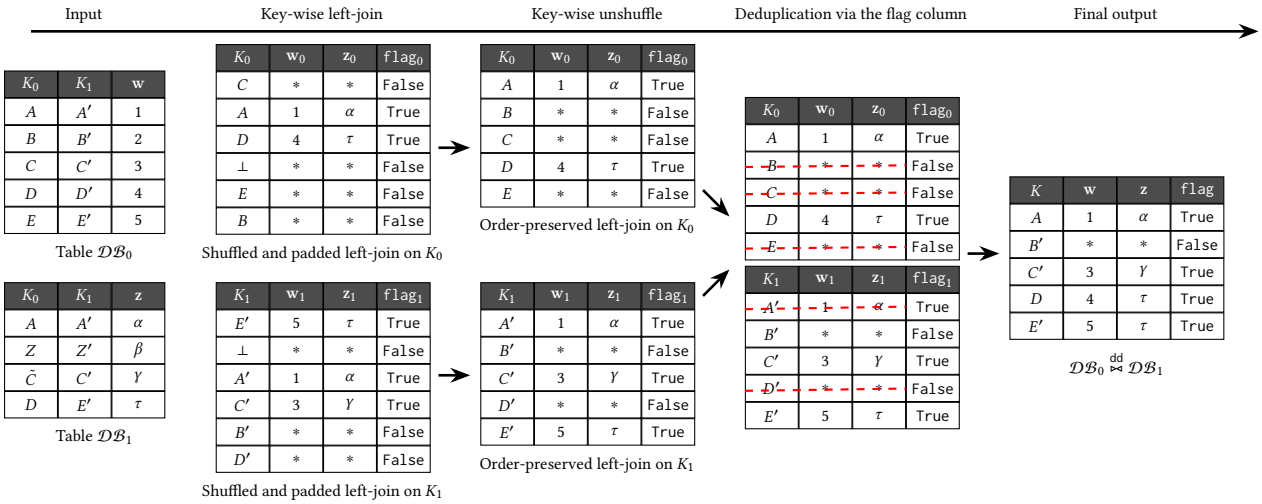


Figure 2: Workflow of our private inner-join framework on two databases $\mathcal{DB}_0$ and $\mathcal{DB}_1$ with two key columns. We first perform an order-preserving left join on each key column using a two-stage process: key-wise shuffled left join and key-wise unshuffling. Specifically, during the first stage, we generate an additional flag column to indicate whether each record has been matched. Each stage is implemented using private protocols to ensure that no information about the input tables or resulting tables is disclosed. The final joined table is constructed by aggregating the left join results and retaining only the first match, using the flag columns to facilitate this selection.

($\overset{\text{dd}}{\bowtie}$) to enable efficient querying using general-purpose secure two-party computation techniques, e.g., garbled circuits.

2.1 Overview

Figure 2 illustrates our framework's workflow, which consists of four stages:

(1) For each key column, we compute a *shuffled, padded left join*, supplemented by an additional flag column to indicate matching status. Padded keys are denoted using $\perp$, while unmatched values are marked by '*'.
(2) To concatenate the outputs of these shuffled left joins and construct the inner join ($\bowtie$) table, we first unshuffle the left join result lists generated in Stage 1. This process yields *order-preserved left join* lists, where all padding rows are relocated to the end of each list.
(3) By simply concatenating these order-preserved lists in a row-wise fashion, we obtain the initial inner join table ($\bowtie$). Notably, this table may contain duplicate entries.
(4) Finally, we leverage the flag column to deduplicate the $\bowtie$ results, yielding the final deduplicated inner join. Unmatched rows are retained to protect the privacy of both the intersection and its size.

We now describe each step in detail.

2.1.1 Shuffled and Padded Single-Key Left Join. We begin by defining the order-preserving left join on a single key column. Let $A_0 = (k_i, w_i)_{i\in I}$ and $A_1 = (k'_j, w'_j)_{j\in J}$ be two single-key databases, where k_i, k'_j denote keys and w_i, w'_j denote associated values. The order-preserving left-join $\texttt{OrderPresLeftJoin}$ generates a list L:

$$L := (w_i, z_i, \texttt{flag}_i)_{i\in I} \leftarrow \texttt{OrderPresLeftJoin}(A_0, A_1).$$

The i-th entry of L is parsed as $(w_i, z_i, \texttt{flag}_i)$ where

$$(z_i, \texttt{flag}_i) = \begin{cases} (w'_j, \texttt{True}) & \text{if } \exists (k_i, w'_j) \in A_1 \\ (0, \texttt{False}) & \text{otherwise} \end{cases} \tag{1}$$

In other words, the flag $\texttt{flag}_i = \texttt{True}$ indicates the i-th entry in A_0 is matched by an entry in A_1 via the key k_i. If no such match exists in A_1, we assign the value 0 and the $\texttt{False}$ flag[1].

The existing circuit-based PSI protocols [7, 36, 37] already implement a **shuffled and padded variant** of the left-join functionality, as depicted in Figure 2.1.3. Specifically, the left-join list L is generated in secret-shared form with a random permutation and dummy padding. (We defer the formal definition of secret sharing to a later section.) This $\mathcal{F}^{\epsilon}_{\texttt{PSLeftJoin}}$ functionality exhibits three key features:

(1) The output list is padded with dummy entries, controlled by a hyperparameter $\epsilon > 0$ which is determined by the concrete implementation of $\mathcal{F}_{\texttt{PSLeftJoin}}$.
(2) The output list is permuted by a random permutation π, which is considered part of P_0's private output.
(3) P_0 knows the positions of the dummy entries in the output list, even though neither party learns the matching.

To concatenate the multiple left-join lists, we must restore the randomly shuffled list L' (generated by Figure 2.1.3) to its original order as defined in (1). Although applying the inverse permutation π^{-1} (known to P_0) would achieve this, π contains private information about P_0's input set and cannot be revealed to P_1. To address this, we employ a private permutation protocol $\mathcal{F}_{\texttt{Perm}}$ (see Figure 4) that allows the parties to compute the inverse permutation π^{-1}

[1]In practice, we assign any value for the the unmatched rows depends on the specific query. For instance, we can assign 0 if we want to sum the assoicated values of the matched rows. On the other hand, if we want to count how many associated values of the matched rows is zero, we can assign a non-zero value for the unmatched rows.

Padded and Shuffled Left-join Functionality $\mathcal{F}^{\epsilon}_{\texttt{PSLeftJoin}}$

Parameters P_σ with a single-key database A_σ. A public parameter $\epsilon > 0$.
Functionality: Upon receiving A_0 and A_1, proceed as follows.

(1) Let $n = |A_0|$ and $n' = \lceil (1+\epsilon)n \rceil$.
(2) Derive a random permutation $\pi : [n'] \mapsto [n']$ from A_0.
(3) Compute the order preserving list $L \leftarrow \texttt{OrderPresLeftJoin}(A_0, A_1)$ according to (1), and append L with $n' - n$ dummy entries in the form of $(0, 0, \texttt{False})$ to obtain a longer list L'.
(4) Give the corresponding secret share of the permuted list $\pi(L')$ to P_0 and P_1. Also give the permutation π to P_0.

Figure 3: Padded and shuffled left-join functionality.

without exposing either the input vector or the permutation itself. Specifically, P_0 provides the permutation π^{-1}, and P_1 provides its shares of the list $\pi(L')$ to $\mathcal{F}_{\texttt{Perm}}$, resulting in corresponding shares of the list L'. The parties then discard the last $n' - n$ dummy entries of L', yielding a clean, ordered output consistent with definition (1).

2.1.2 First-Key Deduplication. First-key deduplication on order-preserving left-join lists can be abstracted as follows. Let $L = (b_l, q_l)_{l=1}^{m}$ be a list where each $b_l \in \{0, 1\}$ is a bit flag indicating a match (i.e., $\texttt{flag} = \texttt{True}$) and q_l is the associated value. If at least one $b_l = 1$, we output the value q_l corresponding to the smallest index l; otherwise, we output zero. We compute this as follows:

(1) Initialize $c = 0$ and $q = 0$, where c is a control flag indicating whether a match has been found.
(2) For each $l = 1$ to m, sequentially update: $q \leftarrow q + ((1-c) \wedge b_l) \cdot q_l$ and $c \leftarrow c \vee b_l$.

After processing, if any $b_l = 1$, q holds the value q_l of the first match (smallest index). The control flag c ensures only the first match contributes to q: once c becomes 1, the factor $(1 - c)$ nullifies subsequent updates. If all $b_l = 0$, q remains 0. The logical operations (AND/OR) and multiplication are implemented using standard secure two-party computation techniques on secretly shared values.

2.1.3 Strong Privacy Guarantees. We evaluate the multi-key inner join while revealing only the *input size*. This is achieved by decomposing the inner join into multiple order-preserving (single-key) left joins. On one hand, the matching information for each key column remains private through the single-key left-join protocol, ensuring that both the intersection and its size are concealed. On the other hand, the order-preserving property of the left join enables deduplication without revealing actual matches on individual key columns, thereby preventing leakage of one-to-many relationships.

2.2 Choosing the Suitable Private Permutation

Several existing protocols implement the ideal functionality $\mathcal{F}_{\text{perm}}$ in Figure 4, including [8, 32, 33]. For our inner-join application, we find that the Mohassel-Sadeghian protocol [32] offers particularly strong advantages. We highlight two key benefits that make it well-suited for our setting:

- **Share Format Flexibility.** The Mohassel-Sadeghian permutation protocol natively supports both Boolean and arithmetic secret sharing schemes—making it highly compatible with our target functionality $\mathcal{F}_{\texttt{Query}}$, whose output format depends on downstream processing. When subsequent operations (e.g., the query function F in $\mathcal{F}_{\texttt{Query}}$) involve arithmetic computations such as linear algebra routines, producing payload vectors directly as arithmetic shares avoids costly conversions between share types. In contrast, other permutation protocols such as [33] require Boolean-share inputs, which can incur expensive Boolean-to-arithmetic conversions when arithmetic processing is needed.
- **Balanced Bandwidth-Computation Tradeoff.** While the Mohassel-Sadeghian protocol incurs higher communication overhead (e.g., approximately 2.3×) compared to state-of-the-art permutation protocols at scale (e.g., for $n = 2^{25}$ elements), this moderate increase in bandwidth usage enables significant computational gains. By relying solely on lightweight symmetric cryptographic primitives, the protocol achieves lower computational latency, making it more efficient overall in practical deployments where total runtime (communication + computation) is the primary constraint.

We further improve upon the Mohassel-Sadeghian's protocol [32] by introducing a novel optimization that reduces its communication cost by half, without increasing computational overhead.

Functionality $\mathcal{F}_{\texttt{perm}}$

Parameters: P_1 with $\mathbf{x} \in \mathbb{Z}^n$, P_0 with $\pi : [n] \mapsto [n]$.
Functionality: Upon receiving the private vector $\mathbf{x}$ from P_1, and the private permutation π from P_0, sample $\mathbf{r} \leftarrow \mathbb{Z}^n$, and compute $\mathbf{y} = \pi(\mathbf{x}) \oplus \mathbf{r}$. Give the masked vector $\mathbf{y}$ to P_0, and give $\mathbf{r}$ to P_1.

Figure 4: Ideal functionality for private permutation. The operation $\oplus$ can be arithmetic addition.

Table 1: Related systems that support private join on two databases with multiple keys.

Method	Match	Privacy	Efficiency	Deduplication	Private Queryable
[24]	Fuzzy	◑	◑	No	No
[41]	Fuzzy	◕	◔	No	No
[4]	Exact	◔	◔	Yes	Yes
[25]	Exact	○	◕	No	Yes
Ours	Exact	○	○	Yes	Yes

The partial order is ◕ ⪯ ◑ ⪯ ◔ ⪯ ○ where ○ is the best.

2.3 Related Work

Table 1 summarizes existing systems that support private joins on multi-key databases.

Systems employing fuzzy matching, such as [24, 41], reveal the contents of the resulting joint table itself. Moreover, [41] also leaks the number of comparisons required to construct the joint table, which, according to their analysis, can disclose approximately 9 bits of information about the input database [41].

In contrast, exact matching systems like [4] only reveal the size of the joint table, but still leak certain one-to-many relationships. For instance, P_0 may learn that two rows in P_1's database match a single row in P_0's database—even if P_0 does not learn the specific matching row. The work of [25] provides stronger privacy guarantees than [4], although it lacks support for deduplication.

Furthermore, these existing systems struggle with scalability, making them unsuitable for large-scale applications. For example, [41] reports a throughput of only 11.1 records per second, while [25] achieves a slightly better rate of 29 records of 4 keys per second, albeit at the cost of transmitting thousands of gigabytes of data.

We propose a significantly more efficient and cryptographically secure system that supports multi-key joins with enhanced privacy and greater functional versatility. Empirically, our system achieves a throughput of approximately 6×10^3 to 1.8×10^4 records of 4 keys per second per CPU core.

Organization. The rest of this work is organized as follows: Section 3 presents necessary cryptographic preliminaries and background concepts. Sections 4 and 5 detail our optimized constructions for two building blocks: private permutation and equality testing respectively. Section 6 introduces our multi-key inner-join protocol, constructed through a hybrid argument combining permutation and equality testing functionalities. Section 7 provides comprehensive experimental evaluation and analysis. Then we conclude with a discussion of related works and final remarks.

3 PRELIMINARIES

3.1 Notation

We denote by κ and λ the computational and statistical security parameters, respectively. The function $\mathbf{1}\{\mathcal{P}\}$ returns 1 when the predicate $\mathcal{P}$ is true and 0 when $\mathcal{P}$ is false. We use $[m]$ to denote the set $\{0, 1, \cdots, m-1\}$ for a natural number $m \in \mathbb{N}$. We write $\mathbb{B} = \{0, 1\}$. We use bold letters such as $\mathbf{a}$ to represent vectors, and use a_j to denote the j-th component of $\mathbf{a}$. Given a permutation π on n items, we use $\pi(\mathbf{x})$ to denote the permuted vector $[x_{\pi(0)}, x_{\pi(1)}, \cdots, x_{\pi(n-1)}]$. run between a party P_0 and a party P_1.

3.2 Threat Model

3.2.1 The Ideal Functionality Paradigm [6]. " In the ideal process all parties secretly hand their inputs to an external trusted party who locally computes the outputs according to the specification, and secretly hands each party its prescribed outputs. This ideal process can be regarded as a "formal specification" of the security requirements of the task. Indeed, in the ideal process both correctness and lack of influence are guaranteed in fiat, since the inputs provided by any adversarial set of parties cannot depend on the inputs provided by the other parties in any way, and furthermore all parties obtain the correct output value according to the specification. (Canetti [6], 2006, page 8). " For example, the $\mathcal{F}_{\mathtt{Query}}$ functionality (Figure 1) specifies two core properties:

- Correctness: All parties obtain the computation results over the joined table, i.e., $F_0(\mathcal{DB}_0 \bowtie \mathcal{DB}_1)$ and $F_1(\mathcal{DB}_0 \bowtie \mathcal{DB}_1)$;
- Privacy: No party learns the actual input databases or the raw joined table itself.

Additionally, the size of each party's own database will be disclosed to the other party.

3.2.2 Simulation-based Security. We consider the semi-honest security model [16], where each party faithfully follows the protocol specification but may attempt to infer additional information about the other party's private inputs based on the received messages. We assume that an adversary Adv can corrupt at most one party, and the corruption strategy is static. That is, the set of corrupted parties is determined before the protocol execution begins and remains fixed throughout.

The privacy requirement ensures that the adversary learns nothing beyond the corrupted party's input and the final output of the protocol. Following prior work [7, 34, 37], we formalize our privacy guarantees using the ideal/real simulation paradigm.

Let $\mathcal{F}$ denote an ideal functionality in which the parties submit their inputs to a trusted third party, who computes the desired function and returns the result. Let Π be the real-world protocol executed by the two parties. We say that Π securely realizes $\mathcal{F}$ if the following condition holds.

Definition 1 ([16, 26]). *A protocol Π securely realizes an ideal functionality $\mathcal{F}$ if for every adversary* Adv *in the real world, there exists a simulator* Sim *in the ideal world such that, for all environments $\mathcal{E}$, the following quantity is negligible in the security parameter κ:*

$$\left|\Pr\left(\mathtt{REAL}_{\Pi,\mathtt{Adv},\mathcal{E}}(\kappa) = 1\right) - \Pr\left(\mathtt{IDEAL}_{\mathcal{F},\mathtt{Sim},\mathcal{E}}(\kappa) = 1\right)\right|$$

As in prior works, our construction combines multiple sub-protocols that realize smaller private computations. To simplify both the description and the security analysis, we use the **hybrid model**. A protocol that invokes an ideal functionality $\mathcal{F}_{\mathsf{sub}}$ is said to operate in the $\mathcal{F}_{\mathsf{sub}}$-hybrid model. This model is similar to the real model, except that certain sub-protocols are replaced by ideal calls to $\mathcal{F}_{\mathsf{sub}}$. One can imagine an oracle that faithfully computes the ideal functionality in the ideal world. This approach is justified by the composition theorem for the semi-honest model [5].

3.2.3 No Guarantee Against Side-Channel Attacks. Side-channel attacks (e.g., timing attacks or access pattern attacks) are not captured by Definition 1. However, in our secure join protocol, both parties are required to fully scan their local databases. Thus, the access pattern in our protocol is data-independent.

3.3 2PC via Secret Sharing

Secure two-party computation (2PC) allows two parties to jointly compute a public function over their inputs while keeping the inputs private. In this work, we utilize additive secret sharing for 2PC. We refer to a value $x \in \mathbb{Z}_{2^\ell}$ that is additively shared among P_0 and P_1 as a secretly shared value, and denote it as $[\![x]\!] = ([\![x]\!]_0, [\![x]\!]_1)$ where P_0 holds a random value $[\![x]\!]_0$ and P_1 holds a random value $[\![x]\!]_1$. To reconstruct x, the two parties exchange her own share, and then compute $x = [\![x]\!]_0 + [\![x]\!]_1 \bmod 2^\ell$. Similarly, we write $\langle y \rangle = (\langle y \rangle_0, \langle y \rangle_1)$ to denote the Boolean sharing of the value $y \in \mathbb{B}^\ell$ such that $y = \langle y \rangle_0 \oplus \langle y \rangle_1$. Also we omit the subscript and only write $[\![x]\!]$ or $\langle y \rangle$ when the ownership is irrelevant from the context.

Given secretly shared values, we can leverage the existing secure computation primitives from [11, 29]: secure addition, secure multiplication and secure logical OR/AND. The output of any secure computation is also a secretly shared value unless it is reconstructed.

3.4 Oblivious Transfer

Oblivious transfer (OT) serves as the cryptographic foundation for non-linear computations in our protocol. A standard 1-out-of-n OT ($\binom{n}{1}$-$\mathtt{OT}_\ell$) involves a sender with n messages $\{m_0, \ldots, m_{n-1}\} \in$

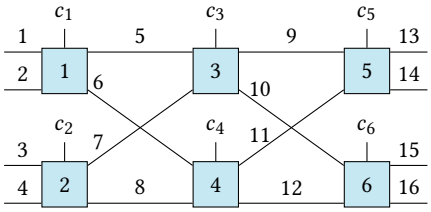


Figure 5: A Beneš network of 6 gates with 16 wires. Each wire is labeled with an index and the function of the gates are controlled by the bit c.

$\{0,1\}^\ell$ and a receiver with a choice index $c \in \{0,\ldots,n-1\}$. The protocol guarantees the receiver learns only m_c while the sender gains no information about c. Efficiency improves significantly when sender messages exhibit structured correlations. We leverage two optimized variants: **Random OT (ROT)** where sender's messages are uniformly random, and **Correlated OT (Δ-OT)** where messages satisfy $m_1 = m_0 + \Delta$ for fixed $\Delta \in \mathbb{Z}_\ell$. We instantiate both ROT and Δ-OT using the Ferret protocol [42], which achieves sublinear communication costs via efficient vector-OLE extensions.

4 BUILDING BLOCK: PRIVATE PERMUTATION VIA SWITCHING NETWORK

The private permutation protocol serves as a foundational primitive in our multi-key framework. This section presents the optimized private permutation protocol proposed in this work.

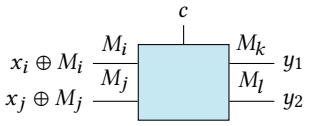
4.1 Switching Network

A switching network $\mathbb{S}$ is a directed acyclic graph composed of configurable logic gates. Each gate contains: two input wires (w_i, w_j); two output wires (w_k, w_l); A control wire determining routing behavior. The gate operates in two modes based on the control signal: If the control wire is off, the gate will directly pass the inputs. If the control wire is on, the gate will swap the two inputs.

Beneš et al. [3] demonstrate that any permutation of n elements can be realized using a switching network with $O(n \log_2 n)$ gates. Figure 5 illustrates an example of Beneš network. The topology of a Beneš network consists of two back-to-back butterfly networks, making its structure permutation-independent. This permutation-independent structure allows the network topology to serve as a public parameter, enabling secure computation where neither party's confidential permutation (encoded via private control bits) is revealed during protocol execution. The Waksman's network [40] optimizes Beneš's network from $n \log_2 n - n/2$ gates to $n \log_2 n - n + 1$ gates. We retain the Beneš construction in our implementation due to a subtle practical difference (e.g., less than 2.7% for $n > 2^{20}$).

4.2 Mohassel et al.'s Private Permutation

We begin by summarizing the Mohassel et al.'s private permutation protocol [32]. Consider a single gate in Figure 6. The protocol proceeds as follows: $\mathcal{S}$ assigns four uniformly random masks M_i, M_j, M_k, and M_l to the gate's input and output wires. C holds blinded wire values $x_i \oplus M_i$ and $x_j \oplus M_j$ (received from $\mathcal{S}$) for the two input wires . The protocol ensures that C learns the blinded



c	y_1	y_2
0	$x_i \oplus M_k$	$x_j \oplus M_l$
1	$x_j \oplus M_k$	$x_i \oplus M_l$

Figure 6: Oblivious evaluation of a gate where inputs/outputs are all randomly masked.

output wire values without revealing the control bit c to $\mathcal{S}$ or the plaintext messages x_i, x_j to C. This is achieved through the following steps:

(1) $\mathcal{S}$ constructs a two-row table: $\text{Row}_0 = M_i \oplus M_k \| M_j \oplus M_l$ and $\text{Row}_1 = M_i \oplus M_l \| M_j \oplus M_k$.
(2) C and $\mathcal{S}$ execute a 1-out-of-2 OT: $\mathcal{S}$ provides the two rows as OT inputs. C uses the control bit c as the OT choice.
(3) If $c = 0$, C receives Row_0 and computes:
$$y_1 = (x_i \oplus M_i) \oplus (M_i \oplus M_k) = x_i \oplus M_k$$
$$y_2 = (x_j \oplus M_j) \oplus (M_j \oplus M_l) = x_j \oplus M_l$$
(4) If $c = 1$, C receives Row_1 and computes:
$$y_1 = (x_j \oplus M_j) \oplus (M_j \oplus M_k) = x_j \oplus M_k$$
$$y_2 = (x_i \oplus M_i) \oplus (M_i \oplus M_l) = x_i \oplus M_l$$

The communication cost for evaluating a single gate is 4ℓ bits, where ℓ is the bit length of $\mathcal{S}$'s inputs.

Random OT Optimization. Instead of first choosing random wire masks for this gate's output wire(s) and using those to determine the OT payloads, one can let the OT extension dictate the first OT payload randomly, and use that to solve for the appropriate wire mask. Particularly, C and $\mathcal{S}$ invokes an ROT which gives $r_0, r_1 \in \{0,1\}^{2\ell}$ to $\mathcal{S}$ and gives r_c to C. They parse the first ROT message as the first row of MS13's random table. That is
$$r_0 = r_0^L \| r_0^R = M_i \oplus M_k \| M_j \oplus M_l.$$
By first fixing some (random) values[2] for the wires M_i, M_j, $\mathcal{S}$ can solve the remaining masks M_k, M_l. Then $\mathcal{S}$ sends two correction words $\text{CW}^0 = M_i \oplus M_l \oplus r_1^L$ and $\text{CW}^1 = M_j \oplus M_k \oplus r_1^R$ to C, allowing him to adjust the ROT values. Particularly, when $c = 1$, C performs the modifications $y_1 = (x_j \oplus M_j) \oplus r_1^R \oplus \text{CW}^1 = x_j \oplus M_k$, and $y_2 = (x_i \oplus M_i) \oplus r_1^L \oplus \text{CW}^0 = x_i \oplus M_l$. On the other hand, if $c = 0$, C simply computes $y_1 = (x_i \oplus M_i) \oplus r_0^L = x_i \oplus M_k$ and $y_2 = (x_j \oplus M_j) \oplus r_0^R = x_j \oplus M_l$. The communication costs for evaluating one gate becomes $O(2\ell)$ bits since the random OT does not incur online communication.

4.3 Proposed Private Permutation with Only Half Communication

The MS13 protocol (even with random OT optimization) generates independent random masks for gate wires. We observe this independence is unnecessary for security - only uniform randomness is required. By introducing the structured correlation:
$$M_i \oplus M_j \oplus M_k \oplus M_l = 0$$

[2]For a gate in the first layer, $\mathcal{S}$ samples M_i, M_j uniformly at random. For a gate in the middle layer, M_i and M_j are given as the output from the previous layer.

Protocol $\Pi_{\texttt{perm}}$

C's inputs: A sequence of control bits $\mathbf{c} = (c_1, c_2, \cdots, c_q) \in \mathbb{B}^q$ for $\mathbb{S}$. We denote the associated permutation with $\mathbb{S}$ and $\mathbf{c}$ by π.
$\mathcal{S}$'s inputs: A message vector $\mathbf{x} = (x_1, x_2, \cdots, x_n)$ where $x_i \in \mathbb{B}^\ell$.
Outputs: C learns $\{z_i = x_{\pi(i)} \oplus v_i\}_{i=1}^n$ and $\mathcal{S}$ learns $\{v_i\}_{i=1}^n$.

1: For each gate u in $\mathbb{S}$, C and $\mathcal{S}$ peform 1-out-of-2 Random OT where C (the receiver) input a bit c_u. After the Random OTs, $\mathcal{S}$ obtains $r_{u,0}, r_{u,1} \in \mathbb{B}^\ell$ and C obtains r_{u,c_u}.
2: For each input wire of $\mathbb{S}$, $\mathcal{S}$ assigns a random bitstring from $\mathbb{B}^\ell$.
3: In topological order, for each gate u with input wires w_i, w_j and output wires w_k, w_l, $\mathcal{S}$ does the following:
   (1) Compute the masking for the output wires $M_k = M_i \oplus r_{u,0}$ and $M_l = M_j \oplus r_{u,0}$.
   (2) Compute a correction word $\mathrm{CW}_u = M_i \oplus M_l \oplus r_{u,1}$.
4: $\mathcal{S}$ sends $\{y_i = x_i \oplus M_i\}_{i=1}^n$ and $\{\mathrm{CW}_u\}_{u=1}^q$ to C.
5: In topological order, for each gate u with input wires w_i, w_j and output wires w_k, w_l, C does the following:
   (1) If $c_u = 0$ then $y_k = y_i \oplus r_{u,0}$ and $y_l = y_j \oplus r_{u,0}$.
   (2) Else then $y_k = y_j \oplus r_{u,1} \oplus \mathrm{CW}_u$ and $y_l = y_i \oplus r_{u,1} \oplus \mathrm{CW}_u$.
6: For each output gate h ($1 \le h \le n/2$) with output wires w_k, w_l, C sets $z_{2h} = y_k, z_{2h+1} = y_l$, and $\mathcal{S}$ sets $v_{2h} = M_k, v_{2h+1} = M_l$.

Figure 7: Proposed private permutation protocol under the $\mathcal{F}_{\texttt{ROT}}$-hybrid model.

achieved via $M_l = M_i \oplus M_j \oplus M_k$, we enable correction word compression. Single correction word per gate suffices due to:

$$\mathrm{CW}^0 \oplus \mathrm{CW}^1 = (M_i \oplus M_l \oplus r_1^L) \oplus (M_j \oplus M_k \oplus r_1^R) = 0$$

This reduces communication to $O(\ell)$ bits per gate (50% savings).

Figure 7 formalizes our optimized private permutation protocol. The protocol is basically a three rounds protocol. In the beginning, C and $\mathcal{S}$ run a batch of ROTs parallelly where C (i.e., owner of the private permutation) provides the choice bits. In Step 2, $\mathcal{S}$ first assign the random masks for each input wires of $\mathbb{S}$. Then $\mathcal{S}$ propagates from an input gate to through to an output gate, computing the corresponding masks for the output wires of each gate, and computing the correction word for each gate. The intersection in Step 4 delivers the masked inputs and correction words to C, allowing him to reconstruct permuted shares in Step 5. The output shares of $\mathcal{S}$ (resp. C) are the maskings over the output wires of $\mathbb{S}$ it computed in Step 3 (resp. Step 5).

Theorem 1. *The protocol in Figure 7 securely computes $\mathcal{F}_{\texttt{perm}}$ in Figure 4 against semi-honest adversaries in the $\mathcal{F}_{\texttt{ROT}}$-hybrid model.*

The correctness of Figure 7 follows the arguments in [15, 32] since we do not change the way how the switching network is evaluated. We provide a sketch proof as below and defer the detailed simulation to full version.

Proof. (Sketch) The protocol's privacy stems from the uniform randomness of all components in C's view:

$$\texttt{View}_C^{\Pi_{\texttt{perm}}} = \left\{\{r_{u,c_u}\}_{u=1}^q, \{y_i = x_i \oplus M_i\}_{i=1}^n, \{\mathrm{CW}_u\}_{u=1}^q\right\}.$$

Formally, it comprises three elements: 1) ROT outputs $\{r_{u,c_u}\}_{u=1}^q$, 2) masked inputs $\{y_i = x_i \oplus M_i\}_{i=1}^n$, and 3) correction words $\{\mathrm{CW}_u\}_{u=1}^q$. Privacy holds because: 1) OT outputs are uniformly random by $\mathcal{F}_{\texttt{ROT}}$ construction, 2) masked inputs y_i are random due to uniform M_i, 3) correction words CW_u are one-time padded (OTP) properly. Recall that $\mathrm{CW}_u = M_i \oplus M_l \oplus r_{u,1}$. The DAG structure of switching network $\mathbb{S}$ ensures mask propagation $M_i = M_{i'} \oplus r_{u_i,0}$ (resp. $M_l = M_{l'} \oplus r_{u_l,0}$) where $M_{i'}$ and $M_{l'}$ originate from distinct input wire masks. This acyclic propagation preserves uniform randomness through successive XOR operations with first ROT messages ($r_{u,0}$). As a result we can rewrite CW_u in the following the form

$$\mathrm{CW}_u = e_u^* \oplus r_{u,0} \oplus r_{u,1}$$

where e_u^* contains only: mask combinations (e.g., $M_i \oplus M_j$) **without** prior ROT messages $r_{u',1}$ from the previous gates ($u' < u$), or the ROT messages $r_{u,0/1}$ on the current gate. C has selected r_{u,c_u} on his choice c_u. The unselected ROT message $r_{u,1-c_u}$ serves as a fresh OTP, making CW_u uniformly random regardless of C's choice c_u.

A simulator $\texttt{Sim}_C^{\Pi_{\texttt{perm}}}$ can generate $\texttt{View}_C^{\Pi_{\texttt{perm}}}$ by sampling all components uniformly at random, as neither $\{y_i\}$ nor $\{\mathrm{CW}_u\}$ contain information beyond what is already masked by independent random values.

On the other hand, $\mathcal{S}$'s view consists of the ROT outputs only. Formally $\texttt{View}_{\mathcal{S}}^{\Pi_{\texttt{perm}}} = \left\{\{r_{u,0}, r_{u,1}\}_{u=1}^q\right\}$. The ROT outputs $\{r_{u,0}, r_{u,1}\}$ are uniformly random due to the $\mathcal{F}_{\texttt{ROT}}$ functionality.

Notably, we do not need to simulate the values $\{v_i\}_i$ as they are the output of $\mathcal{S}$ and will be given to the simulator according to the Definition 1. While we show that if $\mathcal{S}$ is honest, these values $\{v_i\}_i$ are also properly distributed. Particularly, each v_i follows the form $v_i = M_i \oplus r_i^*$ where M_i is the random mask on the i-th input wire, and r_i^* is derived exclusively from ROT messages and contains no additional dependency on other input wire masks. When $\mathcal{S}$ samples the wire masks $\{M_i\}_{i=1}^n$ uniformly at random, they will serve as OTPs in v_i. □

Arithmetic Share Extension. Our method extends naturally to arithmetic shares by interpreting table rows as $M_i - M_k \| M_j - M_l$ and $M_i - M_l \| M_j - M_k$, and then enforcing correlation $M_i + M_j \equiv M_k + M_l \pmod{2^\ell}$. This leads to the correction words $\mathrm{CW}^0 \equiv -\mathrm{CW}^1 \pmod{2^\ell}$. In other words, the communication is still $O(\ell)$ bits per gate in this arithmetic setting.

Complexity. The communication complexity of the secure permutation protocol (Figure 7) is $O(\ell \cdot n \log_2 n)$ using the advanced ROT protocol [42]. This bound holds for both boolean shares and arithmetic shares.

5 BUILDING BLOCK: EQUALITY TEST

5.1 The Necessity of Private Equality Test

As previously described, $\mathcal{F}_{\texttt{PSLeftJoin}}$ the single-key left-join functionality (see Figure 2.1.3) is securely realized using Circuit PSI protocols such as those in [7, 36, 37]. Modern Circuit PSI protocols largely follow the framework introduced by Pinkas et al. [34], which combines Oblivious Programmable Pseudorandom Functions with private equality testing to compute membership predicates $\mathbf{1}\{a_i \in B\}$ for each element $a_i \in A$.

A core building block in Pinkas's construction is the private equality test functionality, denoted $\mathcal{F}_{\texttt{EqT}}$. In this two-party protocol, party C holds input $a \in \{0,1\}^\ell$ and party $\mathcal{S}$ holds input $b \in \{0,1\}^\ell$, where ℓ is the bit-length of the inputs. At the end of the protocol,

both parties obtain Boolean secret shares of the predicate $\mathbf{1}\{a = b\}$, without learning any additional information about each other's inputs. Among the many existing private equality testing protocols, such as [7, 9, 10, 17, 27, 43], we choose the protocol by Chandran et al. [7], which offers a favorable balance between total communication cost and round complexity.

Although the equality test protocol of Chandran et al. performs well in terms of bandwidth, we observe that it still dominates the round complexity of our private inner-join framework. Specifically, while other phases of the protocol—such as left-join execution and private permutation—can be completed in 2 to 4 rounds, the equality testing component may require more than 12 rounds. This observation motivates us to optimize the round complexity of Chandran et al.'s equality testing protocol, aiming to reduce the number of interactive rounds without significantly increasing communication or computational overhead.

5.2 Proposed Private Equality of Constant Rounds Complexity

We first recap Chandran et al.'s equality protocol, then demonstrate our optimizations. Consider the case of single equality testing, i.e., computing $\mathbf{1}\{x = y\}$ where $x, y \in \{0,1\}^\ell$ is the private input from P_0 and P_1 respectively. Let $x = x_1 \| x_0$ and $y = y_1 \| y_0$, where $x_0, y_0, x_1, y_1 \in \{0,1\}^{\ell/2}$. The equality computation can be recursively reduced as: $\mathbf{1}\{x = y\} = \mathbf{1}\{x_1 = y_1\} \wedge \mathbf{1}\{x_0 = y_0\}$. This approach constructs a binary computation tree down to t-bit leaves, which are evaluated using 1-out-of-2^t OT. The results are then combined through logical $\wedge$ operations in a bottom-up traversal as: $\mathbf{1}\{x = y\} = \bigwedge_{i=0}^{d-1} \mathbf{1}\{x_i = y_i\}$. Formally, Chandran et al.'s protocol requires: $d = \lceil \ell/t \rceil$ instances of 1-out-of-2^t OT and $d-1$ instances of logical $\wedge$ operations, and $O(\log_2 \ell - \log_2 t)$ rounds of communication. Notably, when $t = 1$, this construction collapses to the classic GMW protocol [17].

Our constant-round optimization. We present a way to optimize the logarithmic rounds of communications to constant rounds. Specifically, we reform the equation above to (2) below.

$$\mathbf{1}\{x = y\} = \mathbf{1}\left\{ d = \sum_{i=0}^{d-1} \mathbf{1}\{x_i = y_i\} \right\} \tag{2}$$

That is we merge the AND-tree into a single zero-testing procedure by summing the d bits on the leaf nodes: To evaluate (2), we need: 1) **Boolean-to-Arithmetic (B2A) conversion**: Transform secret-shared bits $\mathbf{1}\{x_i = y_i\}$ into arithmetic shares for summation. 2) **Zero-testing**: Verify the final equality through a single 1-out-of-2^k OT instance, where $k = \lceil \log_2(d+1) \rceil$ since we are adding d bits.

This approach achieves constant communication rounds regardless of input width ℓ. The round complexity breaks down as: 2 rounds for leaf nodes equality tests, 2 rounds for B2A conversions, and 2 rounds for zero-testing via OT. For the B2A, we only need outputting $k = \lceil \log_2(1+d) \rceil$ since we are summing up d bits.

Complexity. Based on the Ferret OT [42], the communication complexity of our protocol is $O(d \cdot (2^t + t) + d \cdot (2 + \lceil \log_2 d \rceil) + (2^k + k))$ bits where $d = \lceil \ell/t \rceil$ and $k = \lceil \log_2(d+1) \rceil$. For $\ell = 64, t = 3$, this leads $d = 22$ and $k = 5$, resulting about 435 bits per equality test.

Multi-key Inner-join Protocol with Deduplication $\Pi_{\texttt{InnerJoin}}$

Inputs:

- P_0: $\mathcal{DB}_0$ with $m > 0$ key columns and n_0 records.
- P_1: $\mathcal{DB}_1$ with $m > 0$ key columns and n_1 records.
- Key uniqueness: $\forall i \neq j, S_i, S_j \in \mathcal{DB}_0, S_i \not\simeq S_j$ (Same for $\mathcal{DB}_1$)
- Role preference: $n_0 \leq n_1$ (standard client/server configuration)

Auxiliaries:

- Let $A_{l,\sigma}$ being the subtable of $\mathcal{DB}_\sigma$ with the l-th key column assoicated with the value column for $l \in [m]$ and $\sigma = 0, 1$.
- Let $\tilde{n} = \lceil (1+\epsilon) n_0 \rceil$ where ϵ is the hyperparameter for $\mathcal{F}^{\epsilon}_{\texttt{PSLeftJoin}}$.

Output: Secret shares of the join $\mathcal{DB}_0 \overset{\text{dd}}{\bowtie} \mathcal{DB}_1$.

1: For $l \in [m]$, P_0 and P_1 jointly compute the left-join on the l-th key:
$$\underbrace{(\pi_l, [\![A_{0,l} \bowtie A_{1,l}]\!]_0)}_{\text{to } P_0}, \underbrace{[\![A_{0,l} \bowtie A_{1,l}]\!]_1}_{\text{to } P_1} \leftarrow \mathcal{F}^{\epsilon}_{\texttt{PSLeftJoin}}(A_{0,l}, A_{1,l})$$

2: P_σ parses the entries of the list $[\![A_{0,l} \bowtie A_{1,l}]\!]_\sigma$ as a triple
$$[\![A_{0,l} \bowtie A_{1,l}]\!]_\sigma = \{([\![w_{i,l}]\!]_\sigma, [\![z_{i,l}]\!]_\sigma, \langle \texttt{flag}_{i,l} \rangle_\sigma)\}_{i \in [\tilde{n}]}$$
Group the first field as a vector $[\![\mathbf{w}_l]\!]_\sigma = [[\![w_{0,l}]\!]_\sigma, [\![w_{1,l}]\!]_\sigma \cdots]$. Similarly, construct the second vector $[\![\mathbf{z}_l]\!]$ and the third vector $\langle \texttt{flag}_l \rangle$. Define the concat vector $\mathbf{v}_l = \mathbf{w}_l \| \mathbf{z}_l$ for the value columns.

3: For $l \in [m]$, P_0 and P_1 jointly permute the vectors
$$\langle \widetilde{\texttt{flag}_l} \rangle \leftarrow \mathcal{F}_{\texttt{Perm}}(\pi_l^{-1}, \langle \texttt{flag}_l \rangle)$$
$$[\![\tilde{\mathbf{v}}_l]\!] \leftarrow \mathcal{F}_{\texttt{Perm}}(\pi_l^{-1}, [\![\mathbf{v}_l]\!]),$$
where the permutation π_l^{-1} is input by P_0. After the permutations, P_0 and P_1 drop the last $\tilde{n} - n_0$ entries of the permuted vector.

4: P_σ initializes vectors $\langle \mathbf{c}_0 \rangle_\sigma \leftarrow \mathbf{0}^{n_0}$, $[\![\mathbf{q}_0]\!]_\sigma \leftarrow \mathbf{0}^{n_0}$

5: For $l \in [m]$, P_0 and P_1 jointly compute the first-key deduplication:
- $[\![\mathbf{q}_{l+1}]\!] \leftarrow [\![\mathbf{q}_l]\!] + \langle \neg \mathbf{c}_l \rangle \cdot [\![\tilde{\mathbf{v}}_l]\!]$ ▷ Add newly matched value column.
- $\langle \mathbf{c}_{l+1} \rangle \leftarrow \langle \mathbf{c}_l \rangle \vee \langle \widetilde{\texttt{flag}_l} \rangle$ ▷ Update bitmask.

6: P_σ outputs $[\![\mathbf{q}_m]\!]_\sigma$.

Figure 8: Multi-key inner join protocol with the first-key deduplication.

6 PRIVATE MULTI-KEY INNER-JOIN WITH FIRST-KEY DEDUPLICATION

This section formally presents the proposed multi-key inner-join protocol $\Pi_{\texttt{InnerJoin}}$, leveraging the left-join function $\mathcal{F}_{\texttt{PSLeftJoin}}$ and the private permutation function $\mathcal{F}_{\texttt{perm}}$. Specifically, we investigate the multi-key matching problem under $\overset{\text{dd}}{\bowtie}$ relation while also introducing an alternative logic: Weighted Scores Matching.

6.1 Proposed Multi-key Inner-join Protocol

Figure 8 depicts our multi-key inner-join (i.e., $\overset{\text{dd}}{\bowtie}$) protocol $\Pi_{\texttt{InnerJoin}}$ using the ideal functions $\mathcal{F}_{\texttt{PSLeftJoin}}$ and $\mathcal{F}_{\texttt{Perm}}$. This protocol basically follows the descriptions in §2.1.

(1) In Step 1, we perform column-wise left-join simultaneously using the ideal function $\mathcal{F}_{\texttt{PSLeftJoin}}$. For each $\mathcal{F}_{\texttt{PSLeftJoin}}$ execution, we generate a secretly shared `flag` column, indicating the (shuffled and padded) left-join on each key column. Also, P_0 receives a permutation π_l for each execution which is then used to unshuffle the generated left-join list.

(2) Step 2 is local computation aiming to reorder the entries of each left-join table to three vectors. That is $\mathbf{w}_l$ for the joint values from P_0, $\mathbf{z}_l$ for the joint values from P_1, and $\mathtt{flag}_l$ for the matching status. Note that these vectors are randomly shuffled by π_l. Thus at this point we still can not concatenate them cross the key column, e.g., the row $\mathbf{w}_0$ is unaligned with $\mathbf{w}_1$.
(3) In Step 3, we perform simultaneous column-wise unshuffling π_l^{-1} on the vectors $\mathbf{w}_l$, $\mathbf{z}_l$ and $\mathtt{flag}_l$ using the ideal function $\mathcal{F}_{\mathtt{Perm}}$.

That is computing the (shuffled) left-join table on each key column followed by (private) permutations to enable the first-key deduplication to obtain the final ordered left-join table defined by (1). Moreover, we adopt the common assumption that a client's (i.e., P_0) input size is smaller than a server (i.e., P_1).

The protocol's key features include:

- **Parallel**: The left-join and private permutations on each key column (i.e., Step 1 to Step 3) are computed concurrently.
- **Dimension Compression**: The $\mathcal{F}^{\epsilon}_{\mathtt{PSLeftJoin}}$ function outputs to P_0 an permutation $\pi_l\colon [\tilde{n}] \mapsto [\tilde{n}]$ where typically $1 < \tilde{n}/n_0 < 4$. According to the definition, the dummies entries are placed to the end of list before shuffling. Thus P_0 and P_1 can simply drop the final entries after finishing the inversed permutation of π.
- **Match Tracking**: The indicator vector $\mathbf{c}_l$ obliviously tracks whether a record on the P_0's side is already matched or not. This is done by via bitwise OR $\langle \mathbf{c}_{l+1} \rangle = \langle \mathbf{c}_l \rangle \vee \langle \widetilde{\mathtt{flag}_l} \rangle$ with the (permuted) membership vectors $\widetilde{\mathtt{flag}_{(l)}}$ from the left-join table on the l-th key column.
- **First-key Deduplication:** The permuted value columns $\widetilde{\mathbf{v}}_l$ from the left-join table on the l-the key is aggregated only for rows where no prior key produced a match. Formally:

$$[\![\mathbf{q}_{l+1}]\!] \leftarrow [\![\mathbf{q}_l]\!] + \underbrace{\langle \neg \mathbf{c}_l \rangle}_{\text{Unmatched record selector}} \cdot [\![\widetilde{\mathbf{v}}_l]\!].$$

  This preserves payload integrity through conditional accumulation according to the first-key deduplication logic.

Theorem 2. *The protocol in Figure 8 securely computes the inner-join (i.e., $\overset{\mathsf{dd}}{\bowtie}$ defined in (1)) against semi-honest adversaries in the $\mathcal{F}_{\mathtt{PSLeftJoin}}$-, and $\mathcal{F}_{\mathtt{Perm}}$-hybrid model.*

Proof. (Sketch.) The computation described in Figure 8 consists of two main components: (i) ideal functionality calls to $\mathcal{F}_{\mathtt{PSLeftJoin}}$ and $\mathcal{F}_{\mathtt{Perm}}$, and (ii) secure two-party computations, specifically the multiplications and logical OR operations performed in Step 5.

The security of the $\mathcal{F}_{\mathtt{PSLeftJoin}}$ and $\mathcal{F}_{\mathtt{Perm}}$ functionalities follows from their secure realizations in the hybrid model, as established by prior work. Moreover, the 2PC operations—namely multiplication and logical OR—executed over an additive secret sharing scheme are themselves secure under standard assumptions.

As a result, any adversary interacting with the protocol learns no additional information beyond what is revealed by the final output. Therefore, the protocol securely realizes the desired multi-key inner-join functionality in the semi-honest model. □

Complexity. Let m be the number of key columns, and n_0 and n_1 be the number of rows in the input datasets $\mathcal{DB}_0$ and $\mathcal{DB}_1$, respectively (assuming $n_0 \le n_1$). Let $\ell_{\mathtt{val}}$ denote the bit-length of the value columns in the databases. Our multi-key inner join protocol (Figure 8) requires m instances of $\mathcal{F}^{\epsilon}_{\mathtt{PSLeftJoin}}$, each exchanging $O(2.4\kappa n_0 + (\lambda + \log_2(n_0 n_1))n_1 + n_0\ell_{\mathtt{val}} + (1+\epsilon)n_0 E)$ bits according to the concrete implementation [36],where E is the communication complexity of our equality test protocol (Figure **??**). In addition, our protocol requires 2m instances of $\mathcal{F}_{\mathtt{perm}}$, and in total incurs a communication cost of $O(m \cdot 2\tilde{n}\log_2 \tilde{n} \cdot (\ell_{\mathtt{val}} + 1))$ bits. The key-deduplication step requires $O(10mn_0)$ bits.

6.2 Two Optimizations

We present two optimizations for the $\Pi_{\mathtt{InnerJoin}}$ protocol.

6.2.1 Lazy Masking & Correlated ANDs. Existing Circuit PSI protocols typically realize $[\![\mathbf{v}_l]\!]$ (i.e., the value columns in $\mathcal{F}_{\mathtt{PSLeftJoin}}$'s output) through two sequential operations:

(1) Generating shares $[\![\mathbf{r}_l]\!]$ where $r_{i,l}$ is random for non-matches and $r_{i,l} = v_{l,i}$ otherwise.
(2) Masking via $v_{i,l} = r_{i,l} \cdot \mathtt{flag}_{i,l}$ using the matching flag.

Also, we have one more arithmetic-Boolean multiplication in Step 5 in Figure 8 to acheive the payload accumulation. We show how to save one of the multiplication. Specifically, we first compute the bitwise ANDs: $[\![\mathbf{q}_{l+1}]\!] \leftarrow [\![\mathbf{q}_l]\!] + \left(\langle \neg \mathbf{c}_l \rangle \wedge \langle \widetilde{\mathtt{flag}_l} \rangle\right) \cdot [\![\pi_l^{-1}(\mathbf{r}_l)]\!]$

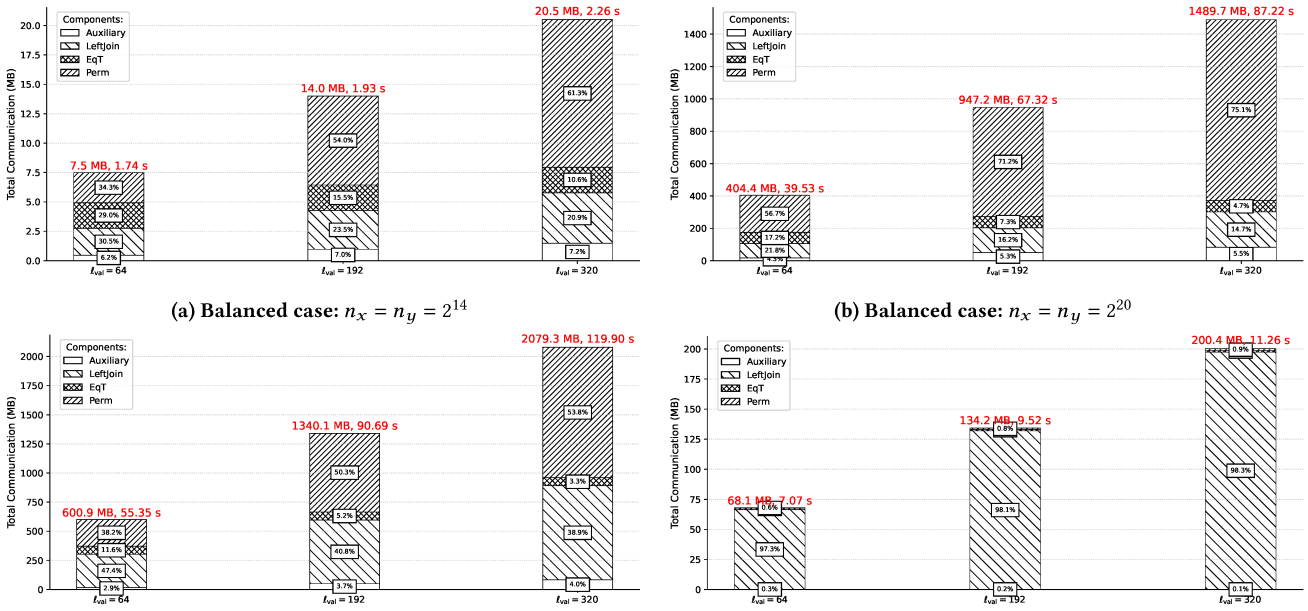
This leads to another optimization by merging the two bit-wise ANDs, i.e., $\langle \neg \mathbf{c}_l \rangle \wedge \langle \widetilde{\mathtt{flag}_l} \rangle$ and $\langle \neg \mathbf{c}_l \rangle \wedge \langle \widetilde{\neg \mathtt{flag}_l} \rangle$ (reformed from the bitwise OR in Step 5) into a single correlated AND operation, as both share the common left operand $\neg \mathbf{c}_l$. By leveraging Ferret OT's efficient ROT, this reform reduces communication costs by 30% - from 5 bits per AND gate to 7 bits for two correlated AND gates (effective rate of 3.5 bits per AND).

6.2.2 One Free Permutation. The protocol originally requires m permutations $(\pi_1^{-1}, \ldots, \pi_m^{-1})$ to align outputs from $\mathcal{F}_{\mathtt{PSLeftJoin}}$ executions. However, the absolute row order in the joined table is irrelevant - only alignment of matched rows matters. We can optimize by fixing alignment to the first key's order. That is for the keys $k_1, \ldots, k_{m-1}$, we define composite permutations $\sigma_j = \pi_1 \circ \pi_j^{-1} \quad \forall j \in [m]/\{0\}$. Then we can align outputs using the permutation $\sigma_j\colon \hat{\mathbf{v}}_j = \sigma_l(\mathbf{v}_j) = \pi_1\left(\pi_j^{-1}(\mathbf{v}_j)\right)$. The orders of $\hat{\mathbf{v}}_1, \cdots, \hat{\mathbf{v}}_{m-1}$ all match to the order of $\mathbf{v}_0$.

On the good side, we eliminate need for π_0^{-1} computation, reducing the number of permutations from m to $m-1$. However, the shared value columns remain at expanded size $\tilde{n}$ instead of original n_0 which increases the downstream query on the joint table. The guideline to use this optimization when the downstream computation on the joined table are simple (e.g., element-wise operations) or the cost of the private permutations $\Pi_{\mathtt{perm}}$ dominates computation costs.

7 EVALUATION

Experimental Setup. All computational cost measurements for our protocols are in terms of total wall-clock runtime for both parties, running on a cloud instance with an Intel(R) Xeon(R) Platinum 8457C CPU (2.60GHz) and 64GB of RAM. Additional platform



(a) Balanced case: $n_x = n_y = 2^{14}$

(b) Balanced case: $n_x = n_y = 2^{20}$

(c) Imbalanced case: $n_x = 2^{20}, n_y = 2^{22}$

(d) Highly imbalanced case: $n_x = 2^{10}, n_y = 2^{20}$

Figure 9: Performance breakdown (per-key) of the proposed multi-key inner-join protocol. The timings were measured in the LAN setting with a single thread. ℓ is the payload width on the $\mathcal{S}$'s side. The Raghuraman et al.'s [36] protocol were used as the underlying Circuit PSI protocol. For the private equality test, we fixed the key width $\ell_{\text{key}} = 64$.

details include: Operating System (Debian 12.2.0), C++ Compiler (GCC 12.2.0), and Java SDK (Amazon Corretto 17.0.14). We consider two network conditions, including **LAN** (1Gbps, 2.0ms ping), and **WAN** (100Mbps, 20ms ping).

Implementation Details. Our C++ implementation builds on the *SecretFlow* framework[3], utilizing the Raghuraman et al.'s Circuit PSI protocol [36] (with $\epsilon = 0.27$) and Ferret OT extension [42].

Evaluation Metrics. We measured the end-to-end running time, including pre-processing time (e.g., the Base OT in Ferret OT) and evaluation time. The Communication cost includes the inbound and out-bound traffic of the two parties.

Baseline Comparisons. We evaluate protocol efficiency against state-of-the-art implementations across two critical components. For the equality protocols, we benchmarked against Yao's Garbled Circuit (GC) implementation from *emp-toolkit*[4] (C++) and CGS22 [7] implementation from *SecretFlow* (C++).

For the permutation protocols, we compared against three leading approaches: Chase et al.'s [8], Garimella et al.'s [15], and Peceny et al.'s [33] implemented via Java from the *mpc4j* framework[5]. All Java baselines use *mpc4j*'s Ferret OT (13.4s, 261.02MB for $n = 2^{24}$ Δ-OTs). Our Ferret's OT implementation (C++) took 4.52s and 258.50MB for the equivalent workload. Runtime comparisons (Java vs. C++) reflect implementation efficiency differences, while communication metrics directly compare protocol designs. Our analysis isolates algorithmic improvements from low-level optimizations through normalized bandwidth measurements.

[3]https://github.com/secretflow
[4]https://github.com/emp-toolkit
[5]https://github.com/alibaba-edu/mpc4j

7.1 Multi-key Inner-join Benchmark Analysis

Let the databases $\mathcal{DB}_0$ and $\mathcal{DB}_1$ both consists of m key columns and one value columns. We write the ℓ_{key} to represent the bitwidth for the keys and ℓ_{val} for the values. Also, the number of records of these two databases are n_0 and n_1 respectively.

Performance on Synthetic Data. We evaluate the performance of our inner join protocol $\Pi_{\texttt{InnerJoin}}$ across four configurations: Two balanced cases: $n_0 = n_1 \in \{2^{14}, 2^{20}\}$, and two imbalanced cases: $(n_0, n_1) \in \{(2^{20}, 2^{22}), (2^{10}, 2^{20})\}$. We consider three bitwidths for the value columns $\ell_{\text{val}} \in \{64, 192, 320\}$ while fix $\ell_{\text{key}} = 64$.

Figure 9 decomposes communication costs into four components: Permutation (Perm), Equality testing (EqT), Single-key left-join, and Auxiliary operations (e.g., bit masking and payload updates). Results represent **per-key** costs with the lazy masking optimization (§ 6.2.1). Multi-key scenarios ($m > 1$) exhibit linear cost scaling with m. Key insights include:

- Our multi-key inner-join protocol proves to be practical for handling large-scale datasets. The complexity of our protocol grows linearly with the sizes of the databases and the width of the associated values. Even when using a single CPU core, each per-key matching operation can be completed within a few minutes. Notably the matching processes for different keys can be executed in parallel. When compared to the plain single-key inner-join, our multi-key inner-join protocol incurs an increase in cost (per-key) by approximately a factor of 1.5× – 4× for balanced configurations.

- The permutation part can take up to 75% of the overall communication. In other words, our optimized permutation protocol that

Table 2: Benchmarks on private permutation protocols. Input bits vector x for $|x| = n$ and $x_i \in \mathbb{B}^\ell$. The output vector is computed as Boolean shares. Results were averaged across 5 runs, and a single thread was used.

n	Method	Time(s) ($\ell_{val} = 8$)			Time(s) ($\ell_{val} = 64$)		
		(MB)	LAN	WAN	(MB)	LAN	WAN
2^{14}	[15]	1.78	5.00	5.03	4.84	4.74	4.77
	[8]	15.52	5.71	6.91	16.51	5.77	7.16
	[33]	5.91	10.32	10.30	6.02	9.67	10.32
	Ours	0.30	0.14	0.17	1.89	0.16	0.30
2^{17}	[15]	5.82	5.40	6.49	35.58	6.65	8.97
	[8]	147.83	18.23	28.81	157.46	17.72	30.10
	[33]	41.93	74.06	78.01	42.80	74.22	80.15
	Ours	2.55	0.56	0.73	17.86	0.69	2.05
2^{20}	[15]	46.02	29.08	30.71	326.02	38.07	58.13
	[8]	1173.35	118.15	196.18	1250.35	117.49	202.81
	[33]	330.06	593.25	626.16	337.06	607.93	622.12
	Ours	23.50	4.30	6.05	167.00	5.77	18.41

halves the communication of the current permutation [15], contributes about 20% – 40% communication savings for the balanced cases.

- Highly asymmetric cases ($n_0 \ll n_1$) transfer bottlenecks to the left-join computation (i.e., Circuit PSI). Even using the optimized Circuit PSI protocol for this unbalance case such as [20, 38], the left-join part is still the bottleneck in this case.

Performance on the Real Dataset. We evaluate our protocol using the North Carolina Voter Register (NCVR) dataset [1], a publicly available dataset that tracks active and inactive voters over time through periodic snapshots of voter registration records.

Each record in the dataset contains five fields: NCID (unique identifier), first name (25 bytes), last name (25 bytes), age (3 bytes), and birth place (3 bytes). We treat the NCID as the primary key and the remaining four fields as value columns. For evaluation, we use two snapshots from the dataset: one taken on `2013-03-26` as $\mathcal{DB}_0$, and another on `2014-05-06` as $\mathcal{DB}_1$. Due to memory constraints, we use subsets of each database containing 5×10^6 records, with each value vector having a total bit-length of $\ell_{val} = 448$.

Our protocol computes the single-key inner-join between these two databases with deduplication enabled. While deduplication has no effect in the single-key setting, we include it to demonstrate compatibility with multi-key configurations.

The results are as follows: **682 seconds** runtime using 2 CPU cores in a LAN setting, and **10.42 GB** total communication, of which approximately 75% was attributed to the private permutation component.

Comparison with Industry Solutions. We examine two private query solutions from industry: one from Google [25] and another from Meta [4], both can be used to solve the targeting the $\mathcal{F}_{\texttt{Query}}$ problem.

Unlike our left-join-to-inner-join conversion approach, these solutions first compute a shared set of encrypted primary keys for the two parties' databases—without revealing the actual keys to either party. This allows them to perform a single-key inner-join over the shared encrypted keys.

The approach in [25] does not eliminate duplicate entries in the result, whereas [4] includes deduplication at the cost of revealing additional information, such as the size of the intersection and certain one-to-many relationships.

Even aside from these privacy limitations, both approaches suffer from poor efficiency:

- The solution in [25] reports a runtime of 6.7 minutes (on 32 vCPUs) for datasets of size $n_0 = n_1 = 2^{14}$ with $m = 4$ keys. In contrast, our approach can complete the same task in just 7 seconds using a single CPU core.

- The solution in [4] reports a runtime of 299 seconds (on 48 vCPUs) for larger datasets of size $n_0 = n_1 = 10^6$ with $m = 5$ keys. Our approach can complete the same task in 200 seconds using only a single CPU core.

Importantly, the reported costs for both [25] and [4] exclude the time required to actually compute the inner-join table; they reflect only the cost of establishing the shared encrypted primary key. In contrast, our approach directly computes the secret-shared inner-join table, providing both better performance and a more complete solution to the private query problem.

Table 3: More benchmarks on the proposed permutation protocol. Results were averaged across 5 runs on the LAN setting, and a single thread was used.

n	$\ell_{val} = 1$	$\ell_{val} = 8$	$\ell_{val} = 32$	$\ell_{val} = 64$	$\ell_{val} = 128$
2^{14}	0.1055MB 0.128s	0.3037MB 0.140s	0.9834MB 0.141s	1.8897MB 0.164s	3.7021MB 0.171s
2^{17}	0.633MB 0.559s	2.547MB 0.561s	9.109MB 0.612s	17.859MB 0.694s	35.359MB 0.844s
2^{20}	5.559MB 4.158s	23.5MB 4.304s	85.0MB 4.898s	167.0MB 5.773s	331.0MB 7.030s
2^{22}	24.34MB 18.115s	103.1MB 18.969s	373.1MB 21.135s	733.1MB 24.763s	1453.1MB 30.510s

7.2 Mircro Benchmarks

7.2.1 Private Permutation Benchmarks. As shown in Table 2, our permutation protocol achieves 50% lower communication costs compared to [15] and demonstrates significantly better computational efficiency (36× for a small ℓ) than [8]. While [33] achieves linear communication complexity that theoretically surpasses our protocol, its computational overhead remains one order of magnitude higher than ours. Notably, this approach **cannot** directly process arithmetic secret shares, limiting its applicability compared to our solution. Due to *mpc4j*'s minimum supported bitwidth ($\ell \geq 8$), we benchmark our permutation protocol's bit-vector ($\ell = 1$) performance separately in Table 3.

The key insight from these results is that our OT-based permutation protocol provides an optimal balance between communication efficiency and computational performance. Furthermore, it offers greater flexibility than existing approaches through native support for both Boolean and arithmetic secret shares.

7.2.2 Private Equality Benchmarks. Table 4 compares our constant-round equality protocol with four state-of-the-art approaches. While Yao's GC achieves the lowest communication rounds (2 rounds), it incurs substantially higher communication costs compared to other OT-based alternatives. The CGS22 protocol [7] demonstrates optimal communication efficiency (50 MB for $n = 2^{20}, \ell = 64$) due

Table 4: Benchmarks on private equality protocols. Input vectors $\mathbf{x}, \mathbf{y} \in \mathbb{Z}_{2^\ell}^n$ for fixed $n = 2^{20}$. For Chandran et al.'s [7], we fix the block size $t = 4$ as suggested in their work. Results were averaged across 5 runs and a single thread was used.

ℓ_{key}	Approach	Rounds	Commu. (MB)	Time(s) LAN	Time(s) WAN
64	Yao GC [43]	2	3136	27.97	265.03
	Chandran et al. [7]	10	50.04	13.56	14.56
	Lu et al. [27](w IKNP OT)	6	1387*	99*	-
	Ours ($t = 3$)	6	53.91	13.68	15.32
96	Yao GC [43]	2	4703	41.89	397.41
	Chandran et al. [7]	12	75.36	20.90	22.47
	Ours ($t = 4$)	6	86.18	20.24	22.19

*Estimated by interpolation.

to optimized bitwise AND operations enabled by Ferret OT [8]. Notably, our protocol achieves comparable communication efficiency (8%–15% higher than CGS22) while maintaining constant rounds (6 rounds). This is particularly critical because equality operations contribute less than 10% of the overall multi-key CPSI communication but dominate the round complexity, making round efficiency a primary optimization target for the multi-key CPSI applications.

The recent constant-round equality test protocol from [27] specializes in precomputation scenarios where parties can establish correlated randomness beforehand. While minimizing evaluation-phase costs, this comes at the expense of substantially increased precomputation overhead. Specifically, they report 0.95 seconds total runtime and 13.23 MB total communication overhead ($n = 10^4, \ell = 64$), whereas our protocol can process the same inputs in 0.13 seconds with 0.52 MB bandwidth.

8 MORE RELATED WORK

8.1 Private Permutation

8.1.1 Garimella et al. [15]. Garimella et al. *optimize* the switching network-based approach [32] via Random OT, reducing communication from $O(4\ell)$ bits per gate to $O(2\ell)$ bits per gate. The protocol achieves overall communication complexity of $O(2\ell \cdot n \log_2 n)$ bits within 3 rounds, supporting both arithmetic and Boolean shares without additional overhead.

8.1.2 Chase et al. [8]. Chase et al.'s private permutation protocol utilizes switching networks and an Oblivious Punctured Vector (OPV) data structure. Its communication efficiency surpasses GMR21 when handling long input bit widths (e.g., $\ell > 400$), making it suitable for shuffling multi-column data with shared permutations. However, the OPV computation incurs significant local CPU time. The protocol requires $O(3dn\lambda \log_2 T + (d+1)n\ell)$ bits with $d = 2\lceil \log_2 n / \log_2 T \rceil - 1$ in 3 rounds, supporting both arithmetic and Boolean shares without additional overhead.

8.1.3 Peceny et al. [33]. Peceny et al. propose a private permutation protocol with linear complexity $O(2n\lceil \ell/\kappa \rceil \kappa)$ bits, leveraging the weak PRF from [2]. While achieving linear communication, three limitations exist: 1) It supports Boolean shares only, requiring B2A conversion [11] for arithmetic shares. The most efficient B2A protocol [11] still exchange about $\ell^2/2$ bits to convert a ℓ-bit Boolean share to an arithmetic share. 2) The share width must align with multiples of security parameter κ (e.g., κ bits for single-bit inputs). 3) It demands $O(n\lceil \ell/\kappa \rceil 14.68\kappa^2)$-bit memory footprint, rendering a long local CPU time.

8.1.4 Private Permutation from AHE. Existing works such as [21, 28] have employed Additive Homomorphic Encryption (AHE) to realize the $\mathcal{F}_{\text{perm}}$ functionality. Although this approach achieves linear communication complexity, its practical efficiency is constrained by the $O(n)$ homomorphic encryption operations.

8.2 Private Equality

8.2.1 GMW-like [17] and Chandran et al. [7]. The textbook GMW-like protocol for private equality evaluation constructs a binary tree of bit-wise AND operations, resulting in *logarithmic* communication rounds. Chandran et al. [7] present optimizations by merging some of the bottom levels in the binary tree using 1-out-of-N OT. Both methods fundamentally require logarithmic round complexity.

8.2.2 Lu et al. [27]. The recent constant-round equality test protocol [27] specializes in precomputation scenarios where parties can establish correlated randomness beforehand. While minimizing evaluation-phase costs, this comes at the expense of substantially increased precomputation overhead.

CONCLUSION

We present the first practical protocol for private multi-key inner-join computation, enabling secure alignment of datasets across multiple identifiers while preserving the privacy of the intersection. By optimizing key components—including halved communication for private permutation, constant-round equality testing, and efficient first-key deduplication—our solution introduces only 1.5× to 4× overhead over the base single-key inner-join protocol.

Our implementation scales to millions of records on commodity hardware, making it suitable for real-world deployment. This work enables privacy-preserving database analysis in critical domains such as healthcare, advertising, and fraud detection, where multi-key matching is essential for improving matching rates and ensuring accurate joint analysis.

Our inner-join protocol assumes that each key column contains unique values within a given database. This uniqueness constraint ensures correct matching and prevents ambiguity during the join operation. However, it may limit applicability in scenarios where duplicate keys naturally occur, such as transaction logs or event-based datasets. We leave the extension to multi-set joins as interesting directions for future work.

REFERENCES

[1] 2025. North Carolina Voter Registry. https://www.ncsbe.gov/results-data/voter-registration-data.

[2] Navid Alamati, Guru-Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. 2024. Improved Alternating-Moduli PRFs and Post-quantum Signatures. In *CRYPTO*. 274–308.

[3] Václad E Beneš. 1964. Optimal rearrangeable multistage connecting networks. *Bell system technical journal* 43, 4 (1964), 1641–1656.

[4] Prasad Buddhavarapu, Benjamin M Case, Logan Gore, Andrew Knox, Payman Mohassel, Shubho Sengupta, Erik Taubeneck, and Min Xue. 2021. Multi-key private matching for compute. *Cryptology ePrint Archive* (2021).

[5] Ran Canetti. 2000. Security and Composition of Multiparty Cryptographic Protocols. *J. Cryptol.* 13, 1 (2000), 143–202.

[6] Ran Canetti. 2006. Security and Composition of Cryptographic Protocols: A Tutorial. Cryptology ePrint Archive, Paper 2006/465. https://eprint.iacr.org/2006/465

[7] Nishanth Chandran, Divya Gupta, and Akash Shah. 2022. Circuit-PSI With Linear Complexity via Relaxed Batch OPPRF. *Proc. Priv. Enhancing Technol.* 2022, 1 (2022), 353–372.

[8] Melissa Chase, Esha Ghosh, and Oxana Poburinnaya. 2020. Secret-Shared Shuffle. In *ASIACRYPT*, Vol. 12493. 342–372.

[9] Geoffroy Couteau. 2018. New Protocols for Secure Equality Test and Comparison. In *ACNS*, Vol. 10892. 303–320.

[10] Ivan Damgård, Matthias Fitzi, Eike Kiltz, Jesper Buus Nielsen, and Tomas Toft. 2006. Unconditionally Secure Constant-Rounds Multi-party Computation for Equality, Comparison, Bits and Exponentiation. In *TCC*, Vol. 3876. 285–304.

[11] Daniel Demmler, Thomas Schneider, and Michael Zohner. 2015. ABY - A Framework for Efficient Mixed-Protocol Secure Two-Party Computation. In *NDSS*.

[12] Michael J. Freedman, Kobbi Nissim, and Benny Pinkas. 2004. Efficient Private Matching and Set Intersection. In *EUROCRYPT*, Vol. 3027. 1–19.

[13] Ying Gao, Lin Qi, Xiang Liu, Yuanchao Luo, and Longxin Wang. 2024. Efficient Fuzzy Private Set Intersection from Fuzzy Mapping. In *ASIACRYPT (Lecture Notes in Computer Science)*, Vol. 15489. 36–68.

[14] Gayathri Garimella, Benjamin Goff, and Peihan Miao. 2024. Computation Efficient Structure-Aware PSI from Incremental Function Secret Sharing. In *CRYPTO (Lecture Notes in Computer Science)*, Vol. 14927. 309–345.

[15] Gayathri Garimella, Payman Mohassel, Mike Rosulek, Saeed Sadeghian, and Jaspal Singh. 2021. Private Set Operations from Oblivious Switching. In *PKC*. 591–617.

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

[17] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *STOC*. 218–229.

[18] Xiaojie Guo, Ye Han, Zheli Liu, Ding Wang, Yan Jia, and Jin Li. 2022. Birds of a Feather Flock Together: How Set Bias Helps to Deanonymize You via Revealed Intersection Sizes. In *USENIX Security*. 1487–1504.

[19] Kyoohyung Han, Dukjae Moon, and Yongha Son. 2022. Improved Circuit-Based PSI via Equality Preserving Compression. In *SAC*. 190–209.

[20] Meng Hao, Weiran Liu, Liqiang Peng, Hongwei Li, Cong Zhang, Hanxiao Chen, and Tianwei Zhang. 2024. Unbalanced Circuit-PSI from Oblivious Key-Value Retrieval. In *USENIX Security Symposium*.

[21] Yan Huang, David Evans, and Jonathan Katz. 2012. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols?. In *NDSS*.

[22] Mihaela Ion, Ben Kreuter, Ahmet Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, Mariana Raykova, David Shanahan, and Moti Yung. 2020. On Deploying Secure Computing: Private Intersection-Sum-with-Cardinality. In *IEEE European Symposium on Security and Privacy, EuroS&P*.

[23] Bo Jiang, Jian Du, and Qiang Yan. 2024. AnonPSI: An Anonymity Assessment Framework for PSI. In *NDSS*.

[24] Basit Khurram and Florian Kerschbaum. 2020. SFour: A Protocol for Cryptographically Secure Record Linkage at Scale. In *ICDE*. 277–288.

[25] Ben Kreuter, Sarvar Patel, and Ben Terner. 2020. Private Identity Agreement for Private Set Functionalities. In *SCN*, Vol. 12238. 172–191.

[26] Yehuda Lindell and Benny Pinkas. 2000. Privacy Preserving Data Mining. In *CRYPTO*. 36–54.

[27] Tianpei Lu, Xin Kang, Bingsheng Zhang, Zhuo Ma, Xiaoyuan Zhang, Yang Liu, Kui Ren, and Chun Chen. 2025. Efficient 2PC for Constant Round Secure Equality Testing and Comparison. USENIX Security Symposium.

[28] Wen-jie Lu, Zhicong Huang, Qizhi Zhang, Yuchen Wang, and Cheng Hong. 2023. Squirrel: A Scalable Secure Two-Party Computation Framework for Training Gradient Boosting Decision Tree. In *USENIX Security*. 6435–6451.

[29] Junming Ma, Yancheng Zheng, Jun Feng, Derun Zhao, Haoqi Wu, Wenjing Fang, Jin Tan, Chaofan Yu, Benyu Zhang, and Lei Wang. 2023. SecretFlow-SPU: A Performant and User-Friendly Framework for Privacy-Preserving Machine Learning. In *USENIX ATC*. 17–33.

[30] Catherine Meadows. 1986. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*. IEEE, 134–134.

[31] Peihan Miao, Sarvar Patel, Mariana Raykova, Karn Seth, and Moti Yung. 2020. Two-Sided Malicious Security for Private Intersection-Sum with Cardinality. In *CRYPTO*, Vol. 12172. 3–33.

[32] Payman Mohassel and Seyed Saeed Sadeghian. 2013. How to Hide Circuits in MPC an Efficient Framework for Private Function Evaluation. In *EUROCRYPT*, Vol. 7881. 557–574.

[33] Stanislav Peceny, Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. 2025. Efficient Permutation Correlations and Batched Random Access for Two-Party Computation. *PKC* (2025), To appear.

[34] Benny Pinkas, Thomas Schneider, Oleksandr Tkachenko, and Avishay Yanai. 2019. Efficient Circuit-Based PSI with Linear Communication. In *EUROCRYPT*, Vol. 11478. 122–153.

[35] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2014. Faster Private Set Intersection Based on OT Extension. In *USENIX*. 797–812.

[36] Srinivasan Raghuraman and Peter Rindal. 2022. Blazing Fast PSI from Improved OKVS and Subfield VOLE. In *CCS*. 2505–2517.

[37] Peter Rindal and Phillipp Schoppmann. 2021. VOLE-PSI: Fast OPRF and Circuit-PSI from Vector-OLE. In *EUROCRYPT*, Vol. 12697. 901–930.

[38] Yongha Son and Jinhyuck Jeong. 2023. PSI with computation or Circuit-PSI for Unbalanced Sets from Homomorphic Encryption. In *ASIA CCS*. 342–356.

[39] Aron van Baarsen and Sihang Pu. 2024. Fuzzy Private Set Intersection with Large Hyperballs. In *EUROCRYPT*, Vol. 14655. 340–369.

[40] Abraham Waksman. 1968. A Permutation Network. *J. ACM* 15, 1 (1968), 159–163.

[41] Ruidi Wei and Florian Kerschbaum. 2023. Cryptographically Secure Private Record Linkage Using Locality-Sensitive Hashing. *Proc. VLDB Endow.* 17, 2 (2023), 79–91.

[42] Kang Yang, Chenkai Weng, Xiao Lan, Jiang Zhang, and Xiao Wang. 2020. Ferret: Fast Extension for Correlated OT with Small Communication. In *CCS*. 1607–1626.

[43] Andrew Chi-Chih Yao. 1986. How to Generate and Exchange Secrets. In *FOCS*. 162–167.