

More Efficient Secret-Shared Joins with Multiplicity via Oblivious Sort Expansion

Xiaoxin Du
College of Cryptology and Cyber
Science, DISSec, Nankai University
Tianjin, China
duxiaoxin@mail.nankai.edu.cn

Xiaojie Guo
Shanghai Qi Zhi Institute
Shanghai, China
guoxiaojie@sqz.ac.cn

Pinzhi Chen
College of Cryptology and Cyber
Science, DISSec, Nankai University
Tianjin, China
chenpinzhi@mail.nankai.edu.cn

Tong Li*
College of Cryptology and Cyber
Science, DISSec, Nankai University
Tianjin, China
tongli@nankai.edu.cn

Zheli Liu
College of Cryptology and Cyber
Science, DISSec, Nankai University
Tianjin, China
liuzheli@nankai.edu.cn

Abstract

Secret-shared SQL-style join is a fundamental building block in secure collaborative data analysis. In practice, join operations frequently involve duplicate keys, giving rise to one-to-many (Join-OM) and many-to-many (Join-MM) relationships. Supporting such joins requires obviously materializing all matching row pairs. Existing protocols achieve this in two costly ways: they either perform oblivious sorting over a larger expanded input or rely on multiple aggregation trees that incur additional logarithmic rounds.

In this work, we present highly efficient protocols for Join-OM and Join-MM over secret-shared databases in the standard semi-honest setting. Our core contribution is *Oblivious Sort Expansion* (OSE), a novel constant-round protocol that may be of independent interest. Rather than obviously sorting the expanded input from scratch, we sort only the original input and use OSE to derive the sorted order after expansion. For Join-OM, OSE eliminates the redundant sorting overhead introduced by input expansion in the state-of-the-art protocol by Asharov et al. (CCS 2023). For Join-MM, we combine OSE with local linear operations to obtain an aggregation-tree-free Join-MM protocol. Experimental results show that our protocols consistently outperform prior protocols, reducing both runtime and communication costs by approximately 27% and 65% for Join-OM and Join-MM, respectively.

CCS Concepts

• **Security and privacy** → **Privacy-preserving protocols**.

Keywords

Secure multiparty computation; privacy-preserving joins; oblivious sort expansion.

ACM Reference Format:

Xiaoxin Du, Xiaojie Guo, Pinzhi Chen, Tong Li, and Zheli Liu. 2026. More Efficient Secret-Shared Joins with Multiplicity via Oblivious Sort Expansion. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 20 pages. <https://doi.org/10.1145/3830454.3846540>

1 Introduction

Secure collaborative analytics is an emerging class of applications that allow mutually distrustful organizations to jointly analyze their combined data while preserving the privacy of individual datasets. Real-world collaborative scenarios include medical studies, credit investigation, and private advertising. Secure multi-party computation (MPC) [17] provides a cryptographic foundation for this setting. Modern MPC-based collaborative analytics systems [9–12, 16, 20, 25, 30, 36, 38–40, 43] typically support SQL-style relational queries, where joins are a fundamental operation for combining sensitive datasets on common keys. Since joins often lie on the critical path of analytical workloads, efficient secure join protocols are essential for practical privacy-preserving data analytics.

A common challenge is that join keys are often non-unique, so the input tables may naturally induce one-to-many (Join-OM) or many-to-many joins (Join-MM). For example, joining a user profile table with a transaction history table on user identifier yields a Join-OM, since each user appears at most once in the profile table but may appear many times in the transaction history. By contrast, joining ad-impression logs with purchase logs on user identifier yields a Join-MM, since a user may generate multiple impressions and multiple purchases. Supporting such joins therefore requires obviously materializing all matching row pairs, without revealing the matching pattern or key multiplicities.

A straightforward way to support joins with multiplicity is to compute the Cartesian product of the two input tables and then filter matching pairs [25], which incurs an impractical $O(n^2)$ overhead for large datasets. To reduce this cost, prior systems have explored additional assumptions or optimizations, including controlled leakage [10, 11, 23, 24, 40], trusted helpers [39], workload-specific techniques [9, 16, 36], and structural restrictions [12, 41]. While these approaches improve efficiency, they may limit flexibility or weaken the security guarantees.

*Corresponding author.



Table 1: Comparison of our join protocols with [6, 8, 20]. Here, $|X| = n_x$, $|Y| = n_y$, and $|Z| = m$, and Z denotes the inner-join result of tables X and Y .

	Protocol	Oblivious Sort on input ¹	soPRF ² on input	Extra Rounds
Join-OM	[20]	$n_x + n_y$	$2n_x + n_y$	$O(\log(n_x + n_y))$
	[8]	$n_x + n_y$	0	$O(\log(n_x + n_y))$
	[6]	$2n_x + n_y$	0	$O(1)^3$
	Ours	$n_x + n_y$	0	$O(1)$
Join-MM	[20]	$n_x + n_y$	$2(n_x + n_y + m)^4$	$O(\log(n_x + n_y) + \log m)$
	[8]	$2(n_x + n_y + m)$	0	$O(\log(n_x + n_y) + \log m)$
	Ours	$n_x + n_y$	$n_x + n_y$	$O(1)$

¹ The number of oblivious-sort rounds depends only on the key bit-length. ² Shared oblivious pseudorandom function. ³ Zero for right outer join, and $O(1)$ otherwise.

⁴ $2(n_x + n_y + m)$ is an upper bound.

More recently, a growing body of work [6, 8, 20] studies join protocols in modern outsourced MPC settings [3, 12, 13, 20, 25, 26, 30, 43]. In this model, data owners secret-share their inputs to a small set of non-colluding servers, which then evaluate joins over secret-shared data. These secret-shared join protocols typically follow a common paradigm: the input tables are first merged and obliviously sorted by the join key, and additional post-sorting processing is used to materialize the join result. For oblivious joins with multiplicity, this initial sorting step is the standard baseline cost [6, 8, 12, 20, 22, 36, 41]. The main additional overhead lies in post-sorting multiplicity handling. Earlier protocols [8, 20] rely on tree-based aggregations after sorting, incurring additional $O(n)$ communication and $O(\log n)$ rounds. Asharov et al. [6] improve upon [8] by eliminating aggregation trees. Instead, they perform a single oblivious stable sort over an expanded table obtained by vertically concatenating the inputs with their replicas. However, this substantially increases the sorting input size, making oblivious sorting the dominant cost. Moreover, this approach is tailored to Join-OM and does not directly generalize to Join-MM.

In this work, we show that the extra costs introduced by both aggregation trees and sorting the expanded table can be avoided. Our key observation is that once the input has been stably sorted, the stable order of its expanded version is determined by the equal-key groups. One can therefore sort the input once, and then derive the stable order of the expanded input directly. Under secret sharing, however, the boundaries and sizes of these groups remain hidden. We introduce *Oblivious Sort Expansion* (OSE) to enable this derivation obliviously, and build on it to design new protocols for both Join-OM and Join-MM that: (i) avoid expanded sorting and (ii) eliminate aggregation trees. As a result, our protocols achieve strictly better concrete efficiency, as confirmed by Table 1. Finally, our protocols can be instantiated in any semi-honest setting with efficient secret-shared sorting and generic secure computation, and can be extended to the honest-majority malicious setting.

Contributions. Our main contributions are as follows.

- **Oblivious Sort Expansion.** We introduce *Oblivious Sort Expansion* (OSE), a novel constant-round protocol that, given a vector and the permutation that stably sorts it, computes the corresponding permutation for its replicated version,

with all inputs and outputs secret-shared. OSE relies only on local linear computation and simple secure primitives, and may be of independent interest.

- **Avoiding Expanded Sorting in Join-OM.** Using OSE, we improve the Join-OM protocol of [6] by reducing the sorting input from three table-length blocks to two, while preserving its constant-round post-sorting overhead. This strictly reduces the dominant sorting cost.
- **A First Aggregation-Tree-Free Join-MM Protocol.** We introduce a new OSE-based Join-MM protocol that eliminates tree-based aggregations without increasing the sorting input size. This removes the logarithmic-depth dependence in [8, 20] and reduces the post-sorting round complexity to $O(1)$. Our construction supports inner, left, right, and full Join-MM variants. To the best of our knowledge, this is the first aggregation-tree-free Join-MM protocol.
- **Implementation and Evaluation.** We implement our protocols and evaluate them in the semi-honest setting under both local and emulated WAN conditions. Our protocols outperform prior work across all evaluated settings. Join-OM reduces both runtime and communication by approximately 27% compared with [6]. Join-MM reduces both runtime and communication by approximately 65% compared with [8] and reduces runtime by up to 53.3% compared with [20].

2 Technique Overview

In this section, we sketch the key ideas behind our constructions. We introduce OSE and then show how it avoids expanded-input sorting in Join-OM and, together with a prefix-sum-based design, eliminates aggregation trees in our Join-MM protocol. For clarity, we use informal notation here and defer the definitions to Section 3.

2.1 Oblivious Sort Expansion

Let $\vec{k} = (k_1, k_2, \dots, k_n)$ be an input vector of length n , and let $(\vec{k} \parallel \vec{k}) = (k_1, \dots, k_n, k_1, \dots, k_n)$ denote its concatenation with itself. We consider the problem of stably sorting $(\vec{k} \parallel \vec{k})$, which serves as a core building block in our Join-OM and Join-MM protocols.

A naive solution is to sort $(\vec{k} \parallel \vec{k})$ directly, which incurs roughly twice the communication cost of sorting $\vec{k}$. Alternatively, one may first sort $\vec{k}$ and then merge the two copies using oblivious merging networks or alternating merges. However, such approaches do not preserve stability. Stability is crucial for both our constructions and the Join-OM protocol of Asharov et al. [6], as it fixes a canonical order within each equal-key group. A stable sort of $(\vec{k} \parallel \vec{k})$ must place all first-copy elements before second-copy elements within each group, while preserving the relative order within each copy.

To exploit this structure, we first compute a permutation σ that stably sorts $\vec{k}$, and then use OSE to expand σ into a permutation that stably sorts $(\vec{k} \parallel \vec{k})$. Applying σ to $\vec{k}$ yields $\vec{k}^{\text{sort}}$, in which equal keys form contiguous groups. The target order of $(\vec{k}^{\text{sort}} \parallel \vec{k}^{\text{sort}})$ is therefore determined group by group.

For example, let $\vec{k}^{\text{sort}} = (a, b_1, b_2, c)$, where each subscript indicates the occurrence index of the same key. Then the stable sorted order of $(\vec{k}^{\text{sort}} \parallel \vec{k}^{\text{sort}})$ is $(a^{(1)}, a^{(2)}, b_1^{(1)}, b_2^{(1)}, b_1^{(2)}, b_2^{(2)}, c^{(1)}, c^{(2)})$, where the superscript indicates the originating copy. We treat the

two copies as separate sequences, each indexed by $(1, 2, \dots, n)$. For each element, its destination position in the expanded order equals its original index plus a shift offset. In this example, the offsets for the first copy are $(0, 1, 1, 3)$, and for the second copy are $(1, 3, 3, 4)$. Adding these offsets to the original indices $(1, 2, 3, 4)$ yields the destination positions $(1, 3, 4, 7)$ and $(2, 5, 6, 8)$, respectively.

At a high level, OSE computes these offsets from the equal-key groups in $\vec{k}^{\text{sort}}$. An element in the first copy is shifted by the total size of all preceding key groups, while an element in the second copy is shifted by the same quantity plus the size of its own group. OSE derives these offsets using prefix sums over group sizes, and then combines the resulting destination positions with the base permutation σ to obtain the stable sorting permutation of $(\vec{k} \parallel \vec{k})$. Thus, OSE avoids sorting the expanded vector directly while preserving stability. It incurs $O(1)$ rounds and linear communication.

2.2 Join-OM and Join-MM Based on OSE

We consider secret-shared joins, where non-colluding servers compute a shared result from secret-shared input tables X and Y . Let $\vec{k}_X$ and $\vec{k}_Y$ denote their join-key columns with sizes n_x and n_y , respectively, and m be the actual Join-MM output size.

2.2.1 Join-OM. Assume that each join key appears at most once in X , whereas duplicate keys may appear in Y . In the state-of-the-art Join-OM protocol [6], the dominant cost stems from obliviously sorting a three-table expanded vector $(\vec{k}_X \parallel \vec{k}_Y \parallel \vec{k}_X)$. Using our OSE protocol, we can avoid sorting replicated data from scratch. Specifically, we first compute the permutation which stably sorts the base vector $(\vec{k}_X \parallel \vec{k}_Y)$. We then invoke OSE to expand it into the sorting permutation for the vector $(\vec{k}_X \parallel \vec{k}_Y \parallel \vec{k}_X \parallel \vec{k}_Y)$ in constant rounds. To match the exact structural requirement of Join-OM, we drop the redundant $\vec{k}_Y$ block by shifting it to the trailing end while maintaining the relative stable order of the target vector $(\vec{k}_X \parallel \vec{k}_Y \parallel \vec{k}_X)$. By fundamentally reducing the sorting input size, we strictly improve both communication and computational efficiency of Join-OM.

2.2.2 Join-MM. In Join-MM, if a key k appears $X(k)$ times in table X and $Y(k)$ times in table Y , then the join output contains $X(k) \cdot Y(k)$ rows for that key. Consequently, for table X (resp. Y), rows containing key k must be duplicated $Y(k)$ (resp. $X(k)$) times.

To realize this output obliviously, a protocol must therefore address two tasks: (i) compute, for each input row, how many times it should be duplicated, (ii) materialize these duplicates without revealing the hidden key multiplicities. A concrete example illustrating these tasks and the subsequent alignment step is provided later in Figure 4.

Existing protocols [8, 20] first obliviously sort the merged key vector $(\vec{k}_X \parallel \vec{k}_Y)$ and then rely on multiple tree-based aggregations to realize both tasks. Each aggregation requires upward and downward passes over the tree, introducing additional $O(n)$ secure multiplications and $O(\log n)$ communication rounds. Our protocol starts from the same sorted layout, but fundamentally changes the post-sorting computation. Instead of aggregation trees, we exploit the fact that linear operations over secret shares are local, and implement both duplication counting and data materialization via *prefix-sum-based propagation with cancellation* (or *prefix-sum with*

cancellation for short) over carefully constructed sequences. This removes all tree-based dependencies and reduces the post-sorting round complexity to $O(1)$.

Duplication Counts. Let σ^{base} be the stable sorting permutation of $(\vec{k}_X \parallel \vec{k}_Y)$. Using OSE, we compute the stable sorting permutation σ of the expanded key vector $(\vec{k}_X \parallel \vec{k}_Y \parallel \vec{k}_X \parallel \vec{k}_Y)$. To compute the duplication counts for rows of X , we construct an auxiliary vector $(0^{n_x} \parallel 1^{n_y} \parallel 0^{n_x} \parallel -1^{n_y})$ and apply σ to it. Because σ preserves the stable sorted order of keys, for each matched key k , the permuted vector forms a segment $(0^{X(k)} \parallel 1^{Y(k)} \parallel 0^{X(k)} \parallel -1^{Y(k)})$, where $Y(k)$ is exactly the number of times each row of X with key k must be duplicated. A local prefix sum over the permuted vector transforms this segment into $(0^{X(k)} \parallel 1, 2, \dots, Y(k) \parallel Y(k)^{X(k)} \parallel Y(k) - 1, \dots, 1, 0)$. Then, we recover the original four-block layout obviously, so that each position in the third block yields the exact duplication count of the corresponding row in X (zero if no match exists).

Data Duplication. Once the duplication counts are known, rows can be materialized by the same propagation-with-cancellation principle. We illustrate the idea on the join-key column. To do this, we first construct an auxiliary vector for the entire column: $(\vec{k}_X \parallel -\vec{k}_X \parallel 0^m)$. We then compute a permutation π that rearranges this vector so that, for each key k in X , the corresponding segment takes the form $(k \parallel 0^{Y(k)} \parallel -k)$. A local prefix sum over the permuted vector propagates k exactly $Y(k)$ times, while the final $-k$ cancels the accumulated sum and resets the computation for the next segment. Then, applying the inverse permutation π^{-1} maps these values back to their original layout, leaving the materialized duplicated values in the final m elements. The same idea is applied column-wise to each payload column of X . To compute the permutation π , a naive construction based on prior soPRF techniques [20] requires $4n_x + m$ evaluations. We instead design a refined method with data-independent preprocessing, reducing the online cost to n_x soPRF evaluations.

3 Preliminaries

In this section, we introduce the necessary preliminaries, including notation, secure multi-party computation, and the input and output of secret-shared join.

3.1 Notations

Let $\vec{v} = (v_1, v_2, \dots, v_n)$ denote a vector and $[n]$ denote the set $\{1, 2, \dots, n\}$. We sometimes write $\vec{v}[i]$ to denote its i -th element, i.e., $\vec{v}[i] = v_i$ for $i \in [n]$. A permutation σ is a bijection on a finite set, which can reorder the sequence $(1, 2, \dots, n)$ into $(\sigma(1), \sigma(2), \dots, \sigma(n))$. Let $\sigma \cdot \vec{v}$ denote applying σ to $\vec{v}$, and the output is $\vec{v}'$, such that $v'_i = v_{\sigma(i)}$. Let σ^{-1} denote the inverse of σ . It follows that applying σ^{-1} after σ restores the original vector, i.e., $\sigma^{-1} \cdot (\sigma \cdot \vec{v}) = \vec{v}$. We denote by $\vec{\sigma}$ the vector of function values of σ , i.e., $\vec{\sigma} = (\sigma_1, \sigma_2, \dots, \sigma_n)$ where $\sigma_i = \sigma(i)$ for $i \in [n]$. Given two permutations σ and ρ , we can compose them as $\rho \cdot (\sigma \cdot \vec{v}) = (\rho \cdot \vec{\sigma}) \cdot \vec{v}$. In our protocols, we often directly compute the destination of each element in a vector $\vec{v}$. We denote this permutation by σ^{inv} , where $\sigma^{\text{inv}}(i)$ specifies the destination of element v_i (i.e., $v'_{\sigma^{\text{inv}}(i)} = v_i$). Although σ^{inv} is semantically equivalent to σ^{-1} , it is obtained directly without explicitly computing σ .

Parameters: Secret-shared permutation $\llbracket \vec{\sigma} \rrbracket$ and input $\llbracket \vec{v} \rrbracket$. $\llbracket \mathcal{F}_{\text{sort}} \rrbracket$ - Stable sorting
• GenPerm: Upon receiving (GenPerm, $\llbracket \vec{v} \rrbracket$) from all parties, $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{v}$ and computes the permutation σ such that applying σ to $\vec{v}$ yields the stable sort of $\vec{v}$. It then provides each party with the secret-shared permutation $\llbracket \vec{\sigma} \rrbracket$. • ApplyPerm: Upon receiving (ApplyPerm, $\llbracket \vec{\sigma} \rrbracket$, $\llbracket \vec{v} \rrbracket$), $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{\sigma}$ and $\vec{v}$, applies σ to $\vec{v}$ to obtain $\vec{v}'$, and sends each party the secret-shared vector $\llbracket \vec{v}' \rrbracket$. • ApplyInv: Upon receiving (ApplyInv, $\llbracket \vec{\sigma} \rrbracket$, $\llbracket \vec{v} \rrbracket$), $\mathcal{F}_{\text{sort}}$ reconstructs $\vec{\sigma}$ and $\vec{v}$, applies σ^{-1} to $\vec{v}$ to obtain $\vec{v}'$, and then sends each party the secret-shared vector $\llbracket \vec{v}' \rrbracket$.

Figure 1: Stable sorting ideal functionality $\mathcal{F}_{\text{sort}}$

3.2 Secure Multi-party Computation

(N,t)-Linear Secret Sharing. An (N, t) -linear secret sharing scheme distributes a secret among N parties such that any coalition of at most t parties learns nothing about the secret, while any coalition of $t + 1$ or more parties can reconstruct it. A key feature of such schemes is linearity: additions on secrets can be carried out locally on the shares, without requiring any interaction among the parties.

We use $\llbracket a \rrbracket$ to denote a secret sharing of a value a among N parties; $\llbracket a \rrbracket^n$ denotes a length- n vector with all entries equal to $\llbracket a \rrbracket$. For a vector $\vec{a} = (a_1, \dots, a_n)$, its sharing is written as $\llbracket \vec{a} \rrbracket = (\llbracket a_1 \rrbracket, \dots, \llbracket a_n \rrbracket)$. We further use the operator $\parallel$ to denote concatenation of two secret-shared vectors. For instance, if $\llbracket \vec{a} \rrbracket = (\llbracket a_1 \rrbracket, \dots, \llbracket a_n \rrbracket)$ and $\llbracket \vec{b} \rrbracket = (\llbracket b_1 \rrbracket, \dots, \llbracket b_m \rrbracket)$, then $\llbracket \vec{a} \rrbracket \parallel \llbracket \vec{b} \rrbracket = (\llbracket a_1 \rrbracket, \dots, \llbracket a_n \rrbracket, \llbracket b_1 \rrbracket, \dots, \llbracket b_m \rrbracket)$. Given a vector $\vec{a}$, $-\vec{a}$ denotes the vector in which each element of $\vec{a}$ is replaced by its additive inverse.

Basic procedures. All procedures below are implemented via $\mathcal{F}_{\text{MPC}}$ and can be instantiated in constant rounds with linear communication overhead. $\mathcal{F}_{\text{MPC}}$ is given in Appendix D.1.

- $\llbracket c \rrbracket = \llbracket a \rrbracket \cdot \llbracket b \rrbracket$. We define the product of two secret-shared vectors of length n as the element-wise product of their corresponding shares, i.e., $\llbracket c \rrbracket = \llbracket \vec{a} \rrbracket \cdot \llbracket \vec{b} \rrbracket = (\llbracket a_1 \rrbracket \cdot \llbracket b_1 \rrbracket, \dots, \llbracket a_n \rrbracket \cdot \llbracket b_n \rrbracket)$.

- $\llbracket \vec{a}' \rrbracket = \text{PREFIXSUM}(\llbracket \vec{a} \rrbracket)$: $\llbracket a'_j \rrbracket = \sum_{i=1}^j \llbracket a_i \rrbracket$. Owing to the linearity of the secret-sharing scheme, PREFIXSUM can be computed locally on the shares without any interaction.

- $\llbracket c \rrbracket \leftarrow \text{EQUAL}(\llbracket a \rrbracket, \llbracket b \rrbracket)$ outputs $\llbracket 1 \rrbracket$ if $\llbracket a \rrbracket = \llbracket b \rrbracket$, and $\llbracket 0 \rrbracket$ otherwise. This definition extends element-wise to vectors: $\llbracket c \rrbracket \leftarrow \text{EQUAL}(\llbracket \vec{a} \rrbracket, \llbracket \vec{b} \rrbracket)$.

- $\llbracket c \rrbracket = \text{IFTHEN}(\llbracket e \rrbracket : \llbracket a \rrbracket, \llbracket b \rrbracket)$: if $\llbracket e \rrbracket = \llbracket 1 \rrbracket$, output $\llbracket c \rrbracket = \llbracket a \rrbracket$. Otherwise, output $\llbracket c \rrbracket = \llbracket b \rrbracket$. This procedure can be implemented through circuit computation as $\llbracket c \rrbracket = \llbracket e \rrbracket \cdot (\llbracket a \rrbracket - \llbracket b \rrbracket) + \llbracket b \rrbracket$ (i.e., one multiplication).

Ideal functionalities. We next define several ideal functionalities that will be employed in our protocols.

- $\mathcal{F}_{\text{random}}$ - *Random secret generation ideal functionality*. $\mathcal{F}_{\text{random}}$ provides secret sharings of any value. This functionality can be realized efficiently and without interaction using pseudorandom zero sharing [15].

- $\mathcal{F}_{\text{soPRF}}$ - *Shared oblivious PRF ideal functionality* [24]. Given a random key $\llbracket k \rrbracket$ and input $\llbracket x \rrbracket \in \mathbb{Z}_2^\ell$, the functionality outputs

Table X					Table Y				
Key	Values				Key	Values			
$\llbracket k_1 \rrbracket$	$\llbracket X_{1,1} \rrbracket$	$\dots$	$\llbracket X_{1,p_x} \rrbracket$	$\llbracket v_1 \rrbracket$	$\llbracket k_1 \rrbracket$	$\llbracket Y_{1,1} \rrbracket$	$\dots$	$\llbracket Y_{1,p_y} \rrbracket$	$\llbracket v_1 \rrbracket$
$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
$\llbracket k_{n_x} \rrbracket$	$\llbracket X_{n_x,1} \rrbracket$	$\dots$	$\llbracket X_{n_x,p_x} \rrbracket$	$\llbracket v_{n_x} \rrbracket$	$\llbracket k_{n_y} \rrbracket$	$\llbracket Y_{n_y,1} \rrbracket$	$\dots$	$\llbracket Y_{n_y,p_y} \rrbracket$	$\llbracket v_{n_y} \rrbracket$

Table Z (the result of join of table X and Y)									
Key from X	Key from Y	Values from X		Values from Y					
$\llbracket k_1^X \rrbracket$	$\llbracket k_1^Y \rrbracket$	$\llbracket Z_{1,1}^X \rrbracket$	$\dots$	$\llbracket Z_{1,p_x}^X \rrbracket$	$\llbracket Z_{1,1}^Y \rrbracket$	$\dots$	$\llbracket Z_{1,p_y}^Y \rrbracket$	$\llbracket v_1 \rrbracket$	
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$	
$\llbracket k_m^X \rrbracket$	$\llbracket k_m^Y \rrbracket$	$\llbracket Z_{m,1}^X \rrbracket$	$\dots$	$\llbracket Z_{m,p_x}^X \rrbracket$	$\llbracket Z_{m,1}^Y \rrbracket$	$\dots$	$\llbracket Z_{m,p_y}^Y \rrbracket$	$\llbracket v_m \rrbracket$	

Figure 2: The inputs and output of secret-shared join.

$\llbracket y \rrbracket \in \mathbb{Z}_2^\ell$ such that $y = \text{PRF}(k, x)$. The ideal functionality can be instantiated in constant rounds with linear communication overhead using a lightweight LowMC circuit [2] or the PRF based on the alternating moduli class of assumptions [1, 37].

- $\mathcal{F}_{\text{sort}}$ - *The sorting ideal functionality*. The ideal functionality $\mathcal{F}_{\text{sort}}$, illustrated in Figure 1, supports three commands: **GenPerm** which computes the stable sorting permutation of a secret-shared vector; **ApplyPerm** which applies the permutation to a secret-shared vector; and **ApplyInv** which applies the inverse of the permutation to a secret-shared vector.

Our protocol is compatible with any linear secret-sharing scheme that supports efficient oblivious sorting, and thus applies to both the two-party additive secret sharing [19] and three-party replicated secret sharing [4]. Instantiations of the required sorting protocols are given in [37] for the two-party setting and in [5, 14] for the three-party setting.

The GENPERM command runs in a number of rounds linear in the bit length of the input elements. Both APPLYPERM and APPLYINV execute in constant rounds with $O(n)$ communication. Hence, when the input bit length is constant (e.g., 1 or 2 bits), GENPERM also runs in $O(1)$ rounds with $O(n)$ communication.

3.3 Input and Output of Secret-Shared Joins

Figure 2 illustrates the inputs and output of our join protocols, where all entries are secret-shared. The input tables X and Y contain n_x and n_y rows, respectively, and the output table Z contains m rows. We write X^{sort} and Y^{sort} for the tables X and Y stably sorted by the join key. Throughout, we consider only equi-joins.

- **Keys**: The first column of each table stores the join key, represented as vectors $\llbracket \vec{k}_X \rrbracket$, $\llbracket \vec{k}_Y \rrbracket$, and $\llbracket \vec{k}_Z \rrbracket$. Keys may contain duplicates, and all values are assumed to be positive integers.

- **Values**: Suppose table X contributes p_x payload columns and table Y contributes p_y payload columns. The payload columns of Z are obtained by concatenating those of X and Y : the first p_x columns come from X , followed by the next p_y from Y . We denote the i -th payload column of X (resp. Y) by X_i (resp. Y_i).

- **Validity**: The last column of each table is a validity bit, equal to 1 for valid rows and 0 otherwise. Invalid rows are assumed to contain zeros. The corresponding vectors are denoted $\llbracket \vec{v}_X \rrbracket$, $\llbracket \vec{v}_Y \rrbracket$, and $\llbracket \vec{v}_Z \rrbracket$.

Protocol 1 ($(\llbracket \vec{\sigma} \rrbracket \leftarrow \text{OSE}(\llbracket \vec{k} \rrbracket, \llbracket \vec{\sigma}^{\text{base}} \rrbracket))$):

Input: Secret shares of the vector $\llbracket \vec{k} \rrbracket$ and its sorting permutation $\llbracket \vec{\sigma}^{\text{base}} \rrbracket$ of length n .

Output: Secret shares of the sorting permutation $\llbracket \vec{\sigma} \rrbracket$ for $(\llbracket \vec{k} \rrbracket \parallel \llbracket \vec{k} \rrbracket)$.

The protocol:

- (1) Send $(\text{ApplyPerm}, \llbracket \vec{\sigma}^{\text{base}} \rrbracket, \llbracket \vec{k} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{k}^{\text{sort}} \rrbracket$.
 $\triangleright$ Line 2: Encode group sizes sparsely.
- (2) $(\llbracket \vec{g}^{\text{first}} \rrbracket, \llbracket \vec{g}^{\text{last}} \rrbracket) \leftarrow \text{GROUPCOUNT}(\llbracket \vec{k}^{\text{sort}} \rrbracket)$.
 $\triangleright$ Lines 3–6: Compute destination indices.
- (3) $\llbracket \vec{f} \rrbracket \leftarrow \text{PREFIXSUM}(\llbracket \vec{g}^{\text{first}} \rrbracket)$.
- (4) $\forall i \in [n], \llbracket h_i \rrbracket = \llbracket f_i \rrbracket + i$.
- (5) $\llbracket \vec{f}' \rrbracket \leftarrow \text{PREFIXSUM}(\llbracket [0], \llbracket g_1^{\text{last}} \rrbracket, \dots, \llbracket g_{n-1}^{\text{last}} \rrbracket \rrbracket)$.
- (6) $\forall i \in [n], \llbracket h'_i \rrbracket = \llbracket f'_i \rrbracket + i$.
 $\triangleright$ Lines 7–8: Compose permutations.
- (7) $\forall i \in [n], \llbracket \sigma'_i \rrbracket = \llbracket \sigma_i^{\text{base}} \rrbracket + n$.
- (8) Send $(\text{ApplyInv}, (\llbracket \vec{h}' \rrbracket \parallel \llbracket \vec{h} \rrbracket), (\llbracket \vec{\sigma}^{\text{base}} \rrbracket \parallel \llbracket \vec{\sigma}' \rrbracket))$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{\sigma} \rrbracket$.
- (9) **Output:** $\llbracket \vec{\sigma} \rrbracket$.

$\vec{k}$	$\vec{k}^{\text{sort}}$	$\vec{g}^{\text{first}}$	$\vec{f}$	$\vec{h}$	$\vec{g}^{\text{last}}$	$\vec{f}'$	$\vec{h}'$
a	a	2	2	3	0	0	1
b	a	0	2	4	2	0	2
c	b	3	5	8	0	2	5
b	b	0	5	9	0	2	6
b	b	0	5	10	3	2	7
a	c	1	6	12	1	5	11

Figure 3: An example execution of the OSE protocol.

4 Oblivious Sort Expansion

In this section, we present *Oblivious Sort Expansion* (OSE), the core building block underlying our Join-OM and Join-MM protocols. We first describe the protocol, then analyze its correctness and cost.

We define the ideal functionality $\mathcal{F}_{\text{OSE}}$ as follows. On input a secret-shared vector $\llbracket \vec{k} \rrbracket$ together with its stable sorting permutation $\llbracket \vec{\sigma}^{\text{base}} \rrbracket$, it reconstructs them, computes the stable sorting permutation $\llbracket \vec{\sigma} \rrbracket$ of $(\llbracket \vec{k} \rrbracket \parallel \llbracket \vec{k} \rrbracket)$, and returns $\llbracket \vec{\sigma} \rrbracket$ to the parties. We present OSE in Protocol 1, together with an illustrative example in Figure 3. At a high level, OSE first applies the base permutation to obtain the sorted vector $\llbracket \vec{k}^{\text{sort}} \rrbracket = \llbracket \vec{\sigma}^{\text{base}} \rrbracket \cdot \llbracket \vec{k} \rrbracket$. It then computes the stable sorting permutation of $(\llbracket \vec{k}^{\text{sort}} \rrbracket \parallel \llbracket \vec{k}^{\text{sort}} \rrbracket)$ through the following two steps.

Step 1: Encode group sizes sparsely. We invoke GROUPCOUNT on $\llbracket \vec{k}^{\text{sort}} \rrbracket$ (Protocol 2), adapting the GROUPBY.COUNT technique of Asharov et al. [6] to compute group sizes and encode them at group boundaries. The protocol outputs two secret-shared vectors $\llbracket \vec{g}^{\text{first}} \rrbracket$ and $\llbracket \vec{g}^{\text{last}} \rrbracket$. For each maximal group of equal keys, its size is placed at the first element of the group in $\llbracket \vec{g}^{\text{first}} \rrbracket$ and at the last element of the group in $\llbracket \vec{g}^{\text{last}} \rrbracket$; all other entries are zero.

Step 2: Compute destination indices via prefix sums. We treat the two copies as separate sequences, each indexed by $(1, 2, \dots,$

Protocol 2 ($(\llbracket \vec{g}^{\text{first}} \rrbracket, \llbracket \vec{g}^{\text{last}} \rrbracket) \leftarrow \text{GROUPCOUNT}(\llbracket \vec{k}^{\text{sort}} \rrbracket)$):

Input: Secret shares of the sorted vector $\llbracket \vec{k}^{\text{sort}} \rrbracket$ of length n .

Output: Secret shares of vectors $\llbracket \vec{g}^{\text{first}} \rrbracket$ and $\llbracket \vec{g}^{\text{last}} \rrbracket$, where for each group in $\llbracket \vec{k}^{\text{sort}} \rrbracket$, the group size is encoded at the first and last positions of the group, respectively, and all other entries are zero.

The protocol:

- $\triangleright$ Lines 1–6: Generate the group-count vector $\llbracket \vec{g}^{\text{first}} \rrbracket$.
- (1) $\forall i \in [2, n]$, set $\llbracket s_i \rrbracket \leftarrow \text{EQUAL}(\llbracket k_i^{\text{sort}} \rrbracket, \llbracket k_{i-1}^{\text{sort}} \rrbracket)$, $\llbracket s_1 \rrbracket = \llbracket 0 \rrbracket$ and $\llbracket e_i \rrbracket = 1 - \llbracket s_i \rrbracket$.
- (2) Send $(\text{GenPerm}, \llbracket \vec{e} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{\rho} \rrbracket$.
- (3) $\forall i \in [n]$, $\llbracket a_i \rrbracket \leftarrow \text{IFTHEN}(\llbracket e_i \rrbracket : n - i + 1, n)$.
- (4) Send $(\text{ApplyPerm}, \llbracket \vec{\rho} \rrbracket, \llbracket \vec{a} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{b} \rrbracket$.
- (5) $\forall i \in [n]$, $\llbracket c_i \rrbracket = \llbracket b_i \rrbracket - \llbracket b_{i+1} \rrbracket$ with $\llbracket b_{n+1} \rrbracket = \llbracket 0 \rrbracket$.
- (6) Send $(\text{ApplyInv}, \llbracket \vec{\rho} \rrbracket, \llbracket \vec{c} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{g}^{\text{first}} \rrbracket$.
 $\triangleright$ Lines 7–12: Generate the group-count vector $\llbracket \vec{g}^{\text{last}} \rrbracket$.
- (7) $\forall i \in [1, n-1]$, set $\llbracket e'_i \rrbracket = \llbracket e_{i+1} \rrbracket$ and $\llbracket e'_n \rrbracket = \llbracket 1 \rrbracket$.
- (8) Send $(\text{GenPerm}, (1 - \llbracket \vec{e}' \rrbracket))$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{\rho}' \rrbracket$.
- (9) $\forall i \in [n]$, $\llbracket a'_i \rrbracket \leftarrow \text{IFTHEN}(\llbracket e'_i \rrbracket : i, n)$.
- (10) Send $(\text{ApplyPerm}, \llbracket \vec{\rho}' \rrbracket, \llbracket \vec{a}' \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{b}' \rrbracket$.
- (11) $\forall i \in [n]$, $\llbracket c'_i \rrbracket = \llbracket b'_i \rrbracket - \llbracket b'_{i-1} \rrbracket$ with $\llbracket b'_0 \rrbracket = \llbracket 0 \rrbracket$.
- (12) Send $(\text{ApplyInv}, \llbracket \vec{\rho}' \rrbracket, \llbracket \vec{c}' \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{g}^{\text{last}} \rrbracket$.
- (13) **Output:** $(\llbracket \vec{g}^{\text{first}} \rrbracket, \llbracket \vec{g}^{\text{last}} \rrbracket)$.

n). For the second copy, the required offset of each element is obtained by taking a prefix sum over $\llbracket \vec{g}^{\text{first}} \rrbracket$. For the first copy, we first right-shift $\llbracket \vec{g}^{\text{last}} \rrbracket$ by one position and then take a prefix sum. Adding the resulting offsets to the original indices yields the exact destination indices of all elements. Collectively, these destination indices define the inverse stable sorting permutation of $(\llbracket \vec{k}^{\text{sort}} \rrbracket \parallel \llbracket \vec{k}^{\text{sort}} \rrbracket)$.

Finally, we compose this permutation with the duplicated base permutation induced by $\llbracket \vec{\sigma}^{\text{base}} \rrbracket$, obtaining the desired stable sorting permutation $\llbracket \vec{\sigma} \rrbracket$ of $(\llbracket \vec{k} \rrbracket \parallel \llbracket \vec{k} \rrbracket)$.

Correctness. Let $\mathcal{G} = (G_1, \dots, G_t)$ denote the partition of $\llbracket \vec{k}^{\text{sort}} \rrbracket$ into maximal contiguous groups of equal keys, and let $|G_i|$ be the size of group G_i . Write $s_i = 1 + \sum_{j=1}^{i-1} |G_j|$ for the starting index of G_i in $\vec{k}^{\text{sort}}$. In the stable sorted order of $(\llbracket \vec{k}^{\text{sort}} \rrbracket \parallel \llbracket \vec{k}^{\text{sort}} \rrbracket)$, the two copies of each group appear consecutively as $(G_1^{(1)}, G_1^{(2)}, G_2^{(1)}, G_2^{(2)}, \dots, G_t^{(1)}, G_t^{(2)})$. Hence, the starting positions of the first and second copies of G_i in the expanded sorted vector are $t_i^{(1)} = 1 + \sum_{j=1}^{i-1} 2|G_j|$ and $t_i^{(2)} = t_i^{(1)} + |G_i|$, respectively. Therefore, the offset that must be added to every element of G_i in the first copy is $t_i^{(1)} - s_i = \sum_{j=1}^{i-1} |G_j|$, while the offset for every element of G_i in the second copy is $t_i^{(2)} - s_i = \sum_{j=1}^i |G_j|$. These are exactly the quantities computed by the protocol. The vector $\llbracket \vec{g}^{\text{first}} \rrbracket$ places $|G_i|$ at the first position of each group and 0 elsewhere, so its prefix sum assigns $\sum_{j=1}^i |G_j|$ to every element of group G_i , which is precisely the offset needed for the second copy. Similarly, after right-shifting $\llbracket \vec{g}^{\text{last}} \rrbracket$ by one position, the value $|G_i|$ is aligned with the first position of G_{i+1} . Its prefix sum therefore assigns $\sum_{j=1}^{i-1} |G_j|$ to every element of group G_i , which is exactly the offset needed for the first copy. Adding

these offsets to the original indices yields the destination indices of all elements in the two copies. These destination indices define the inverse stable sorting permutation of $(\llbracket \vec{k}^{\text{sort}} \rrbracket \parallel \llbracket \vec{k}^{\text{sort}} \rrbracket)$. Composing this permutation with the duplicated base permutation induced by $\llbracket \vec{\sigma}^{\text{base}} \rrbracket$ therefore yields the stable sorting permutation of $(\llbracket \vec{k} \rrbracket \parallel \llbracket \vec{k} \rrbracket)$, as required.

Cost analysis. OSE invokes EQUAL once, IFTHEN twice, 1-bit GENPERM twice, and APPLYPERM and APPLYINV three times each. By Section 3, these underlying subprotocols all run in constant rounds with linear communication. Hence, OSE itself runs in $O(1)$ rounds and $O(n)$ communication.

Comparison with [7]. Attrapadung et al. [7] propose a general group-wise stable sorting technique, which can serve as a building block for realizing the core step of OSE, i.e., computing the stable sorting permutation of $(\vec{k}^{\text{sort}} \parallel \vec{k}^{\text{sort}})$. Specifically, after interleaving the two copies, we construct an alternating binary source-indicator vector indicating which copy each element originates from, and the task reduces to stably sorting this indicator vector within each equal-key group. Following Protocol 2 of [7], such a realization requires one EQUAL, three IFTHENS, one 1-bit GENPERM, and two calls each to APPLYPERM and APPLYINV on vectors of length $2n$. In contrast, our OSE (lines 2-6) requires one EQUAL, two IFTHENS, two 1-bit GENPERMS, and two calls each to APPLYPERM and APPLYINV, all on vectors of length n . Based on the costs of these primitives, OSE incurs only half the computation and communication cost of the [7]-based realization for computing the stable sorting permutation of $(\vec{k}^{\text{sort}} \parallel \vec{k}^{\text{sort}})$.

Applications of OSE. When combined with *prefix-sum with cancellation*, OSE supports group-wise operations such as oblivious data duplication and within-group index computation without tree-based aggregation, thereby avoiding its $O(\log n)$ rounds and $2n$ online multiplications. Beyond joins, Appendix E extends this technique to both whole-partition and prefix-running window aggregates, supporting sum, count, and avg in both settings, as well as min and max for whole partitions.

5 Secure Protocols for Join-OM and Join-MM

In this section, we present our protocols for Join-OM and Join-MM over secret-shared tables. We begin with an overview, where Join-OM is described end-to-end and Join-MM is introduced through its high-level workflow. We then present the two main building blocks of Join-MM, combine them into the Inner Join-MM protocol, and finally discuss outer Join-MM, Self-Join, and cost analysis.

5.1 Overview

We begin with Join-OM, whose construction follows directly from OSE and can therefore be described at the overview level. We then turn to Join-MM, for which we present only the main workflow and an illustrative example here, deferring the detailed procedures and the complete protocol to the following subsections.

5.1.1 Join-OM. In the Join-OM setting, we assume that the join keys in X are unique. Asharov et al. [6] proposed a secret-shared Join-OM protocol that achieves, to the best of our knowledge, the lowest communication cost among prior works. Their protocol requires obliviously sorting the concatenated vector $(\llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket)$

Protocol 3 ($\llbracket \vec{\sigma} \rrbracket \leftarrow \text{GENPERMFORJOIN-OM}(\llbracket \vec{k}_X \rrbracket, \llbracket \vec{k}_Y \rrbracket)$):

Input: Secret shares of the key vectors $\llbracket \vec{k}_X \rrbracket$ and $\llbracket \vec{k}_Y \rrbracket$.

Output: Secret shares of the sorting permutation $\llbracket \vec{\sigma} \rrbracket$ for $(\llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket \parallel \llbracket \vec{k}_X \rrbracket)$.

The protocol:

- (1) Send $(\text{GenPerm}, (\llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket))$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{\sigma}^{\text{base}} \rrbracket$.
 - (2) $\llbracket \vec{\sigma}^{\text{exp}} \rrbracket \leftarrow \text{OSE}(\llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket, \llbracket \vec{\sigma}^{\text{base}} \rrbracket)$.
 - (3) Generate vector $\llbracket \vec{e} \rrbracket = (\llbracket 0 \rrbracket^{n_x} \parallel \llbracket 0 \rrbracket^{n_y} \parallel \llbracket 0 \rrbracket^{n_x} \parallel \llbracket 1 \rrbracket^{n_y})$.
 - (4) Send $(\text{ApplyPerm}, \llbracket \vec{\sigma}^{\text{exp}} \rrbracket, \llbracket \vec{e} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{f} \rrbracket$.
 - (5) Send $(\text{GenPerm}, \llbracket \vec{f} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{\rho} \rrbracket$.
 - (6) Send $(\text{ApplyPerm}, \llbracket \vec{\rho} \rrbracket, \llbracket \vec{\sigma}^{\text{exp}} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{\sigma}' \rrbracket$.
 - (7) **Output:** $\llbracket \vec{\sigma} \rrbracket = (\llbracket \sigma'_1 \rrbracket, \dots, \llbracket \sigma'_{2n_x+n_y} \rrbracket)$.
-

Protocol 4 ($\llbracket Z \rrbracket \leftarrow \text{RIGHTOUTERJOIN-OM}(\llbracket X \rrbracket, \llbracket Y \rrbracket)$):

Input: Secret shares of the input tables $\llbracket X \rrbracket, \llbracket Y \rrbracket$, where the join key in $\llbracket X \rrbracket$ is unique.

Output: Secret shares of the join result $\llbracket Z \rrbracket$.

The protocol:

- (1) $\llbracket \vec{\sigma} \rrbracket \leftarrow \text{GENPERMFORJOIN-OM}(\llbracket \vec{k}_X \rrbracket, \llbracket \vec{k}_Y \rrbracket)$.
 - (2) For each column $\llbracket \vec{c} \rrbracket$ of table $\llbracket X \rrbracket$, do the following:
 - (a) Construct the augmented vector $\llbracket \vec{c}^{\text{exp}} \rrbracket = \llbracket \vec{c} \rrbracket \parallel \llbracket 0 \rrbracket^{n_y} \parallel \llbracket -\vec{c} \rrbracket$.
 - (b) Send $(\text{APPLYPERM}, \llbracket \vec{\sigma} \rrbracket, \llbracket \vec{c}^{\text{exp}} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{c}' \rrbracket$.
 - (c) $\llbracket \vec{c}^{\text{ps}} \rrbracket \leftarrow \text{PREFIXSUM}(\llbracket \vec{c}' \rrbracket)$.
 - (d) Send $(\text{APPLYINV}, \llbracket \vec{\sigma} \rrbracket, \llbracket \vec{c}^{\text{ps}} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{c}^{\text{dup}} \rrbracket$.
 - (e) Append $(\llbracket c_{n_x+1}^{\text{dup}} \rrbracket, \dots, \llbracket c_{n_x+n_y}^{\text{dup}} \rrbracket)$ as a column to $\llbracket X^{\text{dup}} \rrbracket$.
 - (3) $\llbracket Z \rrbracket \leftarrow$ the concatenation of $\llbracket X^{\text{dup}} \rrbracket$ and $\llbracket Y \rrbracket$ as illustrated in Figure 2.
 - (4) **Output:** $\llbracket Z \rrbracket$.
-

$\llbracket \vec{k}_X \rrbracket$), which involves $2n_x + n_y$ sorting keys. Using OSE, we instead compute the sorting permutation of $(\llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket)$ and derive the permutation required for Join-OM in constant rounds, as formalized in GENPERMFORJOIN-OM (Protocol 3). Consequently, GENPERMFORJOIN-OM performs oblivious sorting over only $n_x + n_y$ keys, reducing the sorting cost by n_x compared to [6].

For completeness, we provide the right outer Join-OM protocol in Protocol 4. We replace the direct invocation of GENPERM through $\mathcal{F}_{\text{sort}}$ used in the protocol of [6], which computes the sorting permutation of $(\llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket \parallel \llbracket \vec{k}_X \rrbracket)$, with our customized procedure GENPERMFORJOIN-OM. The remainder of the protocol remains identical to that in [6]. As in [6], Protocol 4 can be extended to support inner joins, left outer joins, and full outer joins.

5.1.2 Join-MM. We now present the detailed protocol for Join-MM. The protocol proceeds in four stages: (1) merge the input tables and apply an oblivious stable sort to the merged table by the join key; (2) compute duplication counts for all rows; (3) materialize the duplicated tables; and (4) align the duplicates so that all valid cross-pairs are formed. Figure 4 illustrates Stages 2–4 on $\llbracket X^{\text{sort}} \rrbracket$ and $\llbracket Y^{\text{sort}} \rrbracket$, which are obtained from the merged sorted layout

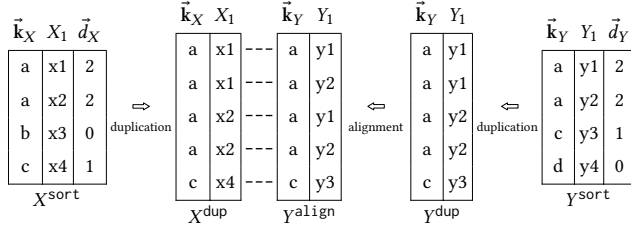


Figure 4: Example of Join-MM

produced in Stage 1. First, the protocol computes the duplication-count vectors $\vec{d}_X$ and $\vec{d}_Y$ for $[X^{\text{sort}}]$ and $[Y^{\text{sort}}]$, respectively. For example, if a row in $[X]$ with key $[a]$ has two matching rows in $[Y]$, then its duplication count in $[\vec{d}_X]$ is $[2]$, whereas an unmatched key such as $[b]$ receives a duplication count of $[0]$. Next, an oblivious data-duplication step materializes the duplicated tables $[X^{\text{dup}}]$ and $[Y^{\text{dup}}]$. Because this step duplicates rows consecutively, directly concatenating the two duplicated tables would not produce all required cross-pairs. Therefore, the protocol performs a final oblivious alignment step that permutes $[Y^{\text{dup}}]$ into an interleaved layout $[Y^{\text{align}}]$. A row-wise concatenation of $[X^{\text{dup}}]$ and $[Y^{\text{align}}]$ then yields the exact Join-MM result.

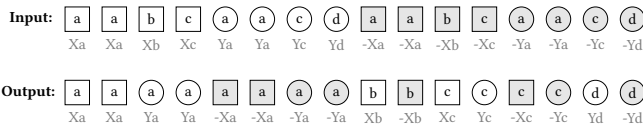
5.2 Computing Duplication Counts

Protocol 5 presents **DUPCOUNT**, which computes duplication counts for each row of $[X]$ and $[Y]$. Specifically, for each row $x \in X$, the duplication count equals the number of rows in Y that share the same join key, and the count for each row in Y is defined analogously. **DUPCOUNT** takes as input the join-key columns $[\vec{k}_X]$ and $[\vec{k}_Y]$, and a sort permutation $[\vec{\sigma}^{\text{base}}]$ for $([\vec{k}_X] \parallel [\vec{k}_Y])$. **It outputs the following values:**

- $[\vec{\sigma}_X]$ and $[\vec{\sigma}_Y]$: secret shares of the sorting permutations for $[\vec{k}_X]$ and $[\vec{k}_Y]$, respectively;
- $[\vec{d}_X]$ and $[\vec{d}_Y]$: secret shares of the duplication-count vectors for $[X^{\text{sort}}]$ and $[Y^{\text{sort}}]$, respectively. Each entry specifies the number of times the corresponding row in $[X^{\text{sort}}]$ and $[Y^{\text{sort}}]$ should be duplicated, respectively;
- $[\vec{a}_Y]$: secret shares of the within-group index for the rows of $[Y^{\text{sort}}]$. We output $[\vec{a}_Y]$ since the within-group indices of $[Y^{\text{sort}}]$ are required in subsequent protocols.

An execution of Protocol 5. To illustrate the procedure, we use the same example in Figure 4.

Step 1: Generate the sorting permutation $[\vec{\sigma}]$ over the secret-shared join key vector $([\vec{k}_X] \parallel [\vec{k}_Y] \parallel [\vec{k}_X] \parallel [\vec{k}_Y])$ using OSE. For clarity, although this reordering is not explicitly performed in the protocol, we show below the input sequence and the resulting order obtained by applying $[\vec{\sigma}]$, in order to demonstrate the effect of the permutation.



Protocol 5 $(([\vec{\sigma}_X], [\vec{\sigma}_Y], [\vec{d}_X], [\vec{d}_Y], [\vec{a}_Y]) \leftarrow \text{DUPCOUNT}([\vec{k}_X], [\vec{k}_Y], [\vec{\sigma}^{\text{base}}]))$:

Input: Secret shares of the key vectors $[\vec{k}_X]$ and $[\vec{k}_Y]$.

Output: See the description in Section 5.2.

The protocol:

- (1) $[\vec{\sigma}] \leftarrow \text{OSE}([\vec{k}_X] \parallel [\vec{k}_Y], [\vec{\sigma}^{\text{base}}])$.
 ▶ **Lines 2–5: Compute duplication counts.**
- (2) Generate vectors $[\vec{f}_X], [\vec{f}_Y], [\vec{e}]$, which are defined as

$$[\vec{f}_X] = ([0]^{n_x} \parallel [1]^{n_y} \parallel [0]^{n_x} \parallel [-1]^{n_y}),$$

$$[\vec{f}_Y] = ([1]^{n_x} \parallel [0]^{n_y} \parallel [-1]^{n_x} \parallel [0]^{n_y}),$$

$$[\vec{e}] = ([00]^{n_x} \parallel [01]^{n_y} \parallel [10]^{n_x} \parallel [11]^{n_y}).$$
- (3) Send the commands (ApplyPerm, $[\vec{\sigma}], [\vec{v}]$) for $\vec{v} \in \{\vec{f}_X, \vec{f}_Y, \vec{e}\}$ to $\mathcal{F}_{\text{sort}}$, and receive the corresponding outputs $[\vec{g}_X], [\vec{g}_Y]$, and $[\vec{e}']$.
- (4) $[\vec{g}'_X] \leftarrow \text{PREFIXSUM}([\vec{g}_X])$.
- (5) $[\vec{g}'_Y] \leftarrow \text{PREFIXSUM}([\vec{g}_Y])$.
 ▶ **Lines 6–11: Extract the duplication-count vectors.**
- (6) Send (GenPerm, $[\vec{e}']$) to $\mathcal{F}_{\text{sort}}$ and receive $[\vec{p}]$.
- (7) Send (ApplyPerm, $[\vec{p}], [\vec{\sigma}]$) to $\mathcal{F}_{\text{sort}}$ and receive $[\vec{\sigma}']$.
- (8) $[\vec{\sigma}_X] \leftarrow ([\sigma'_1], \dots, [\sigma'_{n_x}])$.
- (9) $[\vec{\sigma}_Y] \leftarrow ([\sigma'_{n_x+1}] - n_x, \dots, [\sigma'_{n_x+n_y}] - n_x)$.
- (10) Send the commands (ApplyPerm, $[\vec{p}], [\vec{v}]$) for $\vec{v} \in \{\vec{g}'_X, \vec{g}'_Y\}$ to $\mathcal{F}_{\text{sort}}$, and receive the corresponding outputs $[\vec{h}_X], [\vec{h}_Y]$.
- (11) Extract the corresponding subvectors:

$$[\vec{d}_X] \leftarrow ([\vec{h}_X[n_x + n_y + 1]], \dots, [\vec{h}_X[2n_x + n_y]]),$$

$$[\vec{d}_Y] \leftarrow ([\vec{h}_Y[n_x + 1]], \dots, [\vec{h}_Y[n_x + n_y]]),$$

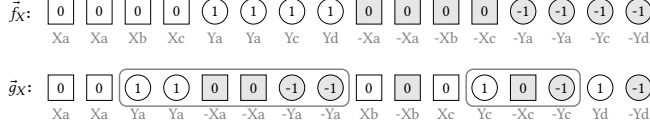
$$[\vec{a}_Y] \leftarrow ([\vec{h}_X[n_x + 1]], \dots, [\vec{h}_X[n_x + n_y]]).$$
- (12) **Output:** $([\vec{\sigma}_X], [\vec{\sigma}_Y], [\vec{d}_X], [\vec{d}_Y], [\vec{a}_Y])$.

Here, the keys enclosed in squares $\boxed{x}$ originate from X , while those enclosed in circles $\odot y$ originate from Y . Since two copies of $\vec{k}_X$ and $\vec{k}_Y$ are included, the second copies are shown with a gray background, as $\boxed{x}$ or $\odot y$. Under each element, we also display X_i or Y_i in gray, indicating that the key is i and the element belongs to table X or Y . In addition, the elements in the second copies of $\vec{k}_X$ and $\vec{k}_Y$ are distinguished by a minus sign. In the output described above, the key i from the second copy of $\vec{k}_X$ (i.e., $-X_i$) is positioned between the two copies of the matching key from $\vec{k}_Y$, namely Y_i and $-Y_i$. Similarly, the key i from the first copy of Y (i.e., Y_i) is located between the two copies of the matching key from table X , that is, X_i and $-X_i$.

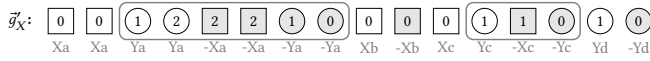
Step 2: Compute duplication counts using a prefix-sum technique. The protocol applies $[\vec{\sigma}]$ to two secret-shared vectors, $[\vec{f}_X] = ([0]^{n_x} \parallel [1]^{n_y} \parallel [0]^{n_x} \parallel [-1]^{n_y})$ and $[\vec{f}_Y] = ([1]^{n_x} \parallel [0]^{n_y} \parallel [-1]^{n_x} \parallel [0]^{n_y})$. Taking $[\vec{f}_X]$ as an example, the two $[0]^{n_x}$ blocks correspond to the two copies of X , while the $[1]^{n_y}$ and $[-1]^{n_y}$ blocks correspond to the entries of Y .

The protocol then applies the permutation $[\vec{\sigma}]$ to $[\vec{f}_X]$ and obtains $[\vec{g}_X]$. For any join key k , if it appears $X(k)$ times in X and

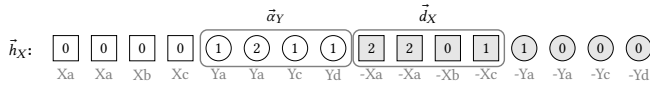
matches $Y(k)$ rows in Y , then $\llbracket \vec{g}_X \rrbracket$ contains a subsequence of the form $(\llbracket 1 \rrbracket^{Y(k)} \parallel \llbracket 0 \rrbracket^{X(k)} \parallel \llbracket -1 \rrbracket^{Y(k)})$. The resulting vector $\llbracket \vec{g}_X \rrbracket = \llbracket \sigma \cdot \vec{f}_X \rrbracket$ is shown below, where the subsequences corresponding to this case are enclosed by gray rounded rectangles.



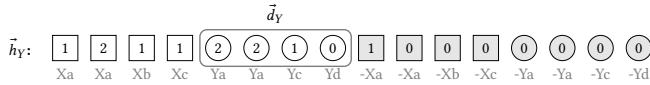
After that, the protocol computes a prefix sum over $\llbracket \vec{g}_X \rrbracket$, denoted by $\llbracket \vec{g}'_X \rrbracket$. As a result, each entry in $\llbracket \vec{g}'_X \rrbracket$ that corresponds to the second copy of X is replaced by the number of matching rows in Y , yielding the desired duplication counts.



Step 3: Extract the duplication-count vectors for $\llbracket X^{\text{sort}} \rrbracket$ and $\llbracket Y^{\text{sort}} \rrbracket$. The protocol computes a permutation $\llbracket \vec{\rho} \rrbracket$, which stably partitions the conceptual sorted order induced by $\llbracket \vec{\sigma} \rrbracket$ into four contiguous blocks ordered as $(X \parallel Y \parallel X \parallel Y)$, while preserving the relative order within each block. By composing $\llbracket \vec{\rho} \rrbracket$ and $\llbracket \vec{\sigma} \rrbracket$, we derive the sorting permutations $\llbracket \vec{\sigma}_X \rrbracket$ and $\llbracket \vec{\sigma}_Y \rrbracket$ for $\llbracket \vec{k}_X \rrbracket$ and $\llbracket \vec{k}_Y \rrbracket$, respectively. When the permutation $\llbracket \vec{\rho} \rrbracket$ is applied to $\llbracket \vec{g}'_X \rrbracket$, we obtain the resulting vector $\llbracket \vec{h}_X \rrbracket$. The contiguous subvector $(\llbracket \vec{h}_X[n_x + n_y + 1] \rrbracket, \dots, \llbracket \vec{h}_X[2n_x + n_y] \rrbracket)$ constitutes the duplication-count vector for $\llbracket X^{\text{sort}} \rrbracket$. In addition, the contiguous subvector $(\llbracket \vec{h}_X[n_x + 1] \rrbracket, \dots, \llbracket \vec{h}_X[n_x + n_y] \rrbracket)$ records, for each row of $\llbracket Y^{\text{sort}} \rrbracket$, its index within the corresponding join-key group.



Similarly, $\llbracket \vec{h}_Y \rrbracket$ can be obtained in a similar manner. The subvector $(\llbracket \vec{h}_Y[n_x + 1] \rrbracket, \dots, \llbracket \vec{h}_Y[n_x + n_y] \rrbracket)$ constitutes the duplication-count vector for $\llbracket Y^{\text{sort}} \rrbracket$.



5.3 Data Duplication

In this subsection, we describe how to perform oblivious duplication on $\llbracket k \rrbracket$ based on its duplication-count vector $\llbracket d \rrbracket$. We introduce two subprotocols, **GENDUPPERM** (Protocol 6) and **DATAUP** (Protocol 7). **GENDUPPERM** generates a duplication permutation consistent with a given duplication-count vector. Given this permutation, **DATAUP** implements oblivious data duplication. To illustrate this procedure, consider the following example:

Example 1. Let $\llbracket \vec{k} \rrbracket = (\llbracket a \rrbracket, \llbracket a \rrbracket, \llbracket b \rrbracket, \llbracket c \rrbracket)$ and $\llbracket \vec{d} \rrbracket = (\llbracket 2 \rrbracket, \llbracket 2 \rrbracket, \llbracket 0 \rrbracket, \llbracket 1 \rrbracket)$. Duplicating $\llbracket \vec{k} \rrbracket$ according to $\llbracket \vec{d} \rrbracket$ produces $\llbracket \vec{k}^{\text{dup}} \rrbracket = (\llbracket a \rrbracket, \llbracket a \rrbracket,$

Protocol 6 ($\llbracket \vec{\pi} \rrbracket \leftarrow \text{GENDUPPERM}(\llbracket \vec{d} \rrbracket)$):

Input: A length- n secret-shared duplication-count vector $\llbracket \vec{d} \rrbracket$.

Output: Secret shares of duplication permutation $\llbracket \vec{\pi} \rrbracket$.

The protocol:

- (1) $\llbracket \vec{d}' \rrbracket \leftarrow \text{PREFIXSUM}(\llbracket \vec{d} \rrbracket)$.
- (2) Open $m \leftarrow \llbracket d'_n \rrbracket$ to all parties.
- (3) For every $i \in [n]$ ($\llbracket d'_0 \rrbracket = \llbracket 0 \rrbracket$):

$$\llbracket \alpha_i \rrbracket = \llbracket d'_{i-1} \rrbracket + 2(i-1) + 1,$$

$$\llbracket \beta_i \rrbracket = \llbracket d'_i \rrbracket + 2i,$$

$$\llbracket \gamma_i \rrbracket = \llbracket d'_i \rrbracket + i.$$

► **Lines 4–10: Identify the remaining positions in $[m+2n]$ excluding $\llbracket \vec{\alpha} \rrbracket$ and $\llbracket \vec{\beta} \rrbracket$.**

- (4) $\llbracket \vec{\phi}^{\text{inv}} \rrbracket \leftarrow \text{COMPLETEPERM}(\llbracket \vec{\gamma} \rrbracket, m+n)$.
- (5) Generate vectors $\llbracket \vec{a} \rrbracket, \llbracket \vec{b} \rrbracket$, via calls to $\mathcal{F}_{\text{random}}$, which are defined as

$$\llbracket \vec{a} \rrbracket = (\llbracket 1 \rrbracket^n \parallel \llbracket 0 \rrbracket^m), \llbracket \vec{b} \rrbracket = (\llbracket 2 \rrbracket^n \parallel \llbracket 0 \rrbracket^m).$$

- (6) Send the commands $(\text{ApplyInv}, \llbracket \vec{\phi}^{\text{inv}} \rrbracket, \llbracket \vec{v} \rrbracket)$ for $\vec{v} \in \{\vec{a}, \vec{b}\}$ to $\mathcal{F}_{\text{sort}}$, and receive the corresponding outputs $\llbracket \vec{a}' \rrbracket$, and $\llbracket \vec{b}' \rrbracket$.
- (7) $\llbracket \vec{b}'' \rrbracket \leftarrow \text{PREFIXSUM}(\llbracket \vec{b}' \rrbracket)$ with $\llbracket b'_1 \rrbracket = \llbracket 1 \rrbracket$.
- (8) Send $(\text{GenPerm}, \llbracket \vec{a}' \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{\rho} \rrbracket$.
- (9) Send $(\text{ApplyPerm}, \llbracket \vec{\rho} \rrbracket, \llbracket \vec{b}'' \rrbracket)$, to $\mathcal{F}_{\text{sort}}$, and receive the corresponding outputs $\llbracket \vec{c} \rrbracket$.
- (10) Construct $\llbracket \vec{c}' \rrbracket$ of length m such that, $\forall i \in [m]$, $\llbracket c'_i \rrbracket = \llbracket c_i \rrbracket + i$.
- (11) $\llbracket \vec{\pi}^{\text{inv}} \rrbracket = (\llbracket \vec{a} \rrbracket \parallel \llbracket \vec{\beta} \rrbracket \parallel \llbracket \vec{c}' \rrbracket)$.
- (12) Send $(\text{ApplyInv}, \llbracket \vec{\pi}^{\text{inv}} \rrbracket, \llbracket [2n+m] \rrbracket)$, to $\mathcal{F}_{\text{sort}}$, and receive the corresponding output $\llbracket \vec{\pi} \rrbracket$.
- (13) **Output:** $\llbracket \vec{\pi} \rrbracket$.

$\llbracket a \rrbracket, \llbracket a \rrbracket, \llbracket c \rrbracket)$, where each $\llbracket k_i \rrbracket$ appears exactly $\llbracket d_i \rrbracket$ times in the output.

Generating Duplication Permutation. We present **GENDUPPERM** in Protocol 6, which generates a duplication permutation $\llbracket \vec{\pi} \rrbracket$ from a duplication-count vector $\llbracket \vec{d} \rrbracket$ of length n .

High-level idea. Define the augmented vector

$$\llbracket \vec{k}^{\text{aug}} \rrbracket = (\llbracket \vec{k} \rrbracket \parallel \llbracket -\vec{k} \rrbracket \parallel \llbracket 0 \rrbracket^m), \quad \text{where } m = \sum_{i=1}^n d_i.$$

The goal of **GENDUPPERM** is to construct a permutation $\llbracket \vec{\pi} \rrbracket$ such that, in the permuted vector $\llbracket \pi \cdot \vec{k}^{\text{aug}} \rrbracket$, each element $\llbracket k_i \rrbracket$ is immediately followed by exactly $\llbracket d_i \rrbracket$ copies of $\llbracket 0 \rrbracket$ and then a single copy of $\llbracket -k_i \rrbracket$.

Constructing destination indices. We first determine the destination positions of $\llbracket k_i \rrbracket$ and $\llbracket -k_i \rrbracket$. Let $\llbracket \alpha_i \rrbracket$ and $\llbracket \beta_i \rrbracket$ denote their respective destination indices, which are computed to satisfy $\llbracket \beta_i \rrbracket = \llbracket \alpha_i \rrbracket + \llbracket d_i \rrbracket + 1$. This guarantees that exactly $\llbracket d_i \rrbracket$ zero elements are placed between $\llbracket k_i \rrbracket$ and $\llbracket -k_i \rrbracket$. Next, we compute the destination positions of the m zero elements $\llbracket 0 \rrbracket^m$, which correspond to the remaining unassigned positions in $[m+2n]$ after fixing $\llbracket \vec{\alpha} \rrbracket$ and $\llbracket \vec{\beta} \rrbracket$. By concatenating the destination indices of $\llbracket \vec{k} \rrbracket$, $\llbracket -\vec{k} \rrbracket$, and $\llbracket 0 \rrbracket^m$, we obtain a vector that specifies, for each element in $\llbracket \vec{k}^{\text{aug}} \rrbracket$, its

Protocol 7 ($[\vec{k}^{\text{dup}}] \leftarrow \text{DATA_DUP}([\vec{\pi}], [\vec{k}])$):

Input: Secret shares of the duplication permutation $[\vec{\pi}]$, and the vector $[\vec{k}]$.

Output: Secret shares of the duplicated vector $[\vec{k}^{\text{dup}}]$.

The protocol:

- (1) Generate vector $[\vec{k}^{\text{aug}}] = [\vec{k}] \parallel [-\vec{k}] \parallel [0]^m$.
 - (2) Send the command (ApplyPerm, $[\vec{\pi}], [\vec{k}^{\text{aug}}]$) to $\mathcal{F}_{\text{sort}}$ and receive the corresponding output $[\vec{k}']$.
 - (3) $[\vec{k}'] \leftarrow \text{PREFIXSUM}([\vec{k}'])$.
 - (4) Send the command (ApplyInv, $[\vec{\pi}], [\vec{k}']$) to $\mathcal{F}_{\text{sort}}$ and receive the corresponding output $[\vec{k}^{\text{out}}]$.
 - (5) **Output:** $[\vec{k}^{\text{dup}}] = ([k_{2n+1}^{\text{out}}, \dots, k_{2n+m}^{\text{out}}])$.
-

destination index in the permuted vector. Consequently, this vector represents the inverse permutation of $[\vec{\pi}]$, denoted by $[\vec{\pi}^{\text{inv}}]$. For Example 1, the corresponding $\vec{k}^{\text{aug}}$ and $\vec{\pi}^{\text{inv}}$ are as follows.

$$\vec{k}^{\text{aug}}: \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline a & a & b & c & -a & -a & -b & -c & 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

$$\vec{\pi}^{\text{inv}}: \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline 1 & 5 & 9 & 11 & 4 & 8 & 10 & 13 & 2 & 3 & 6 & 7 & 12 & \\ \hline \end{array}$$

Completing a permutation vector. To compute the positions of the subvector $[0]^m$, we introduce a subprotocol COMPLETEPERM. It takes as input a secret-shared vector $[\vec{p}]$ of distinct values in $[M]$ and a public integer M , and outputs $([\vec{p}] \parallel [\vec{r}])$, where $[\vec{r}]$ contains all values in $[M]$ that do not appear in $[\vec{p}]$. Consequently, the output forms a permutation of $[M]$. The protocol details are deferred to Appendix C.1, where COMPLETEPERM makes online calls to $\mathcal{F}_{\text{soPRF}}$ on input $[\vec{p}]$. A straightforward way to compute the desired duplication permutation is to invoke COMPLETEPERM on $([\vec{\alpha}] \parallel [\vec{\beta}])$ with $M = 2n + m$. However, this approach requires an online soPRF input of size $2n$. To further reduce the online cost, Protocol 6 (Lines 4–10) halves this overhead and requires only an soPRF input of size n . The key idea is to first invoke COMPLETEPERM to generate a permutation that reorders $[0]^m \parallel [-\vec{k}]$ such that each element $[-k_i]$ is preceded by exactly d_i zero elements. This step fixes the relative ordering between the elements of $[-\vec{k}]$ and the zero elements. We then compute the exact destination indices of all zero elements in $[\vec{k}^{\text{aug}}]$ in increasing order, thereby completing the construction of the inverse duplication permutation $[\vec{\pi}^{\text{inv}}]$.

Implementing Data Duplication. We introduce DATA_DUP in Protocol 7, which duplicates the vector $[\vec{k}]$ according to the duplication permutation $[\vec{\pi}]$. We now revisit Example 1 and show how DATA_DUP produces $[\vec{k}^{\text{dup}}]$.

At a high level, DATA_DUP uses a prefix-sum technique to propagate the values in $[\vec{k}]$ into the zero entries of the augmented vector $[\vec{k}^{\text{aug}}]$, and then extracts the resulting duplicated entries. Concretely, DATA_DUP first computes $[\vec{k}'] = [\pi \cdot \vec{k}^{\text{aug}}]$.

$$\vec{k}': \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline a & 0 & 0 & -a & a & 0 & 0 & -a & b & -b & c & 0 & -c \\ \hline \end{array}$$

Protocol 8 ($[Z] \leftarrow \text{INNERJOIN-MM}([X], [Y])$):

Input: Secret shares of the input tables $[X], [Y]$.

Output: Secret shares of the join result $[Z]$.

The protocol: Each party does the following:

- (1) Send (GenPerm, $([\vec{k}_X] \parallel [\vec{k}_Y])$) to $\mathcal{F}_{\text{sort}}$ and receive $[\vec{\sigma}^{\text{base}}]$.
Lines 2–4: Compute duplication counts.
 - (2) Invoke DUPCOUNT on $([\vec{k}_X], [\vec{k}_Y], [\vec{\sigma}^{\text{base}}])$ and receive the outputs $[\vec{\sigma}_X], [\vec{\sigma}_Y], [\vec{d}_X], [\vec{d}_Y], [\vec{\alpha}_Y]$.
 - (3) For each column $[\vec{c}]$ in $[X]$ (resp. $[Y]$), send (ApplyPerm, $[\vec{\sigma}_X], [\vec{c}]$) (resp. (ApplyPerm, $[\vec{\sigma}_Y], [\vec{c}]$)) to $\mathcal{F}_{\text{sort}}$, and collect the outputs to obtain $[X^{\text{sort}}]$ (resp. $[Y^{\text{sort}}]$).
 - (4) Update $[\vec{d}_X] \leftarrow [\vec{d}_X] \cdot [\vec{v}_X^{\text{sort}}], [\vec{d}_Y] \leftarrow [\vec{d}_Y] \cdot [\vec{v}_Y^{\text{sort}}]$.
Lines 5–6: Data duplication.
 - (5) Invoke GENDUPERM on $[\vec{d}_X]$ and $[\vec{d}_Y]$ and receive the corresponding outputs $[\vec{\pi}_X]$ and $[\vec{\pi}_Y]$.
 - (6) Invoke DATA_DUP on $([\vec{\pi}_X], [\vec{c}])$ (resp. $([\vec{\pi}_Y], [\vec{c}])$) for each column $[\vec{c}]$ in $[X^{\text{sort}}]$ (resp. $[Y^{\text{sort}}]$) and collect the outputs to obtain $[X^{\text{dup}}]$ (resp. $[Y^{\text{dup}}]$).
Lines 7–8: Align table $[Y^{\text{dup}}]$.
 - (7) Invoke GENALIGNPERM on $([\vec{\alpha}_Y], [\vec{d}_X], [\vec{d}_Y], [\vec{\pi}_X], [\vec{\pi}_Y])$ and receive output $[\vec{\pi}^{\text{inv}}]$.
 - (8) Send (APPLYINV, $[\vec{\pi}^{\text{inv}}], [\vec{c}]$) to $\mathcal{F}_{\text{sort}}$ for each column $[\vec{c}]$ in $[Y^{\text{dup}}]$ and collect the outputs to obtain $[Y^{\text{align}}]$.
 - (9) Concatenate $[X^{\text{dup}}]$ and $[Y^{\text{align}}]$ column-wise as shown in Figure 2 to obtain table $[Z]$, where the validity vector $[\vec{v}_Z]$ is set to $[1]^m$.
 - (10) **Output:** Secret-shared table $[Z]$.
-

It then computes the prefix sum of $[\vec{k}']$ to obtain $[\vec{k}'']$, which obviously propagates each $[k_i]$ across $[d_i]$ consecutive positions. The negative marker $[-k_i]$ cancels the propagated value at its corresponding position, thereby delimiting each duplication segment.

$$\vec{k}'': \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline a & a & a & 0 & a & a & a & 0 & b & 0 & c & c & 0 & \\ \hline \end{array}$$

Finally, we apply the inverse permutation to restore the original order, obtaining $[\vec{k}^{\text{out}}] = [\pi^{-1} \cdot \vec{k}'']$. The subvector $([k_{2n+1}^{\text{out}}, \dots, k_{2n+m}^{\text{out}}])$ then constitutes the duplicated vector $[\vec{k}^{\text{dup}}]$ for $[\vec{k}]$.

$$\vec{k}^{\text{out}}: \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|c|c|} \hline a & a & b & c & 0 & 0 & 0 & 0 & \overbrace{a \ a \ a \ a \ c}^{\vec{k}^{\text{dup}}} & \\ \hline \end{array}$$

5.4 Inner Join-MM

We present our inner Join-MM protocol in Protocol 8, which consists of four stages:

Stage 1: Merging and Sorting Input Tables. Compute the sorting permutation of $([\vec{k}_X] \parallel [\vec{k}_Y])$.

Stage 2: Computing Duplication Counts. Compute the number of times each row in tables $[X]$ and $[Y]$ needs to be duplicated.

Protocol 9 ($\llbracket \vec{\tau}^{\text{inv}} \rrbracket \leftarrow \text{GENALIGNPERM}(\llbracket \vec{\alpha}_Y \rrbracket, \llbracket \vec{d}_X \rrbracket, \llbracket \vec{d}_Y \rrbracket, \llbracket \vec{\pi}_X \rrbracket, \llbracket \vec{\pi}_Y \rrbracket)$):

Input: Secret shares of vectors $\llbracket \vec{\alpha}_Y \rrbracket, \llbracket \vec{d}_X \rrbracket, \llbracket \vec{d}_Y \rrbracket$ and permutations $\llbracket \vec{\pi}_X \rrbracket, \llbracket \vec{\pi}_Y \rrbracket$.

Output: Secret shares of the inverse alignment permutation $\llbracket \vec{\tau}^{\text{inv}} \rrbracket$.

The protocol:

(1) Invoke $\Pi_{\text{DATA DUp}}$ three times:

$$(\llbracket \vec{\pi}_Y \rrbracket, \llbracket \vec{\alpha}_Y \rrbracket) \mapsto \llbracket \vec{\alpha}'_Y \rrbracket,$$

$$(\llbracket \vec{\pi}_Y \rrbracket, \llbracket \vec{d}_Y \rrbracket) \mapsto \llbracket \vec{d}'_Y \rrbracket,$$

$$(\llbracket \vec{\pi}_X \rrbracket, \llbracket \vec{d}_X \rrbracket) \mapsto \llbracket \vec{d}'_X \rrbracket.$$

(2) Generate vector $\llbracket \vec{\beta}_Y \rrbracket = \llbracket 0 \rrbracket^{n_y} \parallel \llbracket -\vec{d}_Y \rrbracket \parallel \llbracket 1 \rrbracket^m$.

(3) Invoke a modified version of $\Pi_{\text{DATA DUp}}$, denoted $\Pi_{\text{DATA DUp}}^*$, which is identical to $\Pi_{\text{DATA DUp}}$ except that line 1 is replaced by $\llbracket \vec{k}^{\text{aug}} \rrbracket \leftarrow \llbracket \vec{k} \rrbracket$. Use inputs $(\llbracket \vec{\pi}_Y \rrbracket, \llbracket \vec{\beta}_Y \rrbracket)$ and receive the output $\llbracket \vec{\beta}'_Y \rrbracket$.

(4) $\llbracket \vec{\tau}^{\text{inv}} \rrbracket = \llbracket [m] \rrbracket - (\llbracket \vec{d}'_Y \rrbracket - \llbracket 1 \rrbracket^m) \cdot (\llbracket \vec{\alpha}'_Y \rrbracket - \llbracket 1 \rrbracket^m) + (\llbracket \vec{d}'_X \rrbracket - \llbracket 1 \rrbracket^m) \cdot (\llbracket \vec{\beta}'_Y \rrbracket - \llbracket 1 \rrbracket^m)$.

(5) **Output:** $\llbracket \vec{\tau}^{\text{inv}} \rrbracket$.

- (1) Invoke DUPCOUNT with inputs $(\llbracket \vec{k}_X \rrbracket, \llbracket \vec{k}_Y \rrbracket, \llbracket \vec{\sigma}^{\text{base}} \rrbracket)$ and receive sorting permutations and duplication-count vectors.
- (2) Permute the tables $\llbracket X \rrbracket$ and $\llbracket Y \rrbracket$ according to the sorting permutations, so that they are ordered by $\llbracket \vec{k}_X \rrbracket$ and $\llbracket \vec{k}_Y \rrbracket$, respectively. The result tables are $\llbracket X^{\text{sort}} \rrbracket$ and $\llbracket Y^{\text{sort}} \rrbracket$.
- (3) Update the duplication-count vectors based on the validity column, setting the duplication count of invalid rows to zero.

Stage 3: Data Duplication. Each row in $\llbracket X^{\text{sort}} \rrbracket$ and $\llbracket Y^{\text{sort}} \rrbracket$ is duplicated according to its corresponding duplication count by invoking the GenDupPerm and DataDup protocols. The result tables are $\llbracket X^{\text{dup}} \rrbracket$ and $\llbracket Y^{\text{dup}} \rrbracket$.

Stage 4: Aligning Table $\llbracket Y^{\text{dup}} \rrbracket$ with $\llbracket X^{\text{dup}} \rrbracket$. To compute the alignment permutation for table $\llbracket Y^{\text{dup}} \rrbracket$, we design the subprotocol GENALIGNPERM in Protocol 9. It specifies the destination position for each row in $\llbracket Y^{\text{dup}} \rrbracket$ so that it aligns with the corresponding row in table $\llbracket X^{\text{dup}} \rrbracket$. Specifically, given the within-group index vector $\llbracket \vec{\alpha}_Y \rrbracket$ of $\llbracket Y^{\text{sort}} \rrbracket$, the duplication-count vectors $\llbracket \vec{d}_X \rrbracket$ and $\llbracket \vec{d}_Y \rrbracket$, and the duplication permutations $\llbracket \vec{\pi}_X \rrbracket$ and $\llbracket \vec{\pi}_Y \rrbracket$, the subprotocol outputs an inverse permutation $\llbracket \vec{\tau}^{\text{inv}} \rrbracket$ such that applying it to $\llbracket Y^{\text{dup}} \rrbracket$ yields the aligned table $\llbracket Y^{\text{align}} \rrbracket$. The final joined table $\llbracket Z \rrbracket$ is obtained by concatenating $\llbracket X^{\text{dup}} \rrbracket$ and $\llbracket Y^{\text{align}} \rrbracket$ column-wise.

5.5 Extensions

Our join constructions can be adapted to control output-size leakage and extended to outer joins and self-joins. We defer the detailed protocols to the appendix and summarize only the main ideas here.

Controlling Join-MM output-size leakage. In Stage 3, INNERJOIN-MM reveals the exact size m of the output table Z to all parties. Alternatively, the protocol may pad the output to either a fixed public upper bound $M \geq m$ or a differentially private upper bound $m' \geq m$ [10]. In both cases, Z is extended to the selected bound with all-zero dummy rows. Since INNERJOIN-MM does not directly

Table 2: Comparison of round complexity.

Protocols	Round Complexity
Join-OM	[20] $r_{\text{sort}} + c_{\text{eq}} + c_{\text{prf}} + 2 \log(n_x + n_y) + c_1$
	[8] $r_{\text{sort}} + c_{\text{eq}} + 2 \log(n_x + n_y) + c_2$
	[6] $r_{\text{sort}} + c_3$
	Ours $r_{\text{sort}} + c_{\text{eq}} + c_4$
Join-MM	[20] $r_{\text{sort}} + c_{\text{eq}} + c_{\text{prf}} + 2(\log(n_x + n_y) + \log m) + c_5$
	[8] $3r_{\text{sort}} + c_{\text{eq}} + 2(\log(n_x + n_y) + \log m) + c_6$
	Ours $r_{\text{sort}} + c_{\text{eq}} + c_{\text{prf}} + c_7$

$r_{\text{sort}} = r_{\text{gp}} + c_{\text{ap}} + c_{\text{ai}}$, where r_{gp} , c_{ap} , and c_{ai} denote the round complexities of GENPERM , APPLYPERM , and APPLYINV , respectively. Note that r_{gp} is $O(\ell)$ and independent of the data sizes. c_{eq} and c_{prf} represent the constant rounds for evaluating equality checks and pseudo-random functions on secret-shared inputs, respectively. The terms c_i ($1 \leq i \leq 7$) denote small constant overheads (e.g., $c_i \leq 5$).

support padded outputs, we present a modified protocol supporting both settings in Appendix C.2.

Outer Join-MM. We present the left outer join protocol in Appendix C.3. At a high level, it extends the result of INNERJOIN-MM by appending $\llbracket X \rrbracket$ to the output of the inner join and setting the validity bit of each row in $\llbracket X \rrbracket$ that matches a row in $\llbracket Y \rrbracket$ to $\llbracket 0 \rrbracket$. The right outer join and full outer join variants are obtained analogously and are therefore omitted here for brevity. The results of the outer joins are also illustrated in Appendix C.3.

Self-Join. A self-join is handled by viewing a single input table as two logical aliases and then applying the same overall framework. In particular, once the required base permutation is derived, the remaining computation follows the corresponding Join-OM or Join-MM pipeline. We defer the detailed self-join protocols to Appendix C.4.

5.6 Cost analysis

Apart from the initial oblivious sort of the $n_x + n_y$ join keys, INNERJOIN-MM requires one invocation of OSE , two soPRF invocations, seven APPLYPERM invocations, five APPLYINV invocations, and three auxiliary GENPERM calls (two on 1-bit keys and one on 2-bit keys). Since OSE , soPRF , APPLYPERM , and APPLYINV all run in constant rounds with linear communication, and GENPERM runs in constant rounds on constant-bit inputs, the entire post-sorting phase of INNERJOIN-MM runs in $O(1)$ rounds with linear communication. Table 2 summarizes the round complexity in detail.

6 Security Analysis

Due to space limitations, we present only the main security theorems and concise proof sketches in the main body. Full proofs are deferred to Appendix D. We define standalone ideal functionalities only for OSE and the top-level join protocols. The remaining protocol boxes, including GROUPCOUNT , DUPCOUNT , GENDUPPERM , COMPLETEPERM , DATA DUp and GENALIGNPERM , are auxiliary subprotocols introduced solely for readability and are treated as being inlined in the surrounding protocols.

THEOREM 1. *Protocol 1 securely realizes $\mathcal{F}_{\text{OSE}}$ in the $(\mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}})$ -hybrid model against semi-honest adversaries.*

Parameters: Secret-shared input tables $\llbracket X \rrbracket$ and $\llbracket Y \rrbracket$.

$[\mathcal{F}_{\text{JOIN-MM}}]$

- Upon receiving $(\text{JOIN-MM}, \llbracket X \rrbracket, \llbracket Y \rrbracket)$ from all parties, $\mathcal{F}_{\text{JOIN-MM}}$ reconstructs X and Y , computes their inner join result Z and its size $m = |Z|$, sends each party the secret-shared table $\llbracket Z \rrbracket$ and its length m .

Figure 5: Join-MM ideal functionality $\mathcal{F}_{\text{join-MM}}$.

PROOF SKETCH. Correctness follows from Section 4. For security, the view of each party consists of its input share, local randomness, and the shares returned to it by $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, and $\mathcal{F}_{\text{sort}}$ during the execution of Protocol 1. All remaining steps are local computations on secret shares, and no intermediate value is revealed in the clear. Given the corrupted party's input and output shares, a simulator samples uniformly random and independent shares for each invocation of $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, and $\mathcal{F}_{\text{sort}}$. By the privacy of the underlying secret-sharing scheme, the simulated view is indistinguishable from the real-world view. The full proof is given in Appendix D.2. $\square$

Let $\mathcal{F}_{\text{JOIN-MM}}$ denote the ideal functionality for many-to-many joins, as defined in Figure 5.

THEOREM 2. *Protocol INNERJOIN-MM securely realizes $\mathcal{F}_{\text{join-MM}}$ in the $(\mathcal{F}_{\text{OSE}}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}}, \mathcal{F}_{\text{soPRF}})$ -hybrid model against semi-honest adversaries.*

PROOF SKETCH. For correctness, Protocol 8 correctly realizes the semantics of the inner join except with negligible probability due to possible encoding collisions in the soPRF-based COMPLETEPERM subroutine (Protocol 10). With appropriately chosen parameters, such collisions occur with probability at most $2^{-\lambda}$.

For security, the proof follows the same overall simulation strategy as in Theorem 1. All non-local interactions in Protocol 8 are mediated through the ideal functionalities $\mathcal{F}_{\text{OSE}}$, $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, $\mathcal{F}_{\text{sort}}$, and $\mathcal{F}_{\text{soPRF}}$, while all remaining steps are local computations on secret shares. Moreover, Protocol 8 explicitly reveals the join result size m . This does not introduce additional leakage beyond the ideal functionality, since $m = |Z|$ is already determined by the output length. Moreover, conditioned on the no-collision event, the soPRF-based COMPLETEPERM subroutine can be simulated from the public input sizes and the revealed output size m . Hence, by the privacy of the underlying secret-sharing scheme and the security of the realized soPRF, the simulated view is indistinguishable from the real view. The full proof is given in Appendix D.3. $\square$

Join-OM. We defer the formal security theorem and its proof for the Join-OM protocol (Protocol 4) to Appendix D.4.

Remark. Our protocol can be extended to the malicious adversary setting assuming an honest majority. We defer the formal description of the modifications and the corresponding security proofs to Appendix D.5.

7 Experimental Results

This section presents the experimental setup and evaluation of our proposed OSE, Join-OM and Join-MM protocols, together with comparisons against prior works [6, 8, 20].

Table 3: OSE cost comparison with baseline.

n	Local Time (sec.)			Comm. (MB)		
	Base	Sort+OSE (OSE)	Red. (%)	Base	Sort+OSE (OSE)	Red. (%)
2^8	0.04	0.04 (0.01)	0	2.10	1.16 (0.12)	44.8
2^{12}	0.17	0.12 (0.01)	29.4	32.1	17.25 (1.89)	46.3
2^{16}	2.01	1.21 (0.11)	39.8	491.6	275.4 (29.58)	44.0
2^{20}	38.5	18.2 (1.67)	52.7	7866	4454 (521.3)	43.4

7.1 Experimental Setup

All protocols are implemented within the SecretFlow framework, using SPU [26, 42] as the underlying secure computation engine. Unless otherwise specified, we instantiate SPU with the three-party semi-honest ABY³ [27] framework and set the input bit-length to $\ell = 32$. We set the computational security parameter to $\kappa = 128$ and the statistical security parameter to $\lambda = 40$. All join experiments use two input tables of equal size, i.e., $n = n_x = n_y$. We consider table sizes $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}\}$, and each table consists of two columns: one key column and one payload column.

All experiments are conducted on a single workstation with 32 GB RAM and a 13th Gen Intel Core i7-13700KF CPU (16 cores, 3.4 GHz). In the default local setting, the three parties communicate over localhost with sub-millisecond latency. Unless otherwise specified, we emulate WAN conditions using Linux Traffic Control on the loopback interface, with the bandwidth limited to 1 Gbps and the one-way delay set to 20 ms. All reported performance numbers are averaged over multiple runs.

For the comparison with [20] in Tables 6 and 8, we adjust our default setup to match their baseline implementation. Specifically, we use the Semi2k framework for both implementations and evaluate with an input bit-length of $\ell = 64$, since the baseline is available only in that setting.

7.2 OSE Evaluation

Table 3 reports a microbenchmark of OSE. Here, *Base* denotes the baseline approach that directly invokes GENPERM on the expanded vector $(\vec{k} \parallel \vec{k})$, while *Sort+OSE* first computes the stable sorting permutation of $\vec{k}$ and then invokes OSE to derive that of $(\vec{k} \parallel \vec{k})$. The values in parentheses report the standalone overhead of OSE once the base sorting permutation is already available. The results show that the *Sort+OSE* approach consistently outperforms this baseline, reducing runtime by 29% – 53% and communication by about 45%. These gains come from replacing a direct sort on $2n$ elements with a sort on only n elements followed by OSE. The standalone overhead of OSE is modest. At $n = 2^{20}$, for example, OSE itself costs only 1.67 seconds and 521.3 MB, much smaller than the corresponding costs of the full *Sort+OSE* pipeline. This shows that the main benefit comes from avoiding the expanded-input sort, while OSE adds only a relatively small overhead.

7.3 Join-OM Evaluation

We implement both our Join-OM protocol and the protocol of [6] in SPU. Table 4 reports runtime in the local and emulated WAN

Table 4: Join-OM cost comparison.

n	Local Time (sec.)			WAN Time (sec.)			Comm. (MB)		
	[6]	Ours	Red. (%)	[6]	Ours	Red. (%)	[6]	Ours	Red. (%)
2^8	0.05	0.05	0	4.47	4.41	1.3	2.93	2.28	22.2
2^{12}	0.26	0.19	26.9	6.59	5.75	12.7	47.0	34.4	26.8
2^{16}	3.10	2.24	27.7	20.01	14.67	26.7	752.5	547.8	27.2
2^{20}	59.7	43.7	26.8	277.7	204.3	26.4	12040	8738	27.4

settings, as well as total communication. For $n \geq 2^{12}$, our protocol reduces local runtime and communication by around 27%. For $n \geq 2^{16}$, the runtime reductions under the WAN setting are comparable to those under the local setting, demonstrating that our optimization remains effective in high-latency environments. This gain comes from reducing the sorting input size from $3n$ to $2n$. Since oblivious sorting dominates the cost of the protocol in [6], removing one extra n -length table from the sorted input yields a substantial reduction in both runtime and communication. The observed improvement is slightly smaller than the idealized 33% reduction suggested by the near-linear scaling of oblivious sorting on SPU, because our protocol incurs an additional OSE step after sorting. This gap indicates that the extra constant-round overhead introduced by OSE is small in practice.

7.4 Join-MM Evaluation

We compare our Join-MM protocol with the protocols of [8, 20] under both local and emulated WAN settings.

We first compare with [8]. Following its original experimental setting, we set the input bit length to $\ell = 32$ and implement both protocols in SPU. The performance of our implementation of [8] is comparable to that reported in the original paper. In addition to $m = n$, we evaluate larger join outputs with $m = 2n$ and $m = 4n$. As shown in Table 5, when $m = n$, our protocol reduces both runtime and communication by around 66%. As m increases, the advantage generally becomes more pronounced, with runtime and communication reductions reaching up to around 80%, respectively. When $n = 2^{20}$ and $m = 4n$, the implementation of [8] exhausts the available memory and cannot complete, whereas our protocol completes in 93.52 s with a communication cost of 14,768 MB. These results demonstrate the improved scalability of our protocol for joins with high matching multiplicities.

We next compare with [20] using the semi-honest Join-MM implementation provided by SecretFlow-SCQL [16], which is built on the same SPU framework as ours. Since this implementation is available only for 64-bit inputs, Table 6 reports local runtimes for $\ell = 64$ under varying output sizes m .¹ For all evaluated settings with $n > 2^8$, our protocol outperforms the protocol of [20], with the improvements generally becoming more pronounced as the input and output sizes increase. At $n = 2^{20}$, the runtime reduction increases from 43.6% for $m = n$ to 53.3% for $m = 4n$, showing that our advantage grows as more matching row pairs must be materialized.

¹SecretFlow-SCQL [16] does not provide direct access to communication statistics; hence, we report only runtime.

Table 5: Join-MM cost comparison with [8].

n		Local Time (sec.)			Comm. (MB)		
		[8]	Ours	Red. (%)	[8]	Ours	Red. (%)
$m = n$	2^8	0.27	0.09	66.7	9.25	2.83	69.4
	2^{12}	1.03	0.38	63.1	135.3	45.3	66.5
	2^{16}	10.55	3.69	65.0	2171	722.0	66.7
	2^{20}	207.8	69.2	66.7	34535	11574	66.5
$m = 2n$	2^8	0.29	0.10	65.5	11.96	2.81	76.5
	2^{12}	1.23	0.39	68.3	183.5	50.42	72.5
	2^{16}	13.17	3.65	72.3	3054	820.3	73.1
	2^{20}	281.2	75.48	73.2	47194	13184	72.1
$m = 4n$	2^8	0.35	0.11	68.6	17.82	3.41	80.9
	2^{12}	1.79	0.55	69.3	259.1	54.44	79.0
	2^{16}	21.37	4.47	79.1	3320	878.7	73.5
	2^{20}	— ¹	93.52	—	—	14768	—

¹ The implementation of [8] ran out of memory at $n = 2^{20}$ and $m = 4n$.

Table 6: Join-MM local runtime comparison with [20] ($\ell = 64$), in seconds.

n	$m = n$			$m = 2n$			$m = 4n$		
	[20]	Ours	Red. (%)	[20]	Ours	Red. (%)	[20]	Ours	Red. (%)
2^8	0.12	0.12	0	0.13	0.13	0	0.16	0.13	18.8
2^{12}	0.91	0.86	5.5	1.03	0.89	13.6	1.35	0.93	31.1
2^{16}	10.76	6.07	43.6	11.8	6.33	46.4	15.19	6.65	56.2
2^{20}	186.4	105.1	43.6	208.4	114.4	45.1	271.1	126.4	53.4

To identify the source of these improvements, we further break down the runtime at $n = m = 2^{20}$. In the comparison with [8] at $\ell = 32$, the initial sort over the merged $2n$ keys takes around 38 s in both implementations, whereas the subsequent phases take approximately 170 s for the protocol of [8] and 31 s for ours, corresponding to a speedup of roughly 5.5×. In the comparison with [20] at $\ell = 64$, the initial sort takes around 64 s in both implementations, while the subsequent phases take approximately 123 s for the protocol of [20] and 41 s for ours, corresponding to a speedup of roughly 3×. The subsequent phases account for approximately 82% and 66% of the total runtimes of the protocols in [8] and [20], respectively. These breakdowns reflect the underlying protocol designs. All three protocols begin by sorting the merged $2n$ join keys, but they differ in the subsequent processing used to materialize all matching row pairs. Our protocol eliminates tree-based aggregations from duplication-count computation and data duplication, thereby avoiding the associated $O(\log n)$ rounds and $O(n)$ online multiplications. For data duplication, we further adapt the soPRF-based approach of [20] to reduce its online cost while avoiding the additional oblivious sorts required by [8]. Since the initial sorting cost is independent of m for fixed n and ℓ , the post-sorting phases

Table 7: Join-MM WAN runtime comparison with [8] under different network latencies, in seconds.

n	20ms			40ms			60ms		
	[8]	Ours	Red. (%)	[8]	Ours	Red. (%)	[8]	Ours	Red. (%)
2^8	32.05	7.80	75.7	63.53	15.29	75.9	94.63	23.73	74.9
2^{12}	42.91	11.23	73.8	85.43	22.16	74.1	122.7	33.63	72.6
2^{16}	81.10	23.47	71.1	136.4	36.65	73.1	188.0	46.75	75.1
2^{20}	813.3	283.9	65.1	1103	390.4	64.6	1406	492.7	65.0

Table 8: Join-MM WAN runtime comparison with [20] under different network latencies ($\ell = 64$), in seconds.

n	20ms			40ms			60ms		
	[20]	Ours	Red. (%)	[20]	Ours	Red. (%)	[20]	Ours	Red. (%)
2^8	18.5	12.9	30.3	36.1	25.5	29.4	57.4	37.7	34.3
2^{12}	24.3	14.3	41.2	46.0	27.1	41.1	69.0	39.9	42.2
2^{16}	70.4	35.5	49.6	123.9	53.2	57.1	207.8	71.5	65.6
2^{20}	718.5	457.1	36.4	1122	675.0	39.8	1809	917.5	49.3

account for a larger fraction of the total runtime as m increases, making our optimizations increasingly beneficial.

We further evaluate the protocols under emulated WAN conditions with $m = n$. In addition to the default one-way delay of 20 ms, we consider delays of 40 and 60 ms while fixing the bandwidth at 1 Gbps. Table 7 shows that our protocol reduces runtime by around 70% compared with [8] across all available measurements. As shown in Table 8, our protocol also consistently outperforms [20], achieving reductions of up to 65.6%. At $n = 2^{20}$, the reduction over [20] increases from 36.4% at 20 ms to 49.3% at 60 ms. This trend is consistent with the reduction in communication rounds achieved by eliminating tree-based aggregation, whose cost is amplified by higher network latency. Overall, these results confirm that our protocol remains effective and becomes increasingly beneficial under high-latency network conditions.

7.5 Component-wise Analysis of Joins

Table 9 reports a component-wise breakdown of the local runtime and communication costs of Join-OM and Join-MM. We separate the total cost into three parts: GENPERM, which captures oblivious sorting; soPRF, which is the main cryptographic primitive used in data duplication; and *Const-Op*, which denotes the remaining constant-round operations beyond sorting and soPRF, including OSE, permutations, prefix sums. Notably, the contribution of Const-Op remains stable across input sizes for Join-OM and Join-MM.

For Join-OM, the additional overhead beyond sorting is small. In particular, the OSE-based step that derives the order of the expanded input from the sorted original input accounts for only about 10% of the total runtime and communication costs. This indicates that replacing direct expanded-input sorting with OSE introduces only modest overhead, while preserving the dominant benefit of sorting fewer keys.

Table 9: Component-wise cost analysis of Join-OM and Join-MM. Runtime is in seconds; communication is in MB.

Protocol	n	Metric	GENPERM	soPRF	Total	Const. (%) ¹
Join-OM	2^8	Time	0.04	0	0.05	20.0
		Comm.	2.10	0	2.28	7.9
	2^{12}	Time	0.17	0	0.19	10.5
		Comm.	32.1	0	34.4	6.7
	2^{16}	Time	2.01	0	2.24	10.3
		Comm.	491.6	0	547.8	10.3
	2^{20}	Time	38.5	0	43.7	11.9
		Comm.	7866	0	8738	10.0
	2^8	Time	0.04	0.009	0.09	45.6
		Comm.	2.10	0.20	2.83	18.7
Join-MM	2^{12}	Time	0.17	0.12	0.38	23.7
		Comm.	32.1	3.12	45.3	22.3
	2^{16}	Time	2.01	0.83	3.69	23.0
		Comm.	491.6	50.0	722.0	25.0
	2^{20}	Time	38.5	13.8	69.2	24.4
		Comm.	7866	832	11574	24.9

¹ “Const.” denotes the percentage of total runtime or communication attributed to constant-round operations not included in GENPERM or soPRF.

For Join-MM, sorting remains the largest single component, but the post-sorting phase is already entirely constant-round. Beyond GENPERM, the remaining cost is split between soPRF and the other constant-round components, namely OSE-based order derivation, duplication-count computation, oblivious data duplication, and final alignment. As shown in Table 9, these non-soPRF constant-round components contribute about 23% of the total runtime and communication cost. Thus, while Join-MM necessarily incurs nontrivial post-sorting work, our redesign removes the logarithmic-round bottlenecks of tree-based aggregation and replaces them with lightweight permutation- and prefix-sum-based operations.

8 Related Work

We review prior work on secure collaborative analytics with join processing, focusing on settings where join keys contain duplicates.

PSI-based techniques [21, 28, 29, 31–35] can be highly efficient, but they are primarily tailored to joins with unique keys and therefore do not address the join with multiplicity setting. Liagouris et al. [25] support non-unique join keys by materializing the Cartesian product of the two input tables and checking equality for every row pair. This approach preserves strong privacy properties, including hiding the exact join output size by padding to a worst-case bound, but incurs $O(n_x n_y)$ communication and is therefore impractical for large datasets.

A number of secure collaborative analytics systems improve efficiency by introducing additional assumptions or by exploiting special workload structure. Several works [10, 11, 23, 24, 40] allow controlled leakage, often combined with differential privacy, to accelerate query processing. Volgushev et al. [39] introduce a selectively trusted party and improve efficiency by sending plaintext

data to that party. Other systems [9, 16, 36] assume that data owners can reduce the input size in advance based on the target SQL query. Wang et al. [41] and Baum et al. [12] focus on restricted query classes, such as free-connex join-aggregate queries. While these approaches improve performance, they inherently limit flexibility, composability, or the strength of the security guarantee.

The most directly related line of work [6, 8, 20] studies join protocols in modern outsourced MPC settings [3, 12, 13, 20, 25, 26, 30, 43], where data owners secret-share their inputs to a small set of non-colluding servers, which then evaluate joins directly over secret-shared data. These protocols typically follow a common paradigm: the input tables are first merged and obviously sorted by the join key, and then additional post-sorting processing is used to realize the multiplicity-preserving join result. Han et al. [20] realize this post-sorting phase using prefix-tree-based traversal networks and soPRF. Their protocols support both Join-OM and Join-MM, but incur additional $O(\log n)$ rounds and $O(n)$ online multiplications due to the traversal-network-based processing. Badrinarayanan et al. [8] likewise support Join-OM and Join-MM, but rely on aggregation trees after sorting. These trees require upward and downward passes, again introducing $O(\log n)$ additional rounds and $O(n)$ online multiplications. Moreover, their Join-MM construction performs extra oblivious sorts over intermediate inputs. Asharov et al. [6] eliminate aggregation trees for Join-OM by performing a single oblivious stable sort over an expanded table obtained by concatenating the inputs with their replicas, followed by local prefix-sum computations. This removes logarithmic-round post-sorting aggregation, but substantially increases the sorting input size, making oblivious sorting the dominant bottleneck. Furthermore, their approach is tailored to Join-OM and does not directly generalize to Join-MM, where both tables may require duplication.

Attrapadung et al. [7] propose a general group-wise stable sorting technique for secure parallel computation. Their technique can serve as a building block for realizing the core step of OSE, namely, generating the stable sorting permutation of $(\vec{k}^{\text{sort}} \parallel \vec{k}^{\text{sort}})$. However, it is a general-purpose sorting technique and does not exploit the replicated-vector structure that motivates OSE in our join protocols. We provide a detailed comparison in Section 4, showing that our construction incurs only half the computation and communication cost of a [7]-based realization.

Our protocols in context. Taken together, the above works expose two main sources of overhead in secret-shared joins with multiplicity: tree-based post-sorting processing and sorting over expanded inputs. Our protocols are designed to address both. OSE is motivated by the replicated-vector structure arising in secure joins and avoids redundant expanded-input sorting. For Join-OM, it reduces the sorting input size of [6] from $2n_x + n_y$ to $n_x + n_y$. For Join-MM, combining OSE with our prefix-sum-with-cancellation technique enables an aggregation-tree-free construction without increasing the sorting input size, yielding constant-round post-sorting computation.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. The work is supported in part by the National Natural Science

Foundation of China (No. 62272251, No. U25B2028) and National Cryptologic Science Fund of China (No. 2025NCSF01010).

References

- [1] Navid Alamati, Guru-Vamsi Policharla, Srinivasan Raghuraman, and Peter Rindal. 2024. Improved Alternating-Moduli PRFs and Post-quantum Signatures. In *Advances in Cryptology - CRYPTO 2024 - 44th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2024, Proceedings, Part VIII (Lecture Notes in Computer Science, Vol. 14927)*. Springer, 274–308.
- [2] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. 2015. Ciphers for MPC and FHE. In *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26–30, 2015, Proceedings, Part I (Lecture Notes in Computer Science, Vol. 9056)*. Springer, 430–454.
- [3] Apple and Google. 2021. Exposure Notification Privacy-preserving Analytics (ENPA). https://covid19-static.cdn-apple.com/applications/covid19/current/statistics/contact-tracing/pdf/ENPA_White_Paper.pdf. Online; accessed April 2025.
- [4] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24–28, 2016*. ACM, 805–817.
- [5] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7–11, 2022*. ACM, 125–138.
- [6] Gilad Asharov, Koki Hamada, Ryo Kikuchi, Ariel Nof, Benny Pinkas, and Junichi Tomida. 2023. Secure Statistical Analysis on Multiple Datasets: Join and Group-By. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26–30, 2023*. ACM, 3298–3312.
- [7] Nuttapon Attrapadung, Kota Isayama, Kunihiko Sadakane, and Kazunari Tozawa. 2024. Secure Parallel Computation with Oblivious State Transitions. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14–18, 2024*. ACM, 3008–3022.
- [8] Saikrishna Badrinarayanan, Sourav Das, Gayathri Garimella, Srinivasan Raghuraman, and Peter Rindal. 2022. Secret-Shared Joins with Multiplicity from Aggregation Trees. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7–11, 2022*. ACM, 209–222.
- [9] Johnes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. 2017. SMCQL: secure querying for federated databases. *Proc. VLDB Endow.* 10, 6 (2017), 673–684.
- [10] Johnes Bater, Xi He, William Ehrlich, Ashwin Machanavajjhala, and Jennie Rogers. 2018. ShrinkWrap: Efficient SQL Query Processing in Differentially Private Data Federations. *Proc. VLDB Endow.* 12, 3 (2018), 307–320.
- [11] Johnes Bater, Yongjoo Park, Xi He, Xiao Wang, and Jennie Rogers. 2020. SAQE: Practical Privacy-Preserving Approximate Query Processing for Data Federations. *Proc. VLDB Endow.* 13, 11 (2020), 2691–2705.
- [12] Eli Baum, Sam Buxbaum, Nitin Mathai, Muhammad Faisal, Vasiliki Kalavri, Mayank Varia, and John Liagouris. 2025. ORQ: Complex Analytics on Private Data with Strong Security Guarantees. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles, SOSP 2025, Lotte Hotel World, Seoul, Republic of Korea, October 13–16, 2025*. ACM, 802–833.
- [13] Boston Women’s Workforce Council. 2024. Data Privacy: Ensuring Secure and Private Data Analysis. <https://thebwcc.org/mpc>. Online; accessed March 2026.
- [14] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Naoto Kiribuchi, and Benny Pinkas. 2019. An Efficient Secure Three-Party Sorting Protocol with an Honest Majority. *IACR Cryptol. ePrint Arch.* 2019 (2019), 695.
- [15] Ronald Cramer, Ivan Damgård, and Yuval Ishai. 2005. Share Conversion, Pseudo-random Secret-Sharing and Applications to Secure Computation. In *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10–12, 2005, Proceedings (Lecture Notes in Computer Science, Vol. 3378)*. Springer, 342–362.
- [16] Wenjing Fang, Shunde Cao, Guojin Hua, Junming Ma, Yongqiang Yu, Qunshan Huang, Jun Feng, Jin Tan, Xiaopeng Zan, Pu Duan, Yang Yang, Li Wang, Ke Zhang, and Lei Wang. 2024. SecretFlow-SCQL: A Secure Collaborative Query pLatform. *Proc. VLDB Endow.* 17, 12 (2024), 3987–4000.
- [17] Dengguo Feng and Kang Yang. 2022. Concretely efficient secure multi-party computation protocols: survey and more. *Secur. Saf.* 1 (2022), 2021001. doi:10.1051/SANDS/2021001
- [18] Jun Furukawa, Yehuda Lindell, Ariel Nof, and Or Weinstein. 2017. High-Throughput Secure Three-Party Computation for Malicious Adversaries and an Honest Majority. In *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual*

- International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 10211)*, 225–255.
- [19] Oded Goldreich, Silvio Micali, and Avi Wigderson. 1987. How to Play any Mental Game or A Completeness Theorem for Protocols with Honest Majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing*, 1987, New York, New York, USA. ACM, 218–229.
- [20] Feng Han, Lan Zhang, Hanwen Feng, Weiran Liu, and Xiang-Yang Li. 2022. Scape: Scalable Collaborative Analytics System on Private Database with Malicious Security. In *38th IEEE International Conference on Data Engineering, ICDE 2022, Kuala Lumpur, Malaysia, May 9-12, 2022*. IEEE, 1740–1753.
- [21] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. 2016. Efficient Batched Oblivious PRF with Applications to Private Set Intersection. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*. ACM, 818–829.
- [22] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. 2020. Efficient Oblivious Database Joins. *Proc. VLDB Endow.* 13, 11 (2020), 2132–2145.
- [23] Sven Laur, Riivo Talviste, and Jan Willemson. 2013. From Oblivious AES to Efficient and Secure Database Join in the Multiparty Setting. In *Applied Cryptography and Network Security - 11th International Conference, ACNS 2013, Banff, AB, Canada, June 25-28, 2013. Proceedings (Lecture Notes in Computer Science, Vol. 7954)*. Springer, 84–101.
- [24] Phi Hung Le, Samuel Ranellucci, and S. Dov Gordon. 2019. Two-party Private Set Intersection with an Untrusted Third Party. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*. ACM, 2403–2420.
- [25] John Liagouris, Vasiliki Kalavri, Muhammad Faisal, and Mayank Varia. 2023. SECRECY: Secure collaborative analytics in untrusted clouds. In *20th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2023, Boston, MA, April 17-19, 2023*. USENIX Association, 1031–1056.
- [26] 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 *2023 USENIX Annual Technical Conference (USENIX ATC 23)*. Boston, MA, 17–33.
- [27] Payman Mohassel and Peter Rindal. 2018. ABY³: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 35–52.
- [28] Payman Mohassel, Peter Rindal, and Mike Rosulek. 2020. Fast Database Joins and PSI for Secret Shared Data. In *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*. ACM, 1271–1287.
- [29] Michele Orrù, Emmanuela Orsini, and Peter Scholl. 2017. Actively Secure 1-out-of-N OT Extension with Application to Private Set Intersection. In *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10159)*. Springer, 381–396.
- [30] Xinyu Peng, Feng Han, Li Peng, Weiran Liu, Zheng Yan, Kai Kang, Xinyuan Zhang, Guoxing Wei, Jianling Sun, and Jinfei Liu. 2024. MapComp: A Secure View-based Collaborative Analytics Framework for Join-Group-Aggregation. *CoRR abs/2408.01246* (2024).
- [31] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2019. SpOT-Light: Lightweight Private Set Intersection from Sparse OT Extension. In *Advances in Cryptology - CRYPTO 2019 - 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2019, Proceedings, Part III (Lecture Notes in Computer Science, Vol. 11694)*. Springer, 401–431.
- [32] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2020. PSI from PaXoS: Fast, Malicious Private Set Intersection. In *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12106)*. Springer, 739–767.
- [33] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. 2015. Phasing: Private Set Intersection Using Permutation-based Hashing. In *24th USENIX Security Symposium (USENIX Security 15)*. USENIX Association, 515–530.
- [34] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2014. Faster Private Set Intersection Based on OT Extension. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*. USENIX Association, 797–812.
- [35] Benny Pinkas, Thomas Schneider, and Michael Zohner. 2018. Scalable Private Set Intersection Based on OT Extension. *ACM Trans. Priv. Secur.* 21, 2 (2018), 7:1–7:35.
- [36] Rishabh Poddar, Sukrit Kalra, Avishay Yanai, Ryan Deng, Raluca Ada Popa, and Joseph M. Hellerstein. 2021. Senate: A Maliciously-Secure MPC Platform for Collaborative Analytics. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 2129–2146.
- [37] Srinivasan Raghuraman, Peter Rindal, and Harshal Shah. 2025. Two Party Secret Shared Joins. *Cryptology ePrint Archive*, Paper 2025/702. <https://eprint.iacr.org/2025/702>
- [38] Junjian Shi, Ye Han, Xiaojie Guo, Zekun Fei, Zheli Liu, Siyi Lv, Tong Li, and Xiaotao Liu. 2025. SMPCache: Towards More Efficient SQL Queries in Multi-Party Collaborative Data Analysis. *IEEE Trans. Knowl. Data Eng.* 37, 4 (2025), 2111–2125.
- [39] Nikolaj Volgushev, Malte Schwarzkopf, Ben Getchell, Mayank Varia, Andrei Lapets, and Azer Bestavros. 2019. Conclave: secure multi-party computation on big data. In *Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019*. ACM, 3:1–3:18.
- [40] Chenghong Wang, Johes Bater, Kartik Nayak, and Ashwin Machanavajjhala. 2022. IncShrink: Architecting Efficient Outsourced Databases using Incremental MPC and Differential Privacy. In *SIGMOD '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022*. ACM, 818–832.
- [41] Yilei Wang and Ke Yi. 2021. Secure Yannakakis: Join-Aggregate Queries over Private Data. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. ACM, 1969–1981.
- [42] Haoqi Wu, Wenjing Fang, Yancheng Zheng, Junming Ma, Jin Tan, and Lei Wang. 2024. Ditto: Quantization-aware Secure Inference of Transformers upon MPC. In *Proceedings of the 41st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 235)*. PMLR, 53346–53365.
- [43] Peizhao Zhou, Xiaojie Guo, Pinzhi Chen, Tong Li, Siyi Lv, and Zheli Liu. 2024. Shortcut: Making MPC-based Collaborative Analytics Efficient on Dynamic Databases. In *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, CCS 2024, Salt Lake City, UT, USA, October 14-18, 2024*. ACM, 854–868.

A Open Science

We provide the artifacts supporting our results at the following anonymous URL: <https://anonymous.4open.science/r/Join-OSE/>. The repository contains the following components:

- **Source Code:** The full implementation of our protocols, including OSE, Join-OM, and Join-MM.
- **Documentation:** A detailed README.md file with instructions for environment setup (via Docker) and reproduction of the results.
- **Data Generation:** As the core contributions of this paper do not rely on specific external datasets, the provided Source Code includes the logic to generate synthetic/random data used for evaluation.

B Ethical Considerations

This work does not involve human subjects, user data, or real-world system deployment, and thus does not raise ethical concerns.

C Detailed Protocol Descriptions

C.1 Completing a position vector.

We present COMPLETEPERM in Protocol 10. Given inputs $[\vec{p}]$ and M , COMPLETEPERM computes the remaining positions in $[M]$ that are not assigned to $[\vec{p}]$ and appending these positions to $[\vec{p}]$. Our COMPLETEPERM protocol is adapted from [20]. Importantly, the computations in the first two steps can be shifted to the offline phase. Specifically, we estimate an upper bound M' on M and invoke $\mathcal{F}_{\text{soPRF}}$ on inputs $[[M']]$. During the online phase, surplus elements are discarded if $M' > M$, while missing elements are generated online if $M' < M$.

C.2 Supporting Padded Join Sizes

In this subsection, we modify Protocol 8 to support either a fixed public upper bound $M \geq m$ or a differentially private upper bound $m' \geq m$, instead of revealing the exact join size $m = |Z|$. For the

Protocol 10 ($(\llbracket \vec{\pi}^{\text{inv}} \rrbracket \leftarrow \text{COMPLETEPERM}(\llbracket \vec{p} \rrbracket, M))$):

Input: Secret shares of the vector $\llbracket \vec{p} \rrbracket$ of length n , where $p_1, \dots, p_n$ are distinct values in $[M]$.

Output: Secret shares of inverse permutation $\llbracket \vec{\pi}^{\text{inv}} \rrbracket$.

The protocol:

- (1) Generate vector $\llbracket \vec{a} \rrbracket = (\llbracket 1 \rrbracket, \dots, \llbracket M \rrbracket)$ and a random key $\llbracket k \rrbracket$ by calling $\mathcal{F}_{\text{random}}$.
 - (2) $\llbracket \vec{a}^{\text{enc}} \rrbracket = \mathcal{F}_{\text{soPRF}}(\llbracket k \rrbracket, \llbracket \vec{a} \rrbracket)$.
 - (3) Generate a random permutation $\llbracket \vec{\pi} \rrbracket$ by calling $\mathcal{F}_{\text{random}}$.
 - (4) Send $(\text{ApplyPerm}, \llbracket \vec{\pi} \rrbracket, \llbracket \vec{a} \rrbracket), (\text{ApplyPerm}, \llbracket \vec{\pi} \rrbracket, \llbracket \vec{a}^{\text{enc}} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$, and receive the corresponding outputs $\llbracket \vec{b} \rrbracket$ and $\llbracket \vec{b}^{\text{enc}} \rrbracket$.
 - (5) $\llbracket \vec{p}^{\text{enc}} \rrbracket = \mathcal{F}_{\text{soPRF}}(\llbracket k \rrbracket, \llbracket \vec{p} \rrbracket)$.
 - (6) Open $\llbracket \vec{b}^{\text{enc}} \rrbracket$ and $\llbracket \vec{p}^{\text{enc}} \rrbracket$.
 - (7) Generate a vector $\vec{c}$ that contains the indices of all elements belonging to $\vec{b}^{\text{enc}}$ but not to $\vec{p}^{\text{enc}}$.
 - (8) Construct a secret-shared vector $\llbracket \vec{r} \rrbracket$, where $\llbracket r_i \rrbracket = \llbracket b_{c_i} \rrbracket$.
 - (9) Output $\llbracket \vec{p} \rrbracket \parallel \llbracket \vec{r} \rrbracket$.
-

Protocol 11 ($(\llbracket \vec{\pi} \rrbracket \leftarrow \text{GENALIGNPERM-PAD}(\llbracket \vec{a}_Y \rrbracket, \llbracket \vec{d}_X \rrbracket, \llbracket \vec{d}_Y \rrbracket, \llbracket \vec{\pi}_X \rrbracket, \llbracket \vec{\pi}_Y \rrbracket, \llbracket \vec{v}_Y^{\text{dup}} \rrbracket))$):

Input: Secret shares of vectors $\llbracket \vec{a}_Y \rrbracket, \llbracket \vec{d}_X \rrbracket, \llbracket \vec{d}_Y \rrbracket$, permutations $\llbracket \vec{\pi}_X \rrbracket, \llbracket \vec{\pi}_Y \rrbracket$ and valid vector $\llbracket \vec{v}_Y^{\text{dup}} \rrbracket$.

Output: Secret shares of the inverse alignment permutation $\llbracket \vec{\pi} \rrbracket$.

The protocol:

- (1) Execute Lines 1-3 of Protocol 9.
 - (2) For $\vec{t} \in \{\vec{d}_X', \vec{d}_Y', \vec{a}_Y', \vec{f}_Y'\}$ and $i \in [m']$, set $\llbracket t_i \rrbracket \leftarrow \text{IFTHEN}(\llbracket \vec{v}_Y^{\text{dup}} \rrbracket[i] : \llbracket t_i \rrbracket, \llbracket 1 \rrbracket)$.
 - (3) Execute Lines 4-5 of Protocol 9.
-

latter setting, Bater et al. [10] proposed a truncated Laplace mechanism that adds non-negative noise to the secret-shared join size $\llbracket m \rrbracket$ and reveals the resulting upper bound m' . When M or m' is used in place of m , **Stage 2** requires no modification: it automatically produces $M - m$ or $m' - m$ additional all-zero rows at the end of $\llbracket X^{\text{dup}} \rrbracket$ and $\llbracket Y^{\text{dup}} \rrbracket$. However, **Stage 3** must be modified to ensure that these padded rows are properly aligned. Specifically, in Line 6 of INNERJOIN-MM, we replace GENALIGNPERM (Protocol 9) with a modified version, denoted by GENALIGNPERM-PAD (Protocol 11), in which the variables corresponding to padded rows are fixed to $\llbracket 1 \rrbracket$, ensuring that these rows remain at the end of the aligned tables. Moreover, in Line 8, we set $\llbracket \vec{v}_Z \rrbracket = \llbracket \vec{v}_Y^{\text{dup}} \rrbracket$. Consequently, the modified protocol reveals only the fixed bound M or the differentially private bound m' , while the padded rows remain invalid and zero-valued and therefore do not affect the join result.

C.3 Outer Join-MM

We present the protocol for left outer Join-MM in Protocol 12. The output table has size $m + n_X$. It is obtained by augmenting the result of the inner Join-MM with all rows of $\llbracket X \rrbracket$. Rows of $\llbracket X \rrbracket$ that appear in the inner join are marked invalid by setting their validity bits to $\llbracket 0 \rrbracket$, whereas rows with no matches in $\llbracket Y \rrbracket$ are retained as valid

Protocol 12 ($(\llbracket Z \rrbracket \leftarrow \text{LEFTJOIN-MM}(\llbracket X \rrbracket, \llbracket Y \rrbracket))$):

Input: Secret shares of the input table $\llbracket X \rrbracket, \llbracket Y \rrbracket$.

Output: Secret shares of the result of left outer join $\llbracket Z \rrbracket$.

The protocol: Each party does the following:

- (1) Execute Lines 1-7 of Protocol 8
 $\triangleright$ Lines 1-3: Obtain the unmatched rows
 - (2) Set $\llbracket \vec{v} \rrbracket \leftarrow \text{EQUAL}(\llbracket d_X \rrbracket, \llbracket 0 \rrbracket^{n_X})$.
 - (3) $\llbracket \vec{v}' \rrbracket = \llbracket \vec{v} \rrbracket \cdot \llbracket \vec{v}_X^{\text{sort}} \rrbracket$
 - (4) $\llbracket \vec{v}_Z \rrbracket = \llbracket 1 \rrbracket^m \parallel \llbracket \vec{v}' \rrbracket$
 - (5) For each column $\llbracket c \rrbracket$ in $\llbracket X^{\text{sort}} \rrbracket$, update $\llbracket c \rrbracket \leftarrow \llbracket c \rrbracket \cdot \llbracket \vec{v}' \rrbracket$. The result table is $\llbracket X^{\text{outer}} \rrbracket$
 - (6) Concatenate $\llbracket X^{\text{outer}} \rrbracket$ and $\llbracket Z \rrbracket$ row-wise according to Figure 2, where the parts corresponding to table Y are filled with zeros and the validity column is $\llbracket \vec{v}_Z \rrbracket$. The result table is $\llbracket Z^{\text{outer}} \rrbracket$.
 - (7) **Output:** $\llbracket Z^{\text{outer}} \rrbracket$.
-

Parameters: Circuit C encoding the functionality to be computed, and the corresponding secret-shared input $\llbracket x \rrbracket$.

$[\mathcal{F}_{\text{MPC}}]$ Upon receiving $(C, \llbracket x \rrbracket)$ from all parties,

- (1) Reconstruct x and compute $y = C(x)$.
 - (2) Return to the parties the secret sharing $\llbracket y \rrbracket$.
-

Figure 6: Secure computation ideal functionality $\mathcal{F}_{\text{MPC}}$

outputs. The results of the outer join operations are illustrated in Table 10.

C.4 Self-Join

Self-join arises when the two sides of the join are aliases of the same input table. We view the secret-shared table $\llbracket X \rrbracket$ as two logical aliases, $\llbracket X^L \rrbracket$ and $\llbracket X^R \rrbracket$, joined on the same key column. When one alias contains at most one anchor row per key, the query corresponds to self-Join-OM; when both aliases may contain multiple rows per key, it corresponds to self-Join-MM. Let $\llbracket \vec{k} \rrbracket$ be the join-key column of $\llbracket X \rrbracket$. We first compute its stable sorting permutation $\llbracket \vec{\rho} \rrbracket$, and then invoke OSE on $(\llbracket \vec{k} \rrbracket, \llbracket \vec{\rho} \rrbracket)$ to derive the stable sorting permutation $\llbracket \vec{\sigma} \rrbracket$ of $\llbracket \vec{k} \rrbracket \parallel \llbracket \vec{k} \rrbracket$. After deriving this base permutation, we reuse the same high-level pipeline as the corresponding Join-OM or Join-MM protocol.

D Formal Security Proofs

D.1 Secure computation ideal functionality

Figure 6 illustrates the ideal functionality $\mathcal{F}_{\text{MPC}}$, which realizes the secure computation of an arbitrary circuit.

D.2 Semi-Honest Security of OSE

THEOREM 3. (Theorem 1, restated) Protocol 1 securely realizes $\mathcal{F}_{\text{OSE}}$ in the $(\mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}})$ -hybrid model against semi-honest adversaries.

PROOF. Correctness follows from Section 4. For security, the real-world view of each party P_i consists of its input shares $\llbracket \vec{k} \rrbracket_i$ and

Table 10: Definition of the output tables for the four Join-MM types, where $|X| = n_x$, $|Y| = n_y$, and m is their inner join size.

Join Type	Key from X	Key from Y	Values from X	Values from Y	valid	Condition
Inner join	$\vec{k}_X[i]$	$\vec{k}_Y[j]$	$X_{i,1}, \dots, X_{i,p_x}$	$Y_{j,1}, \dots, Y_{j,p_y}$	1	$\vec{k}_X[i] = \vec{k}_Y[j]$
For $i \leq m$, left outer join is the same as inner join; For $m < i \leq m + n_x$:						
Left outer join	$\vec{k}_X[i]$	0	$X_{i,1}, \dots, X_{i,p_x}$	0	1	$\forall j : \vec{k}_X[i] \neq \vec{k}_Y[j]$
	0	0	0	0	0	otherwise
For $i \leq m$, right outer join is the same as inner join; For $m < i \leq m + n_y$:						
Right outer join	0	$\vec{k}_Y[i]$	0	$Y_{j,1}, \dots, Y_{j,p_y}$	1	$\forall j : \vec{k}_Y[i] \neq \vec{k}_X[j]$
	0	0	0	0	0	otherwise
For $i \leq m + n_x$, full join is the same as left outer join;						
Full join	For $m + n_x < i \leq m + n_x + n_y$, full join is the complement of right outer join.					

$\llbracket \vec{\sigma}^{\text{base}} \rrbracket_i$, together with all messages received from the ideal functionalities $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, $\mathcal{F}_{\text{sort}}$ during the execution of Protocol 1. The output of P_i consists of its share of the resulting permutation $\llbracket \vec{\sigma} \rrbracket_i$. Observe that in both the real world and the ideal world, the final output received by each party is a secret share of the stable sorting permutation for $\llbracket \vec{k} \rrbracket \parallel \llbracket \vec{k} \rrbracket$. By the security guarantees of the underlying secret-sharing scheme, these output shares are indistinguishable from uniformly random shares and independent of the party's view. Consequently, it suffices to construct a simulator that reproduces the distribution of P_i 's view. We construct a simulator $\mathcal{S}$ that, given $\llbracket \vec{k} \rrbracket_i$, $\llbracket \vec{\sigma}^{\text{base}} \rrbracket_i$, and $\llbracket \vec{\sigma} \rrbracket_i$, generates a simulated view that is indistinguishable from the real-world view of P_i . The simulator proceeds by simulating the outputs of the invoked ideal functionalities. For each invocation of $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, $\mathcal{F}_{\text{sort}}$, the simulator samples uniformly random output shares and return these. By the security guarantees of the invoked ideal functionalities and the privacy of the secret-sharing scheme, the simulated values generated by $\mathcal{S}$ are indistinguishable from those observed in a real execution. Therefore, the real-world and ideal-world views of P_i are indistinguishable. $\square$

D.3 Semi-honest Security of Join-MM

In this subsection, we prove the semi-honest security of INNERJOIN-MM in Protocol 8.

THEOREM 4. (Theorem 2, restated) Protocol 8 securely realizes $\mathcal{F}_{\text{Join-MM}}$ in the $(\mathcal{F}_{\text{OSE}}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}}, \mathcal{F}_{\text{soPRF}})$ -hybrid model against semi-honest adversaries.

PROOF. We prove its correctness and security respectively.

Correctness. We first prove that Protocol 8 correctly implements the semantics of the inner join, assuming that no encoding collisions occur. The protocol proceeds in three stages, which we analyze in turn.

Stage 2: Computing Duplication Counts. Protocol 8 first invokes the DUPCOUNT subprotocol to compute, for each row in the input tables X and Y , its duplication count, i.e., the number of matching rows in the other table with the same join key. We describe

the correctness argument for table X ; the argument for Y is analogous and therefore omitted. The DUPCOUNT protocol constructs the vector $\llbracket \vec{f}_X \rrbracket = \llbracket 0 \rrbracket^{n_x} \parallel \llbracket 1 \rrbracket^{n_y} \parallel \llbracket 0 \rrbracket^{n_x} \parallel \llbracket -1 \rrbracket^{n_y}$, and applies to it the stable sorting permutation σ induced by the concatenated key vector $\llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket \parallel \llbracket \vec{k}_X \rrbracket \parallel \llbracket \vec{k}_Y \rrbracket$. Let $\llbracket \vec{g}_X \rrbracket$ denote the permuted vector. By the stability of the sorting procedure, the permuted vector $\llbracket \vec{g}_X \rrbracket$ decomposes into disjoint contiguous sub-sequences, each corresponding to a distinct join key. There are three possible cases.

- *Matching keys in both tables.* Suppose a key k appears $X(k)$ times in X and $Y(k)$ times in Y . The corresponding sub-sequence in $\llbracket \vec{g}_X \rrbracket$ has the form $(0^{X(k)}, 1^{Y(k)}, 0^{X(k)}, (-1)^{Y(k)})$. Assuming the prefix sum entering this sub-sequence is zero, the resulting prefix-sum values are $(0^{X(k)}, [Y(k)], Y(k)^{X(k)}, Y(k) - 1, Y(k) - 2, \dots, 0)$. Consequently, each row in X with join key k receives the value $Y(k)$, which is exactly its required duplication count.
- *Keys appearing only in X .* If a key k appears $X(k)$ times in X and does not appear in Y , the corresponding sub-sequence is $(0^{X(k)}, 0^{X(k)})$, and all prefix-sum values remain zero. Thus, rows with no matching partner receive a duplication count of zero.
- *Keys appearing only in Y .* If a key k appears $Y(k)$ times only in Y , the corresponding sub-sequence has the form $(1^{Y(k)}, (-1)^{Y(k)})$. Assuming the prefix sum entering this sub-sequence is zero, the prefix sum decreases from $Y(k)$ to 0, and the final value is zero.

In all cases, the prefix sum at the end of each sub-sequence is zero, ensuring that sub-sequences are independent and that duplication counts are computed correctly. The correctness of duplication counts for table Y follows by symmetry.

Finally, since input tables may contain invalid rows whose join keys are set to zero, the DUPCOUNT protocol may assign non-zero duplication counts to such rows. Protocol 8 explicitly resets the duplication counts of invalid rows to zero (line 3 of the protocol), ensuring that they do not contribute to the join result.

Stage 3: Data Duplication. Given the computed duplication counts, Protocol 8 duplicates each column of the sorted table $\llbracket X^{\text{sort}} \rrbracket$ (resp.

$\llbracket Y^{\text{sort}} \rrbracket$) accordingly. For a column c , the protocol constructs an extended vector $c \parallel -c \parallel 0^m$ and applies a suitable permutation so that each value c appears in a sub-sequence of the form $(c, 0^{Y(k)}, -c)$. Computing the prefix sum over this sub-sequence yields $(c, c^{Y(k)}, 0)$, where the last element is zero. Extracting the appropriate positions from the 0^m block therefore produces exactly $Y(k)$ copies of c . Applying this procedure column-wise yields a table in which each row of $\llbracket X^{\text{sort}} \rrbracket$ (resp. $\llbracket Y^{\text{sort}} \rrbracket$) is duplicated according to its duplication-count vector.

Stage 4: Aligning Duplicated Rows. In the final stage, the protocol computes an alignment permutation for the duplicated table $\llbracket Y^{\text{dup}} \rrbracket$. The definition of this permutation follows the same formulation as in [20]. To this end, the parties invoke the protocol GENALIGNPERM (Protocol 9) to securely compute the intermediate vectors required for deriving the alignment permutation. Applying the resulting permutation to $\llbracket Y^{\text{dup}} \rrbracket$ ensures that each duplicated row in $\llbracket Y^{\text{dup}} \rrbracket$ is aligned with the corresponding duplicated row in $\llbracket X^{\text{dup}} \rrbracket$ having the same join key. Consequently, rows with matching join keys appear at identical positions in the two duplicated tables. By concatenating the aligned tables column-wise, the protocol obtains exactly the inner join result table $\llbracket Z \rrbracket$.

For encoding collisions, Protocol 10 is invoked twice independently with different $\mathcal{F}_{\text{soPRF}}$ keys. In these two invocations, $\mathcal{F}_{\text{soPRF}}$ is evaluated on $n_x + m$ and $n_y + m$ distinct values, respectively. Let n' be an upper bound on these evaluation counts (i.e., $n' = \max(n_x + m, n_y + m)$). By the standard birthday bound, for encodings of length l bits, the probability of an internal collision within a single invocation is at most $2^{-l+2\log_2(n')-1}$. By the union bound over the two independent invocations, the probability that *any* collision occurs is bounded by $2 \cdot 2^{-l+2\log_2(n')-1} = 2^{-l+2\log_2(n')}$. Therefore, choosing $l \geq \lambda + 2\log_2 n'$ ensures that all encodings are distinct except with probability at most $2^{-\lambda}$.

Security. The security follows closely from the proof of Theorem 3, with necessary modifications to the simulator $\mathcal{S}$ to account for the additional invocations of ideal functionalities, the public reconstruction of the output size m , and the opened encodings in the COMPLETEPERM subroutine.

Specifically, the real-world view of P_i in Protocol 8 additionally includes the messages received from $\mathcal{F}_{\text{OSE}}$ and $\mathcal{F}_{\text{soPRF}}$, the reconstruction transcript and the reconstructed value of the output size m , and the reconstruction transcripts and opened values of the encodings in the COMPLETEPERM subroutine. For notational convenience, let n_b and n_p denote the public length parameters of one invocation of COMPLETEPERM, where n_b is the length of the opened universe-encoding vector $\vec{b}^{\text{enc}}$ and n_p is the length of the opened selected-encoding vector $\vec{p}^{\text{enc}}$. The secret-shared input vector $\vec{p}$ to this invocation consists of n_p distinct positions in $[n_b]$. In Protocol 8, the two invocations correspond to $(n_b, n_p) = (m + n_x, n_x)$ and $(n_b, n_p) = (m + n_y, n_y)$, respectively. These parameters are known to the simulator, since n_x and n_y are public input sizes and m is revealed as the output size. Conditioned on no encoding collision, $\vec{b}^{\text{enc}}$ is a uniformly random permutation of the PRF encodings of the public universe $[n_b]$, and $\vec{p}^{\text{enc}}$ consists of the PRF encodings of the secret positions $\vec{p}$. Since every entry of $\vec{p}$ lies in $[n_b]$, every element of $\vec{p}^{\text{enc}}$ appears in $\vec{b}^{\text{enc}}$. Moreover, since the permutation

used to form $\vec{b}^{\text{enc}}$ is secret-shared and hidden from the adversary, the positions of these selected encodings in $\vec{b}^{\text{enc}}$ are uniformly random and independent of the actual secret positions $\vec{p}$.

To construct the simulated view, $\mathcal{S}$ proceeds as follows. First, similar to the simulation in Theorem 3, $\mathcal{S}$ simulates the outputs of $\mathcal{F}_{\text{OSE}}$ and $\mathcal{F}_{\text{soPRF}}$ by sampling uniformly random shares. By the privacy guarantees of the secret-sharing scheme, these simulated shares are indistinguishable from the real execution.

Second, $\mathcal{S}$ simulates the public reconstruction of the output size m . Since m is the length of the final join output, it is deterministically determined by the output of the ideal functionality $\mathcal{F}_{\text{join-MM}}$. Thus, $\mathcal{S}$ obtains m from the ideal output. To simulate the reconstruction transcript of m , $\mathcal{S}$ uses P_i 's simulated share of m and samples the other parties' shares uniformly at random subject to the constraint that all shares reconstruct the public value m . This matches the real opening transcript, because the shares of an opened secret are uniformly random subject only to the reconstructed value.

Third, the simulator $\mathcal{S}$ simulates the opened values in COMPLETEPERM using only the public parameters n_b and n_p . For one invocation, $\mathcal{S}$ samples n_b distinct uniform ℓ -bit strings $(r_1, \dots, r_{n_b})$ and uses them as the simulated opening of $\vec{b}^{\text{enc}}$. It then samples an ordered n_p -tuple of distinct indices $(j_1, \dots, j_{n_p})$ uniformly from $[n_b]$, and sets the simulated opening of $\vec{p}^{\text{enc}}$ to $(r_{j_1}, \dots, r_{j_{n_p}})$. Conditioned on no encoding collision, this distribution is identical to the real distribution, up to the security of the realization of $\mathcal{F}_{\text{soPRF}}$. To simulate the reconstruction transcript, $\mathcal{S}$ samples the other parties' shares uniformly at random subject to the constraint that, together with P_i 's simulated share, they reconstruct the target opened values defined above. This matches the real opening transcript, because the shares of an opened secret are uniformly random subject only to the reconstructed value. Since the target opened values have already been sampled from the same distribution as in the real execution, the entire simulated opening of $\vec{b}^{\text{enc}}$ and $\vec{p}^{\text{enc}}$ is distributed as in the real execution and reveals no information about $\vec{p}$ beyond the public parameters n_b and n_p . The remainder of the simulation proceeds exactly as in Theorem 3. Because all simulated shares are uniformly random and the opened values is perfectly simulated using the ideal output, the real-world and ideal-world views of P_i remain computationally indistinguishable.

Hence, we conclude that Protocol 8 securely realizes $\mathcal{F}_{\text{join-MM}}$ in the $(\mathcal{F}_{\text{OSE}}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}}, \mathcal{F}_{\text{soPRF}})$ -hybrid model against semi-honest adversaries. $\square$

D.4 Semi-honest Security for Join-OM Protocol

Let $\mathcal{F}_{\text{join-OM}}$ denote the ideal functionality that, on input two secret-shared tables $\llbracket X \rrbracket$ and $\llbracket Y \rrbracket$, reconstructs them, computes their inner join Z , and returns the resulting secret-shared output table $\llbracket Z \rrbracket$ to the parties.

THEOREM 5. *Protocol 4 securely realizes $\mathcal{F}_{\text{join-OM}}$ in the $(\mathcal{F}_{\text{OSE}}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}})$ -hybrid model against semi-honest adversaries.*

PROOF. The correctness follows from the correctness of the OSE in Protocol 1 and the correctness of the Join-OM protocol in [6].

The security argument closely follows the proof of Theorem 3, as Protocol 4 operates in a similar manner but additionally invokes the ideal functionality $\mathcal{F}_{\text{OSE}}$. Specifically, the real-world view of

a party P_i during the execution of Protocol 4 consists of its input shares, its internal randomness, and the messages it receives from the ideal functionalities $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, $\mathcal{F}_{\text{sort}}$, and $\mathcal{F}_{\text{OSE}}$. To construct a simulated view, the simulator $\mathcal{S}$ proceeds exactly as in the proof of Theorem 3 for the functionalities $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, and $\mathcal{F}_{\text{sort}}$, by sampling uniformly random shares. Furthermore, for the invocation of $\mathcal{F}_{\text{OSE}}$, $\mathcal{S}$ similarly simulates the output messages received by P_i by generating uniformly random secret shares of the appropriate dimension. Because the protocol operates in the $\mathcal{F}_{\text{OSE}}$ -hybrid model, the underlying OSE execution is perfectly secure, meaning its intermediate outputs reveal nothing beyond the final shares. By the privacy properties of the secret-sharing scheme, all intermediate shares received by P_i , including those from $\mathcal{F}_{\text{OSE}}$, appear uniformly random. Therefore, the simulated shares generated by $\mathcal{S}$ are indistinguishable from those in a real execution, rendering the real-world and ideal-world views of P_i computationally indistinguishable. $\square$

D.5 Malicious Security Extensions

In this subsection, we show that our protocols can be extended to achieve security against malicious adversaries under an honest-majority assumption. Specifically, the protocols can be instantiated using a maliciously secure three-party multiplication protocol with honest majority [18] and a maliciously secure sorting protocol [5].

To model malicious behavior, we adopt the standard *with-abort* ideal functionalities. In particular, the ideal functionalities $\mathcal{F}_{\text{OSE}}$, $\mathcal{F}_{\text{Join-MM}}$, $\mathcal{F}_{\text{Join-OM}}$, $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, $\mathcal{F}_{\text{sort}}$, and $\mathcal{F}_{\text{soPRF}}$ are modified as follows: (i) the ideal functionality receives inputs from the honest parties and provides the input of the corrupted party to the ideal-world adversary, and (ii) the adversary is allowed to choose the output share of the corrupted party, upon which the functionality computes consistent output shares for the honest parties or aborts. For Join-MM, we let $\mathcal{F}_{\text{Join-MM}}$ hand the output size to the adversary. Note that this is not private information and is learned by the adversary in the protocol anyway.

THEOREM 6. *Protocol 1 securely realizes $\mathcal{F}_{\text{OSE}}$ with abort in the $(\mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}})$ -hybrid model against malicious adversaries controlling any $t < n/2$ parties.*

PROOF. Correctness follows from Section 4. We next prove the security. Let $\mathcal{A}$ denote a real-world adversary corrupting up to $t < n/2$ parties, and let $\mathcal{S}$ be the ideal-world simulator interacting with $\mathcal{A}$ and emulating the ideal functionalities $\mathcal{F}_{\text{random}}$, $\mathcal{F}_{\text{MPC}}$, $\mathcal{F}_{\text{sort}}$.

In the real-world execution, parties do not interact directly with each other, but only submit inputs to and receive outputs from the invoked ideal functionalities, interleaved with local computations. By definition of the malicious (with-abort) ideal functionalities, the adversary receives the input shares of the corrupted parties and is allowed to choose their output shares. Between invocations of ideal functionalities, the adversary performs local computations whose results determine the inputs to subsequent invocations.

In the ideal world, the simulator $\mathcal{S}$ obtains from $\mathcal{F}_{\text{OSE}}$ the input shares of k and σ corresponding to the corrupted parties. The execution then proceeds by having $\mathcal{S}$ faithfully emulate each invocation of the ideal functionalities. Whenever the adversary outputs a share in a given invocation, the simulator, having knowledge of this value,

can locally compute the corresponding inputs required for the next invocation and provide the adversary with a consistent view. At the end of the execution, $\mathcal{S}$ forwards the adversary's output shares to $\mathcal{F}_{\text{OSE}}$, which computes the corresponding output shares for the honest parties or aborts.

Since the adversary's interaction pattern, received messages, and local computations in the ideal execution are identically distributed to those in the real-world execution, the view of $\mathcal{A}$ in the ideal world is identically distributed to its view in the real world. Thus, the shares of the honest parties are uniform conditioned on the shares of the adversary and the secrets. Consequently, the joint distribution of the adversary's view and the output shares of the honest parties in the ideal world is indistinguishable from that in the real world. Therefore, Protocol 1 securely realizes $\mathcal{F}_{\text{OSE}}$ with abort against malicious adversaries controlling any $t < n/2$ parties. $\square$

THEOREM 7. *Protocol 8 securely realizes $\mathcal{F}_{\text{Join-MM}}$ with abort in the $(\mathcal{F}_{\text{OSE}}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}}, \mathcal{F}_{\text{soPRF}})$ -hybrid model against malicious adversaries controlling any $t < n/2$ parties.*

PROOF. Correctness follows from the same analysis as in Theorem 4.

The security argument and the overall simulation paradigm closely follow the proof of Theorem 6. The ideal-world simulator $\mathcal{S}$ manages the adversary $\mathcal{A}$'s local computations and faithfully emulates the core ideal functionalities $(\mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}})$ exactly as described in the previous proof. Compared to Protocol 1, Protocol 8 additionally invokes two ideal functionalities: $\mathcal{F}_{\text{OSE}}$ and $\mathcal{F}_{\text{soPRF}}$. To simulate these, $\mathcal{S}$ proceeds in the standard manner for *with-abort* functionalities: it perfectly emulates them by extracting the inputs provided by the corrupted parties and allowing $\mathcal{A}$ to dictate the corresponding output shares for the corrupted parties. This maintains a consistent internal state for $\mathcal{A}$. Beyond these additional ideal invocations, Protocol 8 has two public reconstruction steps that do not appear in Protocol 1: the reconstruction of the output size m , and the reconstruction of the encoding vectors $\vec{b}^{\text{enc}}$ and $\vec{p}^{\text{enc}}$ in the subroutine COMPLETEPERM. $\mathcal{S}$ simulates the open step without knowing the honest parties' true inputs.

Specifically, for the reconstruction of m , $\mathcal{S}$ locally computes the corrupted parties' shares of m , denoted as $\llbracket m \rrbracket_{\mathcal{A}}$. This is strictly computable because these shares are derived via local linear operations over the output shares of the APPLYPERM command, which are fully known to $\mathcal{S}$. According to the modified definition of $\mathcal{F}_{\text{Join-MM}}$, the trusted party explicitly hands the actual output size m to the adversary (and thus to $\mathcal{S}$) prior to output delivery. Knowing both m and $\llbracket m \rrbracket_{\mathcal{A}}$, $\mathcal{S}$ samples the honest parties' shares uniformly at random under the strict constraint that, together with $\llbracket m \rrbracket_{\mathcal{A}}$, they perfectly reconstruct the target value m . $\mathcal{S}$ then sends these simulated shares to $\mathcal{A}$. During this simulated reconstruction phase, $\mathcal{S}$ awaits the shares sent back by $\mathcal{A}$. If $\mathcal{A}$ provides inconsistent shares (causing the reconstruction to fail) or explicitly aborts, an honest party in the real world would abort. To mirror this, $\mathcal{S}$ sends an abort command to the trusted party computing $\mathcal{F}_{\text{Join-MM}}$. Otherwise, it sends continue and forwards $\mathcal{A}$'s final output shares to finish the execution.

Next, consider the reconstruction of $\vec{b}^{\text{enc}}$ and $\vec{p}^{\text{enc}}$ in each invocation of the subroutine COMPLETEPERM. Conditioned on the

no-collision event established in the correctness analysis, $\mathcal{S}$ first generates the target opened values exactly as in the semi-honest proof (Theorem 4). To simulate the reconstruction transcripts of these opened encodings, $\mathcal{S}$ locally computes the corrupted parties' shares of $\vec{b}^{\text{enc}}$ and $\vec{p}^{\text{enc}}$, denoted by $\llbracket \vec{b}^{\text{enc}} \rrbracket_{\mathcal{A}}$ and $\llbracket \vec{p}^{\text{enc}} \rrbracket_{\mathcal{A}}$, respectively. These shares are computable by $\mathcal{S}$ because they are obtained from the output shares of the corresponding `APPLYPERM` command and $\mathcal{F}_{\text{soPRF}}$, respectively. Then, $\mathcal{S}$ samples the honest parties' opening shares uniformly at random subject to the constraint that, together with the corrupted parties' shares, they reconstruct the target vectors $\vec{b}^{\text{enc}}$ and $\vec{p}^{\text{enc}}$, respectively. These simulated opening shares are sent to $\mathcal{A}$ as the messages that honest parties would send during reconstruction. Again, if $\mathcal{A}$ sends inconsistent shares, refuses to send shares, or explicitly aborts, then an honest party would abort in the real execution. The simulator therefore sends abort to $\mathcal{F}_{\text{Join-MM}}$. Otherwise, the reconstruction transcripts of $\vec{b}^{\text{enc}}$ and $\vec{p}^{\text{enc}}$ are distributed as in the real execution, and the opened values reveal no information about the secret positions beyond the public size parameters.

The remainder of the simulation proceeds exactly as in Theorem 6. Since all invoked ideal functionalities are emulated with abort, the public reconstruction of m is simulated from the output size provided by $\mathcal{F} * \text{Join-MM}$, and the opened encodings in `COMPLETEPERM` are simulated as described above, the overall distribution of $\mathcal{A}$'s view is identical to that in a real execution. Therefore, these protocol additions preserve the indistinguishability between the real-world and ideal-world views, concluding the proof. $\square$

THEOREM 8. *Protocol 4 securely realizes $\mathcal{F}_{\text{Join-OM}}$ with abort in the $(\mathcal{F}_{\text{OSE}}, \mathcal{F}_{\text{random}}, \mathcal{F}_{\text{MPC}}, \mathcal{F}_{\text{sort}})$ -hybrid model against malicious adversaries controlling any $t < n/2$ parties.*

PROOF. Correctness follows from the same analysis as in Theorem 5. For security, we observe that the structure of Protocol 4 is fundamentally identical to Protocol 1, with the sole exception that it additionally invokes the ideal functionality $\mathcal{F}_{\text{OSE}}$. Notably, unlike Protocol 8, it does not involve any public reconstruction of intermediate values. Therefore, the security seamlessly follows from the simulation paradigm established in Theorem 6. The ideal-world simulator $\mathcal{S}$ proceeds exactly as before, faithfully emulating all invoked ideal functionalities. For the newly introduced $\mathcal{F}_{\text{OSE}}$, $\mathcal{S}$ simply extracts the inputs provided by the corrupted parties and allows the adversary $\mathcal{A}$ to dictate their output shares, consistent with the standard *with-abort* ideal functionality wrapper. Since the entire protocol consists exclusively of local computations interleaved with these perfectly emulated ideal functionalities, $\mathcal{S}$ trivially maintains a consistent internal state for $\mathcal{A}$ without requiring any novel simulation steps. Consequently, the joint distribution of the adversary's view and the honest parties' outputs in the ideal world is identically distributed to that in the real execution. $\square$

E Beyond Joins: Window Aggregates via OSE

We now show that OSE is useful beyond join processing. A window aggregate produces one output value for each input row, rather than collapsing each group into a single output row as in `GROUP BY`. Here we focus on two common cases: whole-partition window

Protocol 13 $\llbracket \vec{w} \rrbracket \leftarrow \text{WINDOWAGGVIAOSE-SORTED}(\llbracket \vec{k}^{\text{sort}} \rrbracket, \llbracket \vec{u} \rrbracket)$

Input: Secret shares of a stably sorted key vector $\llbracket \vec{k}^{\text{sort}} \rrbracket$ and a value vector $\llbracket \vec{u} \rrbracket$ of the same length, represented in the same sorted order.

Output: Secret shares of the window-aggregate vector $\llbracket \vec{w} \rrbracket$.

The protocol: Each party does the following:

- (1) Compute $\llbracket \vec{\sigma} \rrbracket \leftarrow \text{OSE-SORTED}(\llbracket \vec{k}^{\text{sort}} \rrbracket)$.
- (2) Form the auxiliary vector

$$\llbracket \vec{f} \rrbracket \leftarrow \llbracket \vec{u} \rrbracket \parallel [-\vec{u}].$$

- (3) Send $(\text{ApplyPerm}, \llbracket \vec{\sigma} \rrbracket, \llbracket \vec{f} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{f}' \rrbracket$.
 - (4) Compute $\llbracket \vec{h} \rrbracket \leftarrow \text{PREFIXSUM}(\llbracket \vec{f}' \rrbracket)$.
 - (5) Send $(\text{ApplyInv}, \llbracket \vec{\sigma} \rrbracket, \llbracket \vec{h} \rrbracket)$ to $\mathcal{F}_{\text{sort}}$ and receive $\llbracket \vec{g} \rrbracket$.
 - (6) Output the first copy of $\llbracket \vec{g} \rrbracket$ as $\llbracket \vec{w} \rrbracket$.
-

aggregates, where every row in a partition receives the same group-level aggregate, and prefix running window aggregates, where each row receives the aggregate over the prefix of its ordered partition. When the sorted key vector is already available, we use a simplified variant of OSE, denoted by `OSE-SORTED`. It takes the stably sorted key vector $\llbracket \vec{k}^{\text{sort}} \rrbracket$ as input and outputs the stable sorting permutation of $\llbracket \vec{k}^{\text{sort}} \rrbracket \parallel \llbracket \vec{k}^{\text{sort}} \rrbracket$. Compared with OSE, it omits the initial application of $\llbracket \vec{\sigma}^{\text{base}} \rrbracket$ and the final composition step. The key idea is to use `OSE-SORTED` to convert partitioned window computation into a prefix-sum-based propagation with cancellation.

Whole-Partition Window Aggregates. We first consider whole-partition window aggregates, where every row in a partition receives the same group-level aggregate. Our method assumes a sparse representative-row layout, where the aggregate value of each partition is assigned to its first or last row and all other positions are zero. Such a layout may be available directly as a `Group By` [6] intermediate result or obtained by one additional oblivious permutation. Accordingly, in Protocol 13, we instantiate $\llbracket \vec{u} \rrbracket$ as this sparse aggregate vector. The resulting output assigns the aggregate value of each partition to every row in that partition. This directly supports whole-partition `Sum`, `Count`, `Min`, and `Avg` (combining the corresponding sum and count vectors). For `Max`, set $\llbracket \vec{f} \rrbracket \leftarrow -\llbracket \vec{u} \rrbracket \parallel \llbracket \vec{u} \rrbracket$ (line 2), compute suffix sum (line 4) and output the second copy (line 6) in Protocol 13.

Prefix Running Window Aggregates. We next consider a subclass of prefix running window aggregates, where each row receives the aggregate over the prefix of its ordered partition. In Protocol 13, we instantiate $\llbracket \vec{u} \rrbracket$ as the value vector itself in this sorted order. The resulting output yields the desired prefix running aggregates for each partition. This directly supports prefix running `Sum`, `Count`, and `Avg`.