Two Party Secret Shared Joins

Srinivasan Raghuraman[2], Peter Rindal[1], and Harshal Shah[1]

[1] Visa Research
[2] Visa Research and MIT

Abstract. We present concrete techniques for adapting the protocols of Mohassel et al (CCS 2020) and Badrinarayanan et al (CCS 2022) for compute SQL-like querying operations on secret shared database tables to the two party setting. The afore mentioned protocols are presented in a generic setting with access to certain idealized functionalities, e.g. secret shared permutations. However, they only instantiate their protocols in the honest majority three party setting due to other settings being considered too inefficient. We show that this is no longer the case. In particular, the recent work of Peceny et al. (eprint 2024) gives a concretely efficient two party permutation protocol. Additionally, we give a new and highly efficient protocol for evaluating the strong PRF recently proposed by Alamati et al. (Crypto 2024). Building on these advancements, along with a variety of protocol improvements and significant cryptographic engineering, our open source implementation demonstrate concretely efficient two party SQL-like querying functionality on secret shared data. We focus on the two party setting with secret shared input and output tables. The first protocol $\Pi_{\mathsf{Join\text{-}OO}}$ is designed for the setting where the join keys are unique, similar to Private Set Intersection (PSI) except that the inputs and output are secret shared. This protocol is constant round and $O(n)$ running time. The secret protocol $\Pi_{\mathsf{Join\text{-}OM}}$ allows one of the tables to contain repeating join keys. Our instantiations achieves $O(n \log n)$ running time and $O(\log n)$ rounds of interaction.

1 Introduction

In this work we focus on performing *c*omposable SQL-style join operations on secret shared input databases in the semi-honest two-party setting. Given two secret shared database tables, our protocols constructs a secret shared table containing a join of the two tables. These protocols reveal no information about the underlying databases apart from the number of rows they contain and information about the schema.

The first protocol is the most efficient but is restricted to the setting where both tables are being joined on unique primary keys. The advantage of this approach is that the protocol executes in constant round and with linear overhead. The second protocol allows one of the tables to contain duplicate join keys. This added flexibility comes at the cost of a overall running time of $O(n \log n)$ and $O(\log n)$ rounds to join two tables with n rows. Compared to prior works [4,30], our protocols are conceptually very similar with the main innovation being the

ability to efficiently instantiate them in the two party setting, as opposed to the honest majority three party setting. Our innovations are predicated on the recent improvements to two party secure permutations [32].

In recent years, numerous exciting works [10,11,20–23,30,31,33–36,38,40,41] have focused on private computation tasks such as set intersection, union, inner join, secret sharing of set intersections, and related functionalities, showing strong potential for practical deployment. Most of these works target the specialized scenario of private set intersection (PSI), which essentially reveals the full result of an inner join. However, real-world applications often demand a broader set of properties:

– Running any join/functionality without revealing the full result, thus enabling further joins, filtering, or aggregate computations.
– Secret-shared inputs (since data might be the output of a prior MPC computation).
– Support for non-unique keys, which is critical for real-world datasets.

Developing protocols that securely handle these general, composable SQL-like operations—complete with secret-shared inputs, outputs, and potentially repeated join keys—is considerably more challenging, and existing PSI techniques do not straightforwardly extend to this setting.

Our protocols fill this gap by ensuring that input and output data remain secret-shared, which allows them to be chained together. The output of one join can seamlessly serve as the input to the next, supporting sophisticated queries and post-processing. This composability is crucial for practical use cases such as privacy-preserving machine learning [14, 17, 29, 39, 43] , which typically assume the training data is already joined and curated. Without our approach, it remains unclear how to securely perform these intermediate filtering and joining steps on real-world data.

Although earlier work [4] have addressed subsets of these challenges, we are the first to achieve all of these properties in the two-party setting. Doing so required substantial crypto-engineering to integrate diverse protocol components efficiently. Our open-source code reflects years of optimization effort, right down to leveraging SIMD instructions for maximum performance.

By supporting secret-shared inputs, our protocols also enable an outsourced secure-computation model. In this approach, two (or more) non-colluding servers are set up, and any input data—whether from these servers themselves or from external parties—is secret-shared among them, preserving confidentiality. This paradigm has been gaining traction in industry. For example, Mozilla recently deployed a service to collect Firefox telemetry data [42] using two non-colluding servers running the Prio protocol [13]. Additional use cases include privacy-preserving machine learning frameworks such as Facebook's Crypten for PyTorch [17] and Cape Privacy's FTEncrypted for TensorFlow [14], both of which leverage secret-shared inputs.

Database Joins. We assume the two parties have constructed secret shared table X, Y. Let us denote X_i, Y_i as indexing the row i of X, Y respectively. Let $\mathsf{key}(X_i)$ and $\mathsf{key}(Y_i)$ denote the join key being joined on for the ith rows.

The result of a join should contain $(X_i||Y_j)$ for all i, j with matching keys, i.e. $\mathsf{key}(X_i) = \mathsf{key}(Y_j)$. If it is known that $\{\mathsf{key}(X_i)\}_i$ and $\{\mathsf{key}(Y_i)\}_i$ each do not contain duplicates, we say the join is one-to-one. If one table can have duplicate join keys, then the join is one-to-many. Finally, if both tables can contain duplicates its a many-to-many join. In this work to focus on the most common case of obliviously computing joins which have a one-to-one or one-to-many relation. Let's look at works that are closely related to ours.

Our Contributions

Our contributions can be summarized as follows:

- We are the first to present secure protocols for performing SQL-style join operations on secret-shared databases in the two-party semi-honest setting, covering both one-to-one and one-to-many join scenarios.
- Our protocols are extended to support a wide range of SQL-style queries, including aggregate operations such as GROUP BY, AVERAGE, MIN, MAX, and more. This extension allows for complex data analytics to be performed securely on secret-shared data without revealing any sensitive information.
- We develop composable protocols where both the input and output tables are secret-shared, enabling the integration of additional secret-shared computations. This composability allows for executing subqueries and complex query sequences, significantly enhancing the functionality and flexibility of our protocols.
- We implement our protocols in the two-party honest-majority setting and achieve running times competitive with existing n-party protocols. Notably, we can complete a one-to-one join on two tables—each containing one million (2^{20}) rows—in about two minutes.

2 Related Work

The last decade has seen tremendous improvements to our ability to efficiently compute various private database operations such as set intersection, union, inner join, secret-shares of set intersection and related functionalities, which have shown great promise for practical deployment [4, 10, 11, 20–23, 30, 31, 33–36, 38, 40, 41]. The bulk of the focus has been been *private set intersection* (PSI), where there are two parties and each holds a set of identifiers. After the protocols is executed, one of the parties learn the common identifiers and nothing else. While PSI has been applied to numerous problems, stronger security guarantees are sometimes required. For example, PSI is not suitable if the input sets are already secret shared. Moreover, PSI also reveals the intersection in the clear. This prevents post processing the result using additional secure computation techniques. These limitations have been lifted by [4, 30] which can perform various types of join operations with secret shared inputs and output. However, their techniques appeared to not result in efficient protocols in the two party setting.

Ion et al. presented a private set intersection sum protocol [20] that is used by Google Adwords to correlate online advertising with offline sales in a privacy preserving manner. This protocols does not have shared input but supports some computation on the output. Pinkas et al. [34] gives a PSI protocol with plaintext input and secret shared output.

The first protocol to consider the fully secret shared setting is by Blanton et al. [5]. This protocols achieves $\mathcal{O}(n \log n)$ communication and computation complexity, where n denote the number of records. This work only considers one-to-one joins and utilizes an oblivious sorting subroutine.

Laur et al. in [24] present a protocol for database joins/unions in the honest majority setting with secret-shared input databases. Their approach is conceptually simple but leaks the size of the join and only supports one-to-one joins.

Liagouris et al. in [26] proposed a generic MPC-based framework for performing a wide range of SQL queries. Their framework is fully composable and supports duplicates keys. However, their actual protocol is rather primitive in that the join is perform by comparing all $O(n^2)$ pairwise rows between X and Y. On the other hand, their general framework is also compatible with the more optimized join protocols presented here and in the other related works.

The first one-to-one joins protocols with linear overhead and no leakage was presented by [30]. Their protocol is general but at the time was only practical in the three party honest majority setting. Their protocol is composable in that the output of a join can serve as the input for the next join operation enabling a sequence of SQL join computations. Our first protocol can be thought of as an optimized version for the two party setting.

[4] provides the first protocol for handling composable secure joins for one-to-many joins and many-to-many joins. As with [30], their protocols are general but only had efficient instantiations for the three-party honest majority setting. Their one-to-many join protocol leaks no information while their many-to-many join protocol optionally allows a bound on the output table size to be leaked. This is because in general many-to-many joins result in an output table of size $O(n^2)$ which is thought to be impractical for typical values of n. Their many-to-many protocols allows for a upper bound d on the output table size to be revealed and thereby avoiding quadratic overhead. Their protocols are based on sorting and inherit the $O(n \log n)$ running time that sorting implies. Our second protocol heavily relies on their one-to-many join protocol and can be thought of as an optimized two-party version.

Recently, [19] similarly capable composable joins protocols but with a reported running time of $O(n \log^2 n)$. However, this is only due to their choice of a less efficient sorting protocol which can be replaced with the one used here to achieve $O(n \log n)$ time. This work considers the malicious setting but with an three-party honest-majority. Due to the stronger requirements of malicious security their protocol is less efficient than [4].

[28] proposes another composable joins protocol in the semi-honest honest-majority three-party setting. Their protocol is similar to [4]. The critical different being the claim that the sorting subroutine of [4] can be replaced by a custom

group-by operation that can be computed in linear time. While sorting can be replaced, we are not able to verify the security of this new group-by operation and leave it as an open question if their techniques can be used to optimized the two-party protocols we present here.

3 Preliminaries

3.1 Notation.

We use $x := y$ to [re]define the variable x with the value of y. $x = y$ denotes mathematical equality or the bit b which is 1 iff x, y hold the same value. Let $\cdot$ denote the dot product and $\odot$ denote component-wise multiplication. For binary operator $\star \in \{+, -, \cdot, \odot\}$, let $x \star_p y$ denote $(x \star y \mod p)$. Let $[m, n]$ denote the "inclusive" range $\{m, m+1, 2, ..., n\}$ and $[n]$ as shorthand for $[1, n]$. We also define $(m, n]$ as the "left-exclusive" range $\{m+1, 2, ..., n\}$, $[m, n)$ as the "right-exclusive" $\{m, m+1, 2, ..., n-1\}$ and $(n], [n)$ as the shorthands for $(1, n], [1, n)$, respectively. Let V be a vector with elements $V = (V_1, ..., V_n)$. A subvector can be indexed using $V_{[m,n]}$ to denote $(V_m, ..., V_n)$. For a matrix or table M we denote the ith row as M_i and the jth column as $M_{*,j}$. The element in row i and column j is indexed as $M_{i,j}$. We also denote a submatrix by sub-scripting it with the row/column set. Let $\|$ and $//$ denote the horizontal and vertical concatenation of two matrices, respectively.

Typically X, Y will refer to the input tables that are being joined. We use n to represent the number of rows a table has, which for simplicity will assume to be the same for X, Y. When performing a join, a specific column will be used as the join key column. We will denote the column as $\mathsf{key}(X), \mathsf{key}(Y)$ to denote the respective join columns while $\mathsf{key}(X_i), \mathsf{key}(Y_i)$ denotes the join key element of the ith row, respectively. Additionally, each row of a table will contain a special secret shared flag denoting whether it is null. Null rows rows are place holder rows used to hide cardinality of the tables, i.e. padding. We use $\mathsf{IsNull}(X)$ to refer to the column of null flags and $\mathsf{IsNull}(X_i)$ to refer to a bit that is 1 if the ith row is null. We use $X_i := \texttt{Null}$ to denote that X_i should be assign the row of zeros with the null bit set to 1.

We define a permutation of size m as an bijective function $\pi : [m] \rightarrow [m]$. We extend this definition such that when π is applied to a vector V of m elements, then $\pi(V) = (V_{\pi(1)}, ..., V_{\pi(m)})$. Parties are referred to as P_0, P_1. We use κ to denote the computational security parameter, e.g. $\kappa = 128$, and λ as the statistical security parameters, e.g. $\lambda = 40$.

3.2 Secure Computation Framework

We make use of three types of secret shares: binary shares denoted as $[\![x]\!]$, and a permutation sharing $(\![\pi]\!)$ discussed below. The share held by party P_i is denoted as $[\![x]\!]_i, (\![\pi]\!)$. The bracket notations, e.g. $[\![x]\!]$ denotes that distribution that party P_0 holds a random share, e.g. $[\![x]\!]_0 \in F_2$, while party P_1 holds another share, e.g.

Functionality $\mathcal{F}_{\mathsf{MPC}}(\mathcal{C}, [\![X]\!])$:

Return to the parties the secret sharing $\mathsf{share}(\mathcal{C}(x))$.

Fig. 1. Secure Computation ideal functionality $\mathcal{F}_{\mathsf{MPC}}$

$[\![x]\!]_1 \in F_2$, such that the shares add to the underlying value, e.g. $x = [\![x]\!]_0 \oplus [\![x]\!]_1$. We extend this notation to vectors and other structured objects in the natural way, e.g. $[\![V]\!]$ is the sharing of vector $V \in G^n$ for $G = \{0,1\}^m$.

We define $\mathcal{F}_{\mathsf{MPC}}$ in Figure 1 which takes a circuit $\mathcal{C}$ and secret shares $[\![x]\!]_i$ as inputs, and outputs new shares $[\![y]\!]_i$, where y is the output of the circuit $\mathcal{C}$ applied to the reconstructed input x. We assume the functionality can take any type of secret share input. Our implementation realizes $\mathcal{F}_{\mathsf{MPC}}$ using the GMW [18] protocol with silent OT prepressing [9, 37]. Each AND gate in the circuit will consume two random OT instances which can be efficiently generated with an amortized communication of less than one bit per OT.

3.3 Functionalities

We describe our protocols in the hybrid model where we assume the presence of idealized third parties called functionalities. Each functionality performs a predefined operation on inputs from the real parties. A functionality is denoted with $\mathcal{F}$ with subscript containing its name. A real implementation would then implement these functionalities using a concrete protocol, i.e. no trusted third party. We refer to [27] for a more detailed description.

For a secret shared $[\![x]\!]$, we use the shorthand $\mathcal{F}([\![x]\!])$ to denote invoking $\mathcal{F}$ on input $[\![x]\!]_i$ from P_i. If a specific party, say P_0, additionally provides private input y to $\mathcal{F}$, we denote this as $\mathcal{F}([\![x]\!], P_0 : y)$. Finally, public input z is denoted as $\mathcal{F}([\![x]\!], P_0 : y, z)$. When shares are input to a functionality, the functionality gains access to the underlying values, e.g. x, via the reconstruction procedure. Similarly, when a functionality say to output a sharing $[\![x]\!]$, we mean $[\![x]\!]_i$ is output to party P_i.

3.4 Secret-Shared Aggregation Trees

[4] and concurrently in [44], presents a useful functionality $\mathcal{F}_{\mathsf{Agg}}$ called an aggregation tree or segmented prefix sum. This functionality takes as input a shared list X and a shared bit vector B. The list is logically divided up into blocks with the start of a block being denoted by $B_i = 0$. For each block, the functionality will independently apply a prefix sum to the block for some associative sum operator $\star$. For example, if a block begins at i and is of size 3, then the output Z will contain $Z_i := X_i, Z_{i+1} := X_i \star X_{i+1}, Z_{i+2} := X_i \star X_{i+1} \star X_{i+2}$. The operator $\star$ can be any associative operator. [4, 44] gives a protocol for implementing this that takes $O(n \cdot \star_{\mathsf{time}})$ time and $O(\log n \cdot \star_{\mathsf{rounds}})$ rounds, where $\star_{\mathsf{time}}, \star_{\mathsf{rounds}}$ are the time and round complexity of computing the $\star$ circuit. We will make use of a

duplication tree where $\star$ is defined as $\star(x_1, x_2) := x_1$. That is, it simply returns the first argument and the overall effect is that the first element of each block is copied into the rest of the block.

> **Functionality** $\mathcal{F}_{\mathsf{Agg}}([X], [B], \star)$:
>
> Upon input vector $[X] \in D^n$ and control bits $[B] \in \{0,1\}^n$ and associative operator $\star$ from the parties, define $\mathsf{pre\text{-}ind}(i) \in [i]$ to be the maximum value such that $B_{\mathsf{pre\text{-}ind}(i)} = 0$. Output $[X'] \leftarrow \mathsf{share}(X')$ where $X'_i :=^{i}_{j=\mathsf{pre\text{-}ind}(i)} X_j$.

Fig. 2. Functionality $\mathcal{F}_{\mathsf{Agg}}$ for secret-shared aggregation.

3.5 Secret-Shared Permutations

In our protocols we need two permutation functionalities, $\mathcal{F}_{\mathsf{Select}}$ in Figure 3 and $\mathcal{F}_{\mathsf{Perm}}$ in Figure 4. The former takes as input an arbitrary function $\sigma : [m] \to [n]$ from one party and a secret shared vector $[\![X]\!]$ from both, where X is size n. It outputs $[\![X_{\sigma(1)}, ..., X_{\sigma(m)}]\!]$. Typically σ will be a permutation but in general σ can be any function with prescribed domain and range.

The second functionality, $\mathcal{F}_{\mathsf{Perm}}$, takes an input a secret shared permutation $(\![\pi]\!)$ from the parties as well as a secret shared vector $[\![X]\!]$. It permutes the vector X by the permutation π and returns a sharing of the result. The notation $(\![\pi]\!)$ is referred to as a secret shared permutation and denotes that P_0 holds permutation $(\![\pi]\!)_0 \in ([n] \to [n])$ while P_1 holds $(\![\pi]\!)_1 \in ([n] \to [n])$ such that $\pi = (\![\pi]\!)_1 \circ (\![\pi]\!)_0$. That is, they each hold a permutation such that if you first permute X by $(\![\pi]\!)_0$ and then by $(\![\pi]\!)_1$, the effect is permuting X by π, i.e. $\pi(X) = (\![\pi]\!)_1((\![\pi]\!)_0(X))$. Individually, each permutation share $(\![\pi]\!)_p$ looks like a random permutation and therefore does not reveal anything about π. We refer to [32] for more details.

> **Functionality** $\mathcal{F}_{\mathsf{Select}}(p, P_p : \sigma, [X])$:
>
> Upon input of selection function $\pi : [m] \to [n]$ from P_p and shared input vector $[X] \in D^n$, output $[Y'] \leftarrow \mathsf{share}(Y')$ where $Y_i := X_{\sigma(i)}$ for $i \in [m]$.

Fig. 3. Functionality $\mathcal{F}_{\mathsf{Select}}$ for secret-shared selection.

We also make use of a sorting functionality $\mathcal{F}_{\mathsf{Sort}}$ as illustrated in Figure 5. The sorting functionality outputs a secret shared permutation $(\![\pi]\!)$ that can be applied to a secret shared vector to permute the vector using $\mathcal{F}_{\mathsf{Perm}}$ functionality as presented in Figure 4. One could implement $\mathcal{F}_{\mathsf{Sort}}, \mathcal{F}_{\mathsf{Perm}}$ using $\mathcal{F}_{\mathsf{MPC}}$ but often there are more efficient protocols, e.g. [12, 32].

Functionality $\mathcal{F}_{\mathsf{Perm}}(\langle\!\langle\pi\rangle\!\rangle, [\![X]\!])$:

Upon shared input permutation $\pi : [n] \to [n]$ and shared input vector $[\![X]\!] \in D^n$, output $[\![Y']\!] \leftarrow \mathsf{share}(Y')$ where $Y_i := X_{\pi(i)}$ for $i \in [m]$.

Fig. 4. Functionality $\mathcal{F}_{\mathsf{Perm}}$ for secret-shared permutation.

Functionality $\mathcal{F}_{\mathsf{Sort}}([\![X]\!])$:

Upon shared input vector X of length n, let $\pi : [n] \to [n]$ be the stable sorting permutation of X. Output $\langle\!\langle\pi\rangle\!\rangle \leftarrow \mathsf{permShare}(\pi)$.

Fig. 5. Secret-shared Sorting ideal functionality $\mathcal{F}_{\mathsf{sort}}$.

3.6 SQL-like Join Functionality

In this paper, we describe the protocol to obliviously compute the inner join of two tables X, Y. The inner join between X, Y is defined as the table Z where Z contains $\bigcup_k \{X_i \mid \mathsf{key}(X_i) = k\} \times \{Y_i \mid \mathsf{key}(Y_i) = k\}$. The ideal functionality $\mathcal{F}_{\mathsf{Join}}$ for secret shared joins in given in Figure 6. It takes as input two secret shared tables as well as public customization functions that determine how to pad the resulting table and the order it should be returned in.

Our protocols naturally support other types of join operations, left join, full join, union, etc. We refer the reader to [30] for a detailed description on how to perform these as well as other operations such as aggregations (sum , count, max value) and filtering using the where clause.

Functionality $\mathcal{F}_{\mathsf{Join}}([\![X]\!], [\![Y]\!], \mathsf{pad}, \mathsf{ordering})$:
Upon input tables, X, Y and customization functions $\mathsf{pad}, \mathsf{ordering}$, perform the following:

1. For $i \in [n]$, and each $j \in [n]$ such that $(\mathsf{key}(X_j) = \mathsf{key}(Y_i)) \wedge (\mathsf{isNull}(Y_i) = \textsc{False}) \wedge (\mathsf{isNull}(X_j) = \textsc{False})$, set $Z_d := (X_j || Y_i)$ and then $d = d + 1$.
2. Add dummy/null rows to Z until it has $D := \mathsf{pad}(d)$ rows.
3. Let $\pi := \mathsf{ordering}(X, Y)$ be a permutation of D items and permute Z as $Z := \pi(Z)$.
4. Output $[\![Z]\!] \leftarrow \mathsf{share}(Z)$.

Fig. 6. Functionality $\mathcal{F}_{\mathsf{Join}}$ for secret shared Join for tables X and Y.

3.7 Cuckoo Hash Tables

The core data structure that our protocols employ is a cuckoo hash table which is parameterized by a capacity n, two (or more) hash functions h_0, h_1 and a vector T which has $m = O(n)$ slots, $T_1, ..., T_m$. For any x that has been added

to the hash table, there is an invariant that x will be located at $T_{h_0(x)}$ or $T_{h_1(x)}$. Testing if an x is in the hash table therefore only requires inspecting these two locations. x is added to the hash table by inserting x into slot $T_{h_i(x)}$ where $i \in \{0,1\}$ is picked at random. If there is an existing item at this slot, the old item y is removed and reinserted at its other hash function location. Given a hash table with $m \approx 1.3n$ slots and three hash functions, then with overwhelming probability n items can be inserted using $O(n)$ insertions [15]. For technical reasons we require $h_i(x) \neq h_j(x)$ for all x and $i \neq j$. This can be achieved by defining $h_j(x)$ over the range $[m] \backslash \{h_i(x)\}_{i<j}$.

4 One-to-one Joins $\mathit{\Pi}_{\mathsf{Join\text{-}OO}}$

4.1 Overview

In [30], the authors presented an $O(n)$ computation and communication protocol $\mathit{\Pi}_{\mathsf{Join\text{-}OO}}$ in constant-rounds to compute joins over secret-shared input database tables. First, we review the join algorithm without any privacy and then we will discuss how this translates to the secret shared setting. Figure 7 depicts the algorithm with the following phases:

1. Construct a cuckoo hash table T containing the rows of Y based on the join keys. That is, row Y_i is inserted at T_j for some $j \in \{h_0(\mathsf{key}(Y_i)), h_1(\mathsf{key}(Y_i))\}$.
2. To determine the matching rows of X, each X_i needs to be compared with the rows T_j for $j \in \{h_0(\mathsf{key}(X_i)), h_1(\mathsf{key}(X_i))\}$. This comparison is facilitated by mapping these T rows to row X_i. That is, $T_{h_0(\mathsf{key}(X_i))}$ is mapped to a new row S_i^0 and $T_{h_1(\mathsf{key}(X_i))}$ to S_i^1.
3. If there is a matching row Y_j for some X_i, then this Y_j row will be have been copied to either S_i^0 or S_i^1. As such, X_i can be compared with S_i^0 and S_i^1. If there is a match, then the output row Z_i is constructed as X_i concatenated with the matching S_i^j. Otherwise Z_i is assigned `Null`.

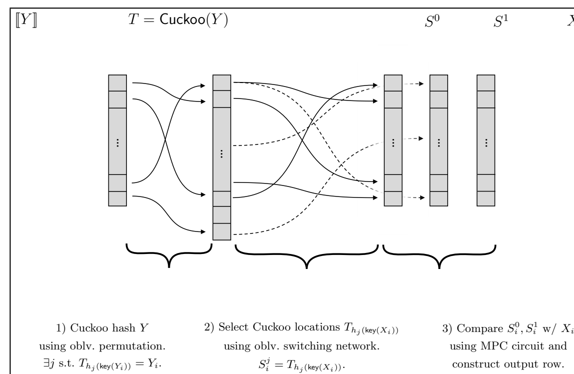


Fig. 7. Overview of the one-to-one join protocol $\mathit{\Pi}_{\mathsf{Join\text{-}OO}}$, diagram from [30].

Given that the input tables are fully secret shared, its not immediately oblivious how to translate this algorithm into a secure protocol. The main challenges are that the rows of X and Y need to be rearranged in an input dependent order. Moreover, this order is determined by the cuckoo hashing algorithm which is not conducive to being run efficiently in MPC. [30] gives a cleaver transformation that avoids these expensive operations.

First, [30] observes that since the join keys are unique, if one first applies a keyed hash to each join keys, then the resulting values for each table individually will be uniformly distributed. As such, party P_0 could learn the hash values for, say, Y while P_1 learns them for X. Party P_0 with the hashes for Y can then run the plaintext cuckoo hashing algorithm to construct T using these hashes. A secret sharing of T can then be constructed by permuting the shares of Y by some plaintext permutation held by P_0, i.e. by invoking $\mathcal{F}_{\mathsf{Select}}$.

Similarly, P_1 can then assign the correct values out of T to S_i^0 and S_i^1 based on the hashes they obtained for X. Again, this assignment can be achieved using the $\mathcal{F}_{\mathsf{Select}}$ functionality where P_1 inputs the plaintext select function while both parties input the shares of T. The final set is to simply compare X_i and S_i^0, S_i^1 using the generic $\mathcal{F}_{\mathsf{MPC}}$ functionality.

[30] suggest implementing the keyed hash using the LowMC [3] block cipher. However, very recently [1] proposed a very efficient MPC friendly PRF which we choose to use instead. The one challenge to using this PRF is that [1] only gave MPC protocols for the setting with plaintext inputs. We address this next by expending their protocols to our settings where the inputs and outputs are secret sharing. Another difference when comparing to [30] is that that our protocol does not match use of a so called *oblivious switching network*. This was a 6 round protocol that implemented the $\mathcal{F}_{\mathsf{Select}}$ functionality in the three party setting. We observe that this can be replaced with a single round permutation protocol of [32].

4.2 Implementing the PRF

The recent work of [1] presents a highly efficient PRF based on the alternating moduli class of assumptions. Essentially, this construction takes as input a key $k \in F_2^n$, a value $x \in F_2^d$ and multiples them by various matrices modulo two different primes, i.e. 2 and 3. More concretely, their strong PRF is defined as

$$F(k, x) := \mathbf{B} \cdot_2 (\mathbf{A} \cdot_3 [k \odot_2 (\mathbf{G} \cdot_2 [x||1])])$$

where $\mathbf{G} \in F_2^{n \times d+1}, \mathbf{A} \in F_3^{m \times n}, \mathbf{B} \in F_2^{t \times m}$ are uniformly distributed. In particular, we make use of the parameterization $d = \kappa, n = 4\kappa, m = 2\kappa, t = \kappa$ which implies $\mathbf{G}$ is an expanding matrix, $\mathbf{A}$ is a square matrix, while $\mathbf{B}$ is a compressing matrix. Without the alternating moduli this function is trivial to invert. However, by switching moduli between subsequent matrix multiplication, the overall function becomes highly nonlinear when viewed with over F_2 or F_3. [1, 2, 8, 16] provide various evidence that this function is thought to be a strong PRF.

While [1] proposed several efficient protocols, these all target the evaluation of their weak PRF construction. That is,

$$F'(k,x) := \mathbf{B} \cdot_2 (\mathbf{A} \cdot_3 [k \odot_2 x]).$$

This simplified function is assumed to be a weak PRF in that result need only looks random if x is chosen uniformly at random. For many application this suffices, however, we will require the result to look random when x is chosen arbitrarily. Moreover, their protocols also assumes x is known to P_0 and k is known to P_1. We fill these gaps by extending their protocols to support their full strong PRF construction with secret shared inputs.

Their protocol for the weak PRF F' works by first computing a secret sharing $[\![w]\!]$ over F_3^n where $w = k \odot x$. The parties can then locally apply the linear map $\mathbf{A}$, i.e. $v := \mathbf{A} \cdot_3 w \in F_3^m$. They then use a specialized protocol for computing a sharing $[\![u]\!]$ such that $u = v \mod 2$. Finally, the parties locally applying the linear map $\mathbf{B}$ to obtain the a sharing of final result $F'(k,x)$. We will reuse the bulk of their protocol but change the computation of $[\![w]\!]$ such that the protocol takes as input the shares of k and x and computes a F_3 sharing of $w = k \odot_2 (\mathbf{G} \cdot_2 [x||1])$.

Starting with a F_2^d sharing of x, the parties can locally compute $[\![x']\!] := \mathbf{G} \cdot_2 [\![x]\!]$. The core challenge is then to compute shares of $k \odot x'$. For our applications, we will evaluate F on many x's and a fixed key k. We decompose this task into two subprotocols, key multiplication and corrections.

Key multiplication. As a building block, we fix some $p \in \{0,1\}$ and begin with the restriction $[\![x']\!]_{\overline{p}} = 0$ and therefore $x' = [\![x']\!]_p$ is held in the clear by P_p, while k remains secret shared and unknown to either party. For $i \in [n]$, the parties will generate an OT with random F_3 messages $h_{i,0}, h_{i,1} \in F_3$, where $P_{\overline{p}}$ will use their key share bit $[\![k_i]\!]_{\overline{p}}$ as the choice bits to learn $h_{i,[k_i]_{\overline{p}}}$. For $i \in [n]$, P_p can compute

$$\begin{aligned}
[\![v_i]\!]_p &:= [\![x'_i]\!]_p [\![k_i]\!]_p -_3 h_{i,0} \\
\delta_i &:= h_{i,0} -_3 h_{i,1} +_3 (1 - 2[\![k_i]\!]_p)[\![x'_i]\!]_p
\end{aligned}$$

and send δ to $P_{\overline{p}}$ who computes

$$[\![v_i]\!]_{\overline{p}} := h_{i,[k_i]_{\overline{p}}} +_3 [\![k_i]\!]_{\overline{p}} \delta_i.$$

Observe that if $[\![k_i]\!]_{\overline{p}} = 0$ then

$$\begin{aligned}
[\![v_i]\!]_p + [\![v_i]\!]_{\overline{p}} =& [\![x'_i]\!]_p [\![k_i]\!]_p - h_{i,0} + h_{i,[k_i]_{\overline{p}}} + [\![k_i]\!]_{\overline{p}} \delta_i \\
=& x'_i [\![k_i]\!]_p - h_{i,0} + h_{i,[k_i]_{\overline{p}}} + [\![k_i]\!]_{\overline{p}} (h_{i,0} - h_{i,1} + (1 - 2[\![k_i]\!]_p) x'_i) \\
=& x'_i [\![k_i]\!]_p \\
=& x'_i k_i
\end{aligned}$$

and otherwise

$$
\begin{aligned}
=& x'_i[\![k_i]\!]_p - h_{i,0} + h_{i,1} + h_{i,0} - h_{i,1} + (1 - 2[\![k_i]\!]_p)x'_i \\
=& x'_i[\![k_i]\!]_p + (1 - 2[\![k_i]\!]_p)x'_i \\
=& x'_i(1 - [\![k_i]\!]_p) \\
=& x'_i k_i
\end{aligned}
$$

Correction. Given that this protocol works when one party holds x' in the clear, a nature suggestion would be to run it twice, for compute shares of $[\![x']\!]_0[\![k]\!]$, then shares of $[\![x']\!]_1[\![k]\!]$ and add them together. However, due to the result being a sharing over F_3 while x' is shared over F_2, the result is $([\![x']\!]_0 +_3 [\![x']\!]_1)[\![k]\!]$ which when both ares are 1 results in a sharing of $2[\![k]\!]$ as opposed to zero. We correcting this discrepancy by multiplying bits $[\![x'_i]\!]_0$ and $[\![x'_i]\!]_1$ to obtains a F_3 sharing and then subtracting off two times the result.

Full PRF Protocol. We are now ready to describe the full protocol for computing the Strong PRF $F(k, x)$ of [1] with fully secret shared input and output. The formal description is in Figure 8. In a small preprocessing, an OTs is performed for each bit of $[\![k]\!]_0$ and $[\![k]\!]_1$ as the choice bit. The main phase beings by expanding the input $[\![x]\!]$ into its expanded form $[\![x']\!] = \mathbf{G} \cdot_2 [[\![x]\!]||1]$. The preprocessed key OTs are then derandomized to perform the computation

$$[\![w_p]\!] = [\![k]\!]_p \odot [\![x']\!]$$

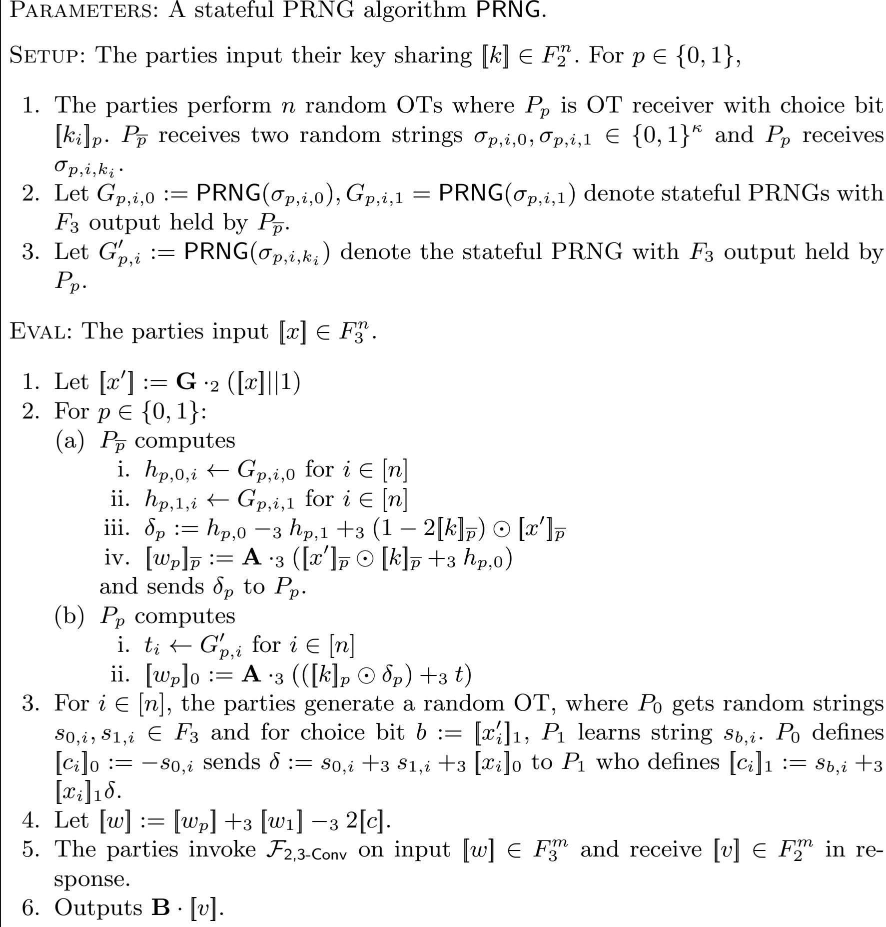
for $p \in \{0, 1\}$. When summed with two times the correction term $[\![c]\!] = [\![x]\!]_0 \odot [\![x]\!]_1$ and then multiplying by $\mathbf{A}$ results in a secret sharing $[\![w]\!] = \mathbf{A}([\![x']\!] \odot [\![k]\!])$. The remainder of the protocol follows the same structure as specified for the weak PRF of [1]. In particular, modulus conversion from F_3 to F_2 is performed to obtain $[\![v]\!] = [\![w]\!] \mod 2$, followed by compression by $\mathbf{B}$.

Security. One can construct a simulator for this protocol using standard techniques in a relatively mechanical. Privacy follows from the fact that the OT receiver only having one out of two OTs message which allows the values associated with the other message to be pseudo random.

4.3 One-to-One Join Protocol $\Pi_{\textsf{Join-OO}}$ Details.

The full protocol is specified in Figure 9. The overview of the protocol is given above. Compared to the overview the full protocol deals with the added complexity of more than two hash functions and the case of null input rows. Nullity is handled by replacing the key for any null row with some large public random value. This ensures that with overwhelming probability the null row will have a unique random.

Theorem 1. *Conditions on X, Y having unique join keys, $\Pi_{\textsf{Join-OO}}$ realizes the $\mathcal{F}_{\textsf{Join}}$ functionality in the semi-honest two-party hybrid setting with $\textsf{pad}(d) = |X|$ and $\textsf{ordering}$ returning the permutation that orders according to X.*



PARAMETERS: A stateful PRNG algorithm PRNG.

SETUP: The parties input their key sharing $[\![k]\!] \in F_2^n$. For $p \in \{0,1\}$,

1. The parties perform n random OTs where P_p is OT receiver with choice bit $[\![k_i]\!]_p$. $P_{\overline{p}}$ receives two random strings $\sigma_{p,i,0}, \sigma_{p,i,1} \in \{0,1\}^\kappa$ and P_p receives σ_{p,i,k_i}.
2. Let $G_{p,i,0} := \mathsf{PRNG}(\sigma_{p,i,0}), G_{p,i,1} = \mathsf{PRNG}(\sigma_{p,i,1})$ denote stateful PRNGs with F_3 output held by $P_{\overline{p}}$.
3. Let $G'_{p,i} := \mathsf{PRNG}(\sigma_{p,i,k_i})$ denote the stateful PRNG with F_3 output held by P_p.

EVAL: The parties input $[\![x]\!] \in F_3^n$.

1. Let $[\![x']\!] := \mathbf{G} \cdot_2 ([\![x]\!]||1)$
2. For $p \in \{0,1\}$:
   (a) $P_{\overline{p}}$ computes
      i. $h_{p,0,i} \leftarrow G_{p,i,0}$ for $i \in [n]$
      ii. $h_{p,1,i} \leftarrow G_{p,i,1}$ for $i \in [n]$
      iii. $\delta_p := h_{p,0} -_3 h_{p,1} +_3 (1 - 2[\![k]\!]_{\overline{p}}) \odot [\![x']\!]_{\overline{p}}$
      iv. $[\![w_p]\!]_{\overline{p}} := \mathbf{A} \cdot_3 ([\![x']\!]_{\overline{p}} \odot [\![k]\!]_{\overline{p}} +_3 h_{p,0})$
      and sends δ_p to P_p.
   (b) P_p computes
      i. $t_i \leftarrow G'_{p,i}$ for $i \in [n]$
      ii. $[\![w_p]\!]_0 := \mathbf{A} \cdot_3 (([\![k]\!]_p \odot \delta_p) +_3 t)$
3. For $i \in [n]$, the parties generate a random OT, where P_0 gets random strings $s_{0,i}, s_{1,i} \in F_3$ and for choice bit $b := [\![x'_i]\!]_1$, P_1 learns string $s_{b,i}$. P_0 defines $[\![c_i]\!]_0 := -s_{0,i}$ sends $\delta := s_{0,i} +_3 s_{1,i} +_3 [\![x_i]\!]_0$ to P_1 who defines $[\![c_i]\!]_1 := s_{b,i} +_3 [\![x_i]\!]_1 \delta$.
4. Let $[\![w]\!] := [\![w_p]\!] +_3 [\![w_1]\!] -_3 2[\![c]\!]$.
5. The parties invoke $\mathcal{F}_{2,3\text{-}\mathsf{Conv}}$ on input $[\![w]\!] \in F_3^m$ and receive $[\![v]\!] \in F_2^m$ in response.
6. Outputs $\mathbf{B} \cdot [\![v]\!]$.

Fig. 8. OT based protocol for the [1] Strong PRF with shared input/output.

Proof (proof sketch).

The simulation of these protocols directly follow from the composibility of the subroutines $\mathcal{F}_{\mathsf{PRF}}$, $\mathcal{F}_{\mathsf{Perm}}$ and $\mathcal{F}_{\mathsf{MPC}}$. First, the output of $\mathcal{F}_{\mathsf{PRF}}$ simply outputs random strings and it is therefore straightforward to simulate. This follows from the fact that the input is unique except for possible the null rows. These are however assigned random keys and therefore the probability of a collision is negligible. $\mathcal{F}_{\mathsf{Perm}}$ and $\mathcal{F}_{\mathsf{MPC}}$ both output secret shared values and therefore are trivial to simulate. Finally, correctness is straight forward to analysis and holds so long as there is no encoding collisions and cuckoo hashing succeeds. Parameters are chosen appropriately so these failure events happen with probability at most $2^{-\lambda}$.

Protocol $\Pi_{\mathsf{Join\text{-}OO}}([X],[Y])$:

1. **[PRF]** The parties sample a shared PRF key $[k]$ along with random public matrices $r, r' \in \{0,1\}^{n\times\kappa}$. For $i \in [n]$, parties invoke $\mathcal{F}_{\mathsf{prf}}([k], [\mathsf{key}(X_i)] \oplus r_i \cdot \mathsf{IsNull}([X_i])), \mathcal{F}_{\mathsf{prf}}([k], [\mathsf{key}(Y_i)] \oplus r'_i \cdot \mathsf{IsNull}([Y_i]))$ and receive $[E_{\mathsf{X},i}], [E_{\mathsf{Y},i}]$ in response. $[E_{\mathsf{X},i}]$ is revealed to P_0 and $[E_{\mathsf{Y},i}]$ is revealed to P_1.
2. **[Build Cuckoo]** P_1 samples cuckoo hash functions $h_1, ..., h_w$ and constructs a cuckoo hash table t for the set E_{Y} s.t. for some $j \in \{h_1(E_{\mathsf{Y},i}), ..., h_w(E_{\mathsf{Y},i})\}$, $E_{\mathsf{Y},i} = t_j$. P_1 defines π_0 such that $\pi_0(j) = i$ where $E_{\mathsf{Y},i} = t_j$. The parties invoke $\mathcal{F}_{\mathsf{Select}}(\pi_0, [Y])$ and receive $[T]$ in response. Note, π_0 is the private input of P_1.
3. **[Select Cuckoo]** P_0 defines $\pi_1, ..., \pi_w$ such that $\pi_l(i) = j$ where $h_l(E_{\mathsf{X},i}) = j$. For $l \in [w]$, the parties invoke $\mathcal{F}_{\mathsf{Select}}(\pi_l, [T]||[E_{\mathsf{Y}}])$ and receive $[S^l]||[E^l_{\mathsf{S}}]$ in response.
4. **[Compare]** For $i \in n$, if $\exists j \in [w]$ s.t. $[\mathsf{IsNull}(S^j_i)] = 0$ and $[E_{\mathsf{X},i}] = [E^l_{\mathsf{S},i}]$ then $[Z_i] := [X_i]||[S^j_i]$. Otherwise $[Z_i] := \texttt{Null}$.
5. **[Output]** Return $[Z]$.

Fig. 9. Join protocols $\Pi_{\mathsf{join\text{-}oo}}$ for one-to-one relations.

5 One to Many Join - $\Pi_{\mathsf{Join\text{-}OM}}$

Overview The core of this protocol was first described in [4]. Compared to $\Pi_{\mathsf{Join\text{-}OO}}$, it relaxes the restriction that the joins keys are unique by allowing one table, denoted as Y, to have duplicates in the join-key column. The other table, X, must have unique join-keys. The protocol first combines the two tables as

$$Z := \begin{bmatrix} \mathsf{key}(X) & X & 0 \\ \mathsf{key}(Y) & 0 & Y \end{bmatrix}$$

The protocol stable sorts the rows of Z by the join key. The row X_i will therefore appear before the matching rows from Y. For example, if X_i matches rows $Y_{j_1}, ..., Y_{j_t}$ then at some position ℓ will appear the row $Z_\ell = [\mathsf{key}(X_i), \quad X_i, \quad 0]$ immediately followed by $Z_{\ell+k} = [\mathsf{key}(Y_{j_k}), \quad 0, \quad Y_{j_k}]$ for $k \in [t]$. The core task of

the protocol is to then construct the pair $[X_i, Y_{j_1}], ..., [X_i, Y_{j_t}]$ from this sequence. In the plaintext setting this can simply be accomplished by copying the X_i value into the next t rows.

The first task to achieve this in the secret shared setting is to determine which rows should be copied into the next. We achieve this by comparing the key for the current row Z_ℓ with the join key for the next row $Z_{\ell+1}$. If they are equal then the Z_ℓ row's X columns should be copied into $Z_{\ell+1}$.

As described this strategy might appear to require $O(n)$ rounds of interaction as the prior row must be copied before the next row can be copied. However, the technique known as an aggregation tree [4] gives a $O(\log n)$ rounds protocol for copying all of these values. Finally, the output table is obtained by unpermuting the rows of Z by the sorting permutation and returning the last n rows.

Comparison. Compared to the prior work [4], the differences of our protocol is largely just the replacement of the implementation of the ideal functionalities.

One-to-Many Join Protocol $\Pi_{\mathsf{Join\text{-}OM}}$ Details. The full protocol is specified in Figure 10. The overview of the protocol is given above. Compared to the overview the full protocol deals with the added complexity of more than two hash functions and the case of null input rows. Nullity is handled by replacing the key for any null row with some large public random value. This ensures that with overwhelming probability the null row will have a unique random.

Theorem 2. *Conditions on X having unique join keys, $\Pi_{\mathsf{Join\text{-}OM}}$ realizes the $\mathcal{F}_{\mathsf{Join}}$ functionality in the semi-honest two-party hybrid setting with $\mathsf{pad}(d) = |Y|$ and $\mathsf{ordering}$ returning the permutation that orders according to Y.*

Proof (proof sketch).

The simulation of these protocols directly follow from the composibility of the subroutines $\mathcal{F}_{\mathsf{Perm}}$,$\mathcal{F}_{\mathsf{Sort}}$ and $\mathcal{F}_{\mathsf{MPC}}$. Correctness is straight forward to analysis.

6 Group-by and Reduce

Let's assume that after the join operation, the resulting table formed is X, which is secret shared between the parties. The resulting table X can be grouped by a specified column using the group-by function in SQL. Group-by functionality can be applied using $\mathcal{F}_{\mathsf{Sort}}$ on the group-by Columns. Once the table is grouped, aggregate functions such as SUM, COUNT, MAX, etc., can be applied to each group. These functions compute a single value for each group based on the values within the group. The full protocol is described in Figure 12. The ideal functionality $\mathcal{F}_{\mathsf{GroupBy}}$ for secret-shared group-by-and-reduce is described in Figure 11. It takes as input a secret-shared table X, an aggregation operator $\star$, and a group-by column K, and outputs a secret-shared table Z containing the aggregated results. Because Z is secret-shared, subsequent operations—such as

Protocol $\Pi_{\mathsf{Join\text{-}OM}}([\![X]\!],[\![Y]\!])$:

1. **[Sort keys]** Let $[\![K]\!] = \mathsf{key}([\![X]\!])//\mathsf{key}([\![Y]\!])$. The parties invoke $(\!\![\pi]\!\!) := \mathcal{F}_{\mathsf{Sort}}([\![K]\!])$.
2. **[Dummy rows]** Locally, the parties prepend n `Null` rows to $[\![X]\!]$ to obtain $[\![X']\!]$.
3. **[Permute X]** The parties invoke $[\![X''||K'']\!] := \mathcal{F}_{\mathsf{Perm}}((\!\![\pi]\!\!), [\![X']\!]||[\![K]\!])$.
4. **[Control bits]** Let $[\![\beta_1]\!] := [\![0]\!]$. For $i \in [2n-1]$ the parties invoke $[\![\beta_{i+1}]\!] := \mathcal{F}_{\mathsf{MPC}}(\mathcal{C}, ([\![K''_i]\!], [\![K''_{i+1}]\!]))$ where $\mathcal{C} : K \times K \to \{0,1\}$ is the equality circuit.
5. **[Duplicate]** The parties invoke $[\![X''']\!] := \mathcal{F}_{\mathsf{Agg}}(\textsc{Prefix}, \mathsf{dup}, [\![X'']\!], [\![\beta]\!])$ where $\mathsf{dup}(x_0, x_1) := x_0$.
6. **[Unpermute X]** The parties invoke $[\![X^*]\!] := \mathcal{F}_{\mathsf{Perm}}((\!\![\pi^-1]\!\!), [\![X''']\!])$.
7. **[Combine]** For $i \in [n]$, the parties set $[\![Z_i]\!] := ([\![X^*_{n+i}]\!], [\![Y_i]\!])$ if $\neg\mathsf{isNull}([\![X^*_{n+i}]\!]) \wedge \neg\mathsf{isNull}([\![Y_i]\!])$ and `Null` otherwise. The parties output $[\![Z]\!]$.

Fig. 10. Protocol $\Pi_{\mathsf{Join\text{-}OM}}$ for secret shared Join for one table X with unique join keys.

a *where* clause or another join—can be easily chained together, showcasing the composability of our protocol.

To improve performance, we introduce a concept called "remove dummies." As the name suggests, remove dummies eliminates null rows, unmatched rows, and intermediate aggregation rows. Removing these rows during each stage of the protocol reduces the size of the input being processed at that stage. However, removing dummies comes with the consequence of leaking some information about the table's size. If we choose to remove dummies, we can skip the unpermute step mentioned in each protocol because we shuffle the resultant table using a random permutation that is unknown to both parties.

Functionality $\mathcal{F}_{\mathsf{GroupBy}}([\![X]\!], [\![K]\!], \star)$:
Upon Aggregation columns X and group by column K and aggregation functions $\star$, perform the following:

1. Let $\pi := \mathsf{sort}(K)$
2. Let $(X'||K') := \pi(X||K)$
3. Let $g := \{i \in [n] \mid i = 1 \vee X'_{i-1} \neq X'_i\}$
4. For $i \in g$, let $X''_i := (X'_i \star ... \star X'_j)$ s.t. $K'_i = ... = K'_j$ and $X''_\ell :=$ `Null` for $\ell \in \{i+1, ..., j\}$.
5. Let $Z := \pi^{-1}(X'')$.
6. Output $[\![Z]\!] \leftarrow \mathsf{share}(Z)$.

Fig. 11. Functionality $\mathcal{F}_{\mathsf{GroupBy}}$ for secret shared groupby and reduce.

Protocol $\Pi_{\mathsf{GroupBy}}([X], [K], \star)$:

1. **[Sort]** The parties invoke $\llbracket\pi\rrbracket := \mathcal{F}_{\mathsf{Sort}}([K])$.
2. **[Permute X]** The parties invoke $[X'||K'] := \mathcal{F}_{\mathsf{Perm}}(\llbracket\pi\rrbracket, [X]||[K])$.
3. **[Control bits]** Let $[\beta_1] := [0]$. For $i \in [n-1]$ the parties invoke $[\beta_{i+1}] := \mathcal{F}_{\mathsf{MPC}}(\mathcal{C}, ([K''_i], [K''_{i+1}]))$ where $\mathcal{C} : K \times K \to \{0, 1\}$ is the equality circuit.
4. **[Aggregate]** The parties invoke $[X''] := \mathcal{F}_{\mathsf{Agg}}(\textsc{Prefix}, \mathsf{agg}, [X''], [\beta])$ where $\mathsf{agg}(x_0, x_1) := x_0 \star x_1$.
5. **[Unpermute X]** The parties invoke $[X^*] := \mathcal{F}_{\mathsf{Perm}}(\llbracket\pi^{-1}\rrbracket, [X''])$.
6. **[Combine]** For $i \in [n]$, the parties set $[W_i] := [X^*_i]$ if $\neg\mathsf{isNull}([X^*_i]) \land \neg([\beta_i])$ and `Null` otherwise. The parties output $[W]$.

Fig. 12. Protocol for groupby and reduce functionality.

7 Evaluation

All experiments for us and [30] were performed on a consumer grade laptop with a Apple M3 Pro with 36GB of RAM. Networking was performed using 10+ gigabyte throughput via localhost with sub millisecond latency. A single thread per party was used. All cryptographic operations are performed with computational security parameter $\kappa = 128$ and statistical security $\lambda = 40$. We consider set/table sizes of $n \in \{2^{12}, 2^{16}, 2^{20}\}$. In our Join protocols, each table has two columns: one 32-bit join key column, and an additional column whose sizes are 16 and 7 bits. For benchmarking the Where protocol, we use a table with two 32-bit columns and apply a 'column 1 ¿= column 2' filter. Finally, the Average protocol is evaluated on a table with three 32-bit columns, where the first column is used for the GroupBy operation, and the remaining two are used for the Aggregate (Average) calculation.

In Table 1 we report our performance numbers and comparison to protocol describe in [4]. Although this protocol computes the same functionality as $\Pi_{\mathsf{Join\text{-}OM}}$, it is in the honest majority setting and therefore is expected to be significantly faster than our dishonest majority protocol. Despite this, we choose to compare with [4] due it it being the closest alternative. As expected, their end to end running time is faster than us. However, note that up until 2^{16} our online running time is very competitive with [4] which demonstrates the effectiveness of our techniques.

We also observe that our $\Pi_{\mathsf{Join\text{-}OO}}$ protocol is approximately 3 times faster than our more general $\Pi_{\mathsf{Join\text{-}OM}}$ protocol. In more detail, $\Pi_{\mathsf{Join\text{-}OM}}$ begins by sorting both tables together which has nk overhead where n is size of table and k is bit length. Although this pre-sorting enables more efficient operations for the rest of the protocol, it also introduces additional overhead at the start. By contrast, $\Pi_{\mathsf{Join\text{-}OO}}$ avoids sorting altogether, which lowers the initial overhead and results in faster overall performance. Additionally, $\Pi_{\mathsf{Join\text{-}OM}}$ incurs an $O(D \log D)$ overhead from its duplication process. In contrast, we replace that step with straightforward comparison circuits that cost only $O(D)$, yielding further performance gains.

Protocol	Size	Offline		Online	
	n	Time(sec)	Com(MB)	Time(sec)	Com(MB)
[4]	2^{12}	-	-	0.21	22.8
	2^{16}	-	-	1.3	364
	2^{20}	-	-	21.6	5,560
Join OO	2^{12}	0.53	54	0.07	27
	2^{16}	7	714	0.66	432
	2^{20}	119	11,278	16	6,904
Join OM	2^{12}	1.4	141	0.1	82
	2^{16}	23	2,381	1.6	1,336
	2^{20}	295	32,372	99	21,633
Where	2^{12}	0.04	5.6	0.008	2.3
	2^{16}	0.67	72	0.05	36
	2^{20}	10.4	1,133.4	1	577
Average	2^{12}	1	111	0.07	52.2
	2^{16}	16	1,778	0.92	838
	2^{20}	377	29,362	36.5	13,556

Table 1. The above table mentions both running time & communication time for both online and offline phase for one to one join, one to many join, applying where clause and applying average with group by clause where n is the size of the table and join\groupby column size is 32 bits.

In Table 1, we see a substantial difference in both communication and running costs between the offline and online phases. The primary overhead in the offline phase stems from generating large numbers of OTs for the protocol which are then concerted into binary beaver triples. However, recent works such as [6, 7, 25] have demonstrated that it is practical to generate beaver triples directly without the need to first construct OTs. Although more work is still required to fully realize these improvements, one can theoretically obtain a $O(\kappa)$ runtime improvement via these new PCG protocols. Therefore, we expect our offline time to continue to decrease "for free" over the next few years. This suggests that the gap between honest majority protocols and two-party protocols is narrowing, even for advanced MPC protocols like ours.

Due to the need to sort the join column and this being the main overhead for $\Pi_{\mathsf{Join\text{-}OM}}$, this protocol scales almost linearly in the column bit length. For example, increasing it from 32 to 64 in our experiments should roughly double the overhead. Moreover, if the key length gets too long then they can be compressed

using a randomized encoding technique as described in [30]. This effectively ensures that the bit length of the join column can never be larger than $\lambda + 2 \log n$.

In all the benchmarks shown in Table 1, we disabled the 'remove dummies' flag that removes inactive rows generated during protocol execution; enabling this flag would likely yield additional speedups. We also tested our protocol in a multi-threaded setup using four threads, achieving a $2.5\times$ performance boost. However, increasing the thread count beyond four did not provide further gains because the workload was insufficient to keep additional threads fully occupied. With a more robust parallel implementation, we expect further performance gains. For instance, we can distribute OT generation across multiple threads, each handling a share of OTs in the offline phase. During the online phase, data can be partitioned and processed concurrently, either through thread pools or by using SIMD instructions to handle operations in bulk. While we already employ SIMD instructions at various stages, there is still room to refine their usage—and potentially adopt newer instruction sets to achieve even better performance. Carefully balancing workloads across threads or hardware accelerators (e.g., GPUs) can significantly reduce runtime, especially as datasets grow larger.

References

1. Alamati, N., Policharla, G., Raghuraman, S., Rindal, P.: Improved alternating moduli prfs and post-quantum signatures. IACR Cryptol. ePrint Arch. p. 582 (2024), https://eprint.iacr.org/2024/582
2. Albrecht, M.R., Davidson, A., Deo, A., Gardham, D.: Crypto dark matter on the torus - oblivious prfs from shallow prfs and TFHE. In: Joye, M., Leander, G. (eds.) Advances in Cryptology - EUROCRYPT 2024 - 43rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zurich, Switzerland, May 26-30, 2024, Proceedings, Part VI. Lecture Notes in Computer Science, vol. 14656, pp. 447–476. Springer (2024). https://doi.org/10.1007/978-3-031-58751-1_16, https://doi.org/10.1007/978-3-031-58751-1_16
3. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for MPC and FHE. In: Oswald, E., Fischlin, M. (eds.) 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, pp. 430–454. Springer (2015). https://doi.org/10.1007/978-3-662-46800-5_17, https://doi.org/10.1007/978-3-662-46800-5_17
4. Badrinarayanan, S., Das, S., Garimella, G., Raghuraman, S., Rindal, P.: Secret-shared joins with multiplicity from aggregation trees. In: Yin, H., Stavrou, A., Cremers, C., Shi, E. (eds.) Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022. pp. 209–222. ACM (2022). https://doi.org/10.1145/3548606.3560670, https://doi.org/10.1145/3548606.3560670
5. Blanton, M., Aguiar, E.: Private and oblivious set and multiset operations. Cryptology ePrint Archive, Report 2011/464 (2011), https://ia.cr/2011/464
6. Bombar, M., Bui, D., Couteau, G., Couvreur, A., Ducros, C., Servan-Schreiber, S.: Foleage: F4 ole-based multi-party computation for boolean circuits. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 69–101. Springer (2025)
7. Bombar, M., Couteau, G., Couvreur, A., Ducros, C.: Correlated pseudorandomness from the hardness of quasi-abelian decoding. In: Annual International Cryptology Conference. pp. 567–601. Springer (2023)
8. Boneh, D., Ishai, Y., Passelègue, A., Sahai, A., Wu, D.J.: Exploring crypto dark matter: - new simple PRF candidates and their applications. In: Beimel, A., Dziembowski, S. (eds.) Theory of Cryptography - 16th International Conference, TCC 2018, Panaji, India, November 11-14, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 11240, pp. 699–729. Springer (2018). https://doi.org/10.1007/978-3-030-03810-6_25, https://doi.org/10.1007/978-3-030-03810-6_25
9. Boyle, E., Couteau, G., Gilboa, N., Ishai, Y., Kohl, L., Rindal, P., Scholl, P.: Efficient two-round OT extension and silent non-interactive secure computation. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019. pp. 291–308. ACM (2019). https://doi.org/10.1145/3319535.3354255, https://doi.org/10.1145/3319535.3354255
10. Chen, H., Huang, Z., Laine, K., Rindal, P.: Labeled psi from fully homomorphic encryption with malicious security. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, Canada, October 14 - 16, 2018. ACM (2018)

11. Chen, H., Laine, K., Rindal, P.: Fast private set intersection from homomorphic encryption. In: CCS (2017)
12. Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. IACR Cryptol. ePrint Arch. p. 695 (2019), https://eprint.iacr.org/2019/695
13. Corrigan-Gibbs, H., Boneh, D.: Prio: Private, robust, and scalable computation of aggregate statistics. In: Akella, A., Howell, J. (eds.) 14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017. pp. 259–282. USENIX Association (2017), https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/corrigan-gibbs
14. Dahl, M., Patriquin, J., Dupis, Y., et. al: Tf encrypted: Encrypted deep learning in tensorflow (2020), https://tf-encrypted.io/
15. Demmler, D., Rindal, P., Rosulek, M., Trieu, N.: Pir-psi: Scaling private contact discovery. Proceedings on Privacy Enhancing Technologies **2018**(4) (2018), https://content.sciendo.com/view/journals/popets/2018/4/article-p159.xml
16. Dinur, I., Goldfeder, S., Halevi, T., Ishai, Y., Kelkar, M., Sharma, V., Zaverucha, G.: Mpc-friendly symmetric cryptography from alternating moduli: Candidates, protocols, and applications. In: Malkin, T., Peikert, C. (eds.) Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 12828, pp. 517–547. Springer (2021). https://doi.org/10.1007/978-3-030-84259-8_18, https://doi.org/10.1007/978-3-030-84259-8_18
17. Facebook: Crypten: A research tool for secure machine learning in pytorch (2020), https://crypten.ai/
18. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game pp. 218–229 (01 1987). https://doi.org/10.1145/28395.28420
19. Han, F., Zhang, L., Feng, H., Liu, W., Li, X.: Scape: Scalable collaborative analytics system on private database with malicious security. In: 2022 IEEE 38th International Conference on Data Engineering (ICDE). pp. 1740–1753 (2022). https://doi.org/10.1109/ICDE53745.2022.00176
20. Ion, M., Kreuter, B., Nergiz, E., Patel, S., Saxena, S., Seth, K., Shanahan, D., Yung, M.: Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738 (2017), https://eprint.iacr.org/2017/738
21. Kiss, Á., Liu, J., Schneider, T., Asokan, N., Pinkas, B.: Private set intersection for unequal set sizes with mobile applications. Proc. Priv. Enhancing Technol. **2017**(4), 177–197 (2017)
22. Kolesnikov, V., Kumaresan, R., Rosulek, M., Trieu, N.: Efficient batched oblivious PRF with applications to private set intersection. In: CCS (2016)
23. Kolesnikov, V., Matania, N., Pinkas, B., Rosulek, M., Trieu, N.: Practical multi-party private set intersection from symmetric-key techniques. In: CCS (2017)
24. Laur, S., Talviste, R., Willemson, J.: From oblivious aes to efficient and secure database join in the multiparty setting. In: International Conference on Applied Cryptography and Network Security. pp. 84–101. Springer (2013)
25. Li, Z., Xing, C., Yao, Y., Yuan, C.: Efficient pseudorandom correlation generators for any finite field. IACR Eurocrypt (2025)
26. Liagouris, J., Kalavri, V., Faisal, M., Varia, M.: Secrecy: Secure collaborative analytics on secret-shared data. CoRR **abs/2102.01048** (2021), https://arxiv.org/abs/2102.01048

27. Lindell, Y.: How to simulate it - a tutorial on the simulation proof technique. Cryptology ePrint Archive, Paper 2016/046 (2016), https://eprint.iacr.org/2016/046, https://eprint.iacr.org/2016/046
28. Luo, Q., Wang, Y., Dong, W., Yi, K.: Secure query processing with linear complexity. CoRR **abs/2403.13492** (2024). https://doi.org/10.48550/ARXIV.2403.13492, https://doi.org/10.48550/arXiv.2403.13492
29. Mohassel, P., Rindal, P.: ABY3: A mixed protocol framework for machine learning. IACR Cryptology ePrint Archive **2018**, 403 (2018), https://eprint.iacr.org/2018/403
30. Mohassel, P., Rindal, P., Rosulek, M.: Fast Database Joins and PSI for Secret Shared Data, p. 1271–1287. Association for Computing Machinery, New York, NY, USA (2020), https://doi.org/10.1145/3372297.3423358
31. Orrù, M., Orsini, E., Scholl, P.: Actively secure 1-out-of-n ot extension with application to private set intersection. In: Handschuh, H. (ed.) Topics in Cryptology – CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings. pp. 381–396. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-52153-4_22
32. Peceny, S., Raghuraman, S., Rindal, P., Shah, H.: Efficient permutation correlations and batched random access for two-party computation. Cryptology ePrint Archive, Paper 2024/547 (2024), https://eprint.iacr.org/2024/547, https://eprint.iacr.org/2024/547
33. Pinkas, B., Schneider, T., Segev, G., Zohner, M.: Phasing: Private set intersection using permutation-based hashing. In: Jung, J., Holz, T. (eds.) 24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015. pp. 515–530. USENIX Association (2015), https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/pinkas
34. Pinkas, B., Schneider, T., Weinert, C., Wieder, U.: Efficient circuit-based PSI via cuckoo hashing. In: EUROCRYPT (2018)
35. Pinkas, B., Schneider, T., Zohner, M.: Faster private set intersection based on OT extension. In: 23rd USENIX Security Symposium (USENIX Security 14). pp. 797–812. USENIX Association, San Diego, CA (2014), https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/pinkas
36. Pinkas, B., Schneider, T., Zohner, M.: Scalable private set intersection based on OT extension. Cryptology ePrint Archive, Report 2016/930 (2016), http://eprint.iacr.org/2016/930
37. Raghuraman, S., Rindal, P., Tanguy, T.: Expand-convolute codes for pseudorandom correlation generators from LPN. In: Handschuh, H., Lysyanskaya, A. (eds.) Advances in Cryptology - CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part IV. Lecture Notes in Computer Science, vol. 14084, pp. 602–632. Springer (2023). https://doi.org/10.1007/978-3-031-38551-3_19, https://doi.org/10.1007/978-3-031-38551-3_19
38. Resende, A.C.D., Aranha, D.F.: Faster unbalanced private set intersection (2018)
39. Riazi, M.S., Samragh, M., Chen, H., Laine, K., Lauter, K.E., Koushanfar, F.: XONN: xnor-based oblivious deep neural network inference. In: Heninger, N., Traynor, P. (eds.) 28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019. pp. 1501–1518. USENIX Association (2019), https://www.usenix.org/conference/usenixsecurity19/presentation/riazi
40. Rindal, P., Raghuraman, S.: Blazing fast psi from improved okvs and subfield vole. Cryptology ePrint Archive, Report 2022/320 (2022), https://ia.cr/2022/320

41. Rindal, P., Schoppmann, P.: VOLE-PSI: fast OPRF and circuit-psi from vector-ole. In: Canteaut, A., Standaert, F. (eds.) Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12697, pp. 901–930. Springer (2021)
42. Robert Helmer, Anthony Miyaguchi, E.R.: Testing privacy-preserving telemetry with prio (2018), https://hacks.mozilla.org/2018/10/testing-privacy-preserving-telemetry-with-prio/
43. Wagh, S., Gupta, D., Chandran, N.: Securenn: 3-party secure computation for neural network training. PoPETs **2019**(3), 26–49 (2019). https://doi.org/10.2478/popets-2019-0035, https://doi.org/10.2478/popets-2019-0035
44. Wang, Y., Yi, K.: Query evaluation by circuits. In: Libkin, L., Barceló, P. (eds.) PODS '22: International Conference on Management of Data, Philadelphia, PA, USA, June 12 - 17, 2022. pp. 67–78. ACM (2022). https://doi.org/10.1145/3517804.3524142, https://doi.org/10.1145/3517804.3524142