Fast Database Joins and PSI for Secret Shared Data

Payman Mohassel, Peter Rindal, Mike Rosulek

September 9, 2020

Abstract

We present a scalable protocol for database joins on secret shared data in the honest-majority three-party setting. The key features of our protocol are a rich set of SQL-like join/select queries and the ability to compose join operations together due to the inputs and outputs being generically secret shared between the parties. Provided that all joins operate on unique primary keys, no information is revealed to any party during the protocol. In particular, not even the sizes of intermediate joins are revealed. All of our protocols are constant-round and achieve $O(n)$ communication and computation overhead for joining two tables of n rows.

These properties make our protocol ideal for *outsourced secure computation*. In this setting several non-colluding servers are setup and the input data is shared among them. These servers then perform the relevant secret shared computation and output the result. This model has recently been gaining traction in industry, e.g. Facebook's Crypten, Cape Privacy's TFEncrypted, Mozilla Telemetry.

We additionally implement two applications on top of our framework. The first application detects voter registration errors within and between agencies of 50 US states, in a privacy-preserving manner. The second application allows several organizations to compare network security logs to more accurately identify common security threats, e.g. the IP addresses of a bot net. In both cases, the practicality of these applications depends on efficiently performing joins on millions of secret shared records. For example, our three party protocol can perform a join on two sets of 1 million records in 4.9 seconds or, alternatively, compute the cardinality of this join in just 3.1 seconds.

1 Introduction

We consider the problem of performing SQL-style join operations on tables that are secret shared among three parties, in the presence of an honest majority. Our proposed protocol takes two or more arbitrarily secret shared database tables and constructs another secret shared table containing a `join` of the two tables, without revealing *any* information beyond the secret shares themselves. Our protocol is constant-round and has computation and communication overhead that is linear

in the size of the tables. Simulation-based security is achieved in the semi-honest setting with an honest-majority. Our protocol can perform inner, left and full joins along with union and arbitrary computation on the resulting table, with the best performance and security guarantees when joins operate on unique primary keys.

New techniques [PSZ14, PSSZ15, PSZ16, KKRT16, PSWW18, CLR17, CHLR18, IKN+17, RA18, KLS+17, OOS17, KMP+17] for performing set intersection, inner join and related functionalities have shown great promise for practical deployment. The vast majority of these works perform *private set intersection* (PSI), which is analogous to revealing the entire result of an inner join. Computing a join *without revealing it* (*i.e.*, performing further joins, filtering, or computing only aggregate information) is significantly harder, and optimizations for such a setting do not automatically translate to our composable setting. We highlight a few notable results that hide the contents of a join (revealing only some function of it):

Ion et al. recently deployed a private set intersection sum protocol[IKN+17] to allow customers of Google Adwords to correlate online advertising with offline sales, while preserving user privacy. Pinkas et al. [PSWW18] also introduced a practical protocol that can compute any (symmetric) function of the intersection and associated data. These protocols can be framed in terms of SQL queries consisting of an inner join followed by an aggregation on the resulting table, e.g. summing a column. Neither of these protocols (and almost no prior related results) support secret-shared inputs, but rather require the source tables to be held in the clear by each party.

The majority of these protocols consider the two party setting and are based on various cryptographic primitives, e.g. exponentiation [IKN+17], oblivious transfer[PSWW18], or fully homomorphic encryption[CLR17]. However, in this work we alter the security model to consider three parties with an honest majority. The motivation is that typical protocols in this setting (e.g.[AFL+16]) require less computation and communication than similar two party protocols, by a factor of at least the security parameter $\kappa = 128$. Moreover, we will see that the honest majority enables various algorithms which are orders of magnitude more efficient, e.g. oblivious permutations require $O(n)$ work instead of $O(n \log n\kappa)$ [MS13].

Given this observation we investigate how to leverage the efficiency gains in the three party setting to construct practical protocols for performing set intersection and other SQL-like operations where both the inputs and outputs are secret shared. One critical aspect of this input/output requirement is that join operations can then be *composed* together, where the output of a join can be the input to another. Allowing this composability greatly increases the ability to perform highly complex queries and enables external parties to contribute data simply by secret sharing it between the primary parties which participate in the protocol.

The fact that we support secret-shared inputs also leads to *outsourced secure computation*. Three non-coluding servers can be established, and inputs can be provided by the servers themselves or by external parties simply by secret sharing

the input among the servers. This model has been gaining traction in industry. Mozilla recently deployed a service to collect telemetry data about Firefox[RH18] using two non-colluding servers running the Prio protocol[CB17]. Other examples include privacy preserving machine learning frameworks which support secret shared inputs such as Facebook's Crypten built for PyTorch[Fac20] and Cape Privacy's FTEncrypted built for TensorFlow[MD20].

Complex joins on secret-shared inputs/outputs are valuable in all of these examples. Most privacy preserving machine learning publications[MR18, Fac20, MD20, WGC19, RSC+19] assume that the data being trained on has already been joined together. However, without a framework like ours it is unclear how this would be accomplished while preserving privacy and efficiency. The generality of our approach allows us to solve these problems and many others, e.g. the two applications presented in Section 5.

1.1 Functionality

Our protocol offers a wide variety of functionality including set intersection, set union, set difference and a variety of SQL-like joins with complex boolean queries. Generally speaking, our protocol works on tables of secret shared data which are functionally similar to SQL tables. This is in contrast to traditional PSI and PSU protocols[PSZ14, PSSZ15, PSZ16, KKRT16] in that each record is now a tuple of values as opposed to a single key.

We define our database tables in the natural way. Each table can be viewed as a collection of rows or as a vector of columns. For a table X, we denote the ith row as $X[i]$ and the jth column as X_j. Our core protocol requires each table to contain unique values in the column defining the join (i.e., we can only join on "unique primary keys"). For example, if we consider the following SQL styled join/intersection query

$$\texttt{select } X_2 \texttt{ from } X \texttt{ inner join } Y \texttt{ on } X_1 = Y_1$$

then the join-keys are X_1 and Y_1. This uniqueness condition can be extended to the setting where multiple columns are being compared for equality. Later on we will discuss the case when such a uniqueness property does not hold. Our protocols also support a `where` clause that filters the selection using an arbitrary predicate of the X and Y rows. Furthermore, the `select` clause can also return a function of the two rows. For example,

$$\begin{aligned}&\texttt{select } X_1, max(X_2, Y_2) \texttt{ from } X \texttt{ inner join } Y \\ &\quad \texttt{on } X_1 = Y_1 \texttt{ where } Y_2 > 23.3\end{aligned}$$

In general, the supported join operations can be characterized in three parts: 1) The select function $S(\cdot)$ that defines how the rows of X, Y are used to construct each output row, e.g. $S(X, Y) = (X_1, max(X_2, Y_2))$. 2) The predicate $P(\cdot)$ that defines the `where` clause, and 3) which columns are being joined on.

Several other types of joins are also supported including left and right joins, set union and set minus (difference) and full joins. A left join takes the inner join and includes all of the missing records from the left table. For the records solely from the left table, the resulting table contains `NULL` for the columns from the right table. Right join is defined symmetrically. A full join is a natural extension where all the missing rows from X and Y are added to the output table.

We define the union of two tables to contain all records from the left table, along with all the records from the right table which are not in the intersection with respect to the join-keys. Note that this definition is not strictly symmetric with respect to the left and right tables due to rows in the intersection coming from the left table. Table minus is similarly defined as all of the left table whose join-column value is not present in the right table.

Beyond these various join operations, our framework supports two broad classes of operations which are a function of a single table. The first is a general SQL select statement which can perform computation on each row (e.g. compute the max of two columns) and filter the results using a `where` clause predicate. The second class is referred to as an aggregation which performs an operation across all of the rows of a table — for example, computing the sum, counts, or the max of a given column.

1.2 Our Results

We present the first practical secure multi-party computation protocol for performing SQL-style database joins with linear overhead and constant rounds. Our protocol is fully composable in that the input and output tables are generically secret shared between the parties. We achieve this result by combining various techniques from private set intersection and secure computation more broadly. We build on the the binary secret sharing technique of [AFL+16] with enhancements described by [MR18]. We then combine this secret sharing scheme with cuckoo hashing[PSZ14], an MPC friendly PRF[ARS+15] and a custom protocol for evaluating an oblivious switching network[MS13]. Using these building blocks our protocol is capable of computing the intersection of two tables of $n = 2^{20}$ rows in 4.9 seconds. Beyond these two specific functionalities, our protocol allows arbitrary computation applied to a shared table. Compared to existing three party protocols with similar functionality (composable), our implementation is roughly 1000× faster. When compared with *non-composable* two party protocol, we observe a larger difference ranging from our protocol being 1.25× slower to 4000× faster depending on the functionality.

Building on our proposed protocol we demonstrate its utility by showcasing two potential applications. The first prototype would involve running our protocol between and within the states of the United States to validate the accuracy of the voter registration data in a privacy preserving way. The Pew Charitable Trust[Smi14] reported 1 in 8 voter registration records in the United States contains a serious error while 1 in 4 eligible citizens remain unregistered. Our privacy preserving protocol identifies when an individual's address is out of date or more seriously if someone is

registered to vote in more than one state which could allow them to cast two votes. Due to how the data is distributed between different governmental agencies, it will be critical that our protocol allows for composable operations. We implement this application and demonstrate that it is practical to run at a national scale (250M records) and low cost.

The second application that we consider allows multiple organizations to compare computer security incidents and logs to more accurately identify unwanted activities, e.g. a bot net. Several companies already offer this service including Facebook's ThreatExchange[thr18] and an open source alternative[alt18]. One of the primary limitations of these existing solutions is the requirement that each organization send their security logs to a central party, e.g. Facebook. We propose using our protocol to distribute the trust of this central party between three parties such that privacy is guaranteed so long as there is an honest majority.

1.3 Related Work

We now review several related works that use secure computation techniques. With respect to functionality the closest related work is that of Blanton and Aguiar[BA12] which describes a relatively complete set of protocols for performing intersections, unions, set difference, etc. and the corresponding SQL-like operations. Moreover, these operations are composable in that the inputs and outputs are secret shared between the parties. At the core of their technique is the use of a generic MPC protocol and an oblivious sorting algorithm that merges the two sets. This is followed by a linear pass over the sorted data where a relation is performed on adjacent items. Their technique has the advantage of being very general. However, the proposed algorithm has complexity $O(n \log^2 n)$, is not constant round, and also requires unique join keys. This results in poor concrete performance as shown in Section 6.

Pinkas, Schneider and Zohner [PSZ14] introduced a paradigm for set intersection that combines a hash table technique known as cuckoo hashing with a randomized encoding technique using oblivious transfer. Due to the hashing technique, the problem is reduced to comparing a single item x to a small set $\{y_1, ..., y_m\}$. Oblivious transfer is then used to interactively compute the randomized encoding x' of x while the other party locally computes the encodings $\{y'_1, ..., y'_m\}$. A plaintext intersection can then be perform directly on these encodings. With the use of several optimization[PSSZ15, PSZ16, KKRT16, OOS17] this paradigm is extremely efficient and can perform a set intersection using $O(n)$ calls to a random oracle and $O(n)$ communication. These protocols are *not composable* since the input must be known in the clear. Making them composable is non-trivial and they would likely introduce a large overhead.

Laur, Talvista and Willemson[LTW13] present techniques in the honest majority setting for composable joins, unions and many other operations at the expense of information leakage. Consider two parties each with a sets X, Y. The parties first generate secret shares of the sets and then use a generic MPC protocol to apply a

pseudorandom function (PRF) F to the shared sets to compute $X' = \{F_k(x) \mid x \in X\}, Y' = \{F_k(y) \mid y \in Y\}$ where the key k is uniformly sampled by the MPC protocol (i.e. neither party knows k). X' and Y' are then revealed to both parties who use this information to infer the intersection, union and many other SQL-like operations. This basic approach dates back to the first PSI protocols [Mea86, HFH99] where the (oblivious) PRF was implemented using a Diffie-Hellman protocol. [LTW13] extended this paradigm to allow the input sets to be secret shared as opposed to being known in the clear.

The primary limitation of this approach is that all operations require all parties to know X' and Y'. This prevents the protocol from being composable without significant information leakage. In particular, the cardinality of $X' \cap Y'$ and the result of the `where` clause for each row is revealed. This is of particular concern when several dataset are being combined. Learning the size of the intersection or the union can represent significant information. For instance, in the threat log application the union of many sets are taken. Each of these unions would reveal how many unique logs the new set has. Alternatively, taking the join between a set of hospital patients and a set of HIV positive patients would reveal how many have HIV. When combined with other information it could lead to the ability to identify some or all of these patients. Beyond this, the provided three party implementation achieved relatively poor performance. A join between two tables of a million records is estimated to require one hour on their three benchmark machines[LTW13]. Looking forward, our protocol can perform a similar join operation in 4 seconds while preventing all leakage.

Bater, Elliott, et al.[BEE+17] describe a outsourced MPC protocol where a client sends a SQL query to one of the computational parties who runs a garbled circuit based protocol amongst themselves. They present optimizations where some the computation is performed outside the MPC. We leave it as future work to explore the application of our new techniques in their setting.

2 Preliminaries

2.1 Security Model

Our protocols are presented in the semi-honest three-party setting with an honest majority. That is, the received messages of any single party are computationally indistinguishable from messages that are only dependent of their final output. We present our ideal functionality in Figure 7. See [AFL+16] for a more details of our simulation based security model.

2.2 Notation

Let $[m]$ denote the set $\{1, 2, ..., m\}$. Let V be a vector with elements $V = (V_1, ..., V_n)$. We also use the notion $V[i]$ to index the ith element V_i. We define a permutation of

size m as an injective function $\pi : [m] \to [m]$. We extend this definition such that when π is applied to a vector V of m elements, then $\pi(V) = (V_{\pi(1)}, ..., V_{\pi(m)})$. The image of a function $f : X \to Y$ is defined as $image(f) := \{y \in Y : \exists x \in X, f(x) = y\}$. Preimage of a pair (f, y) is defined as $preimage(f, y) := \{x \in X : f(x) = y\}$. We use n to represent the number of rows a table has. Parties are referred to as P_0, P_1, P_2. We use κ to denote the computational security parameter, e.g. $\kappa = 128$, and λ as the statistical security parameters, e.g. $\lambda = 40$.

2.3 Secret Sharing Framework

Our protocol builds on the ABY^3 framework of Rindal and Mohassel [MR18] for secure computation of circuits. That is, we use their binary/arithmetic addition and multiplication protocols along with their share conversion protocols. We will use the notation that $[\![x]\!]$ is a 2-out-of-3 *binary replicated secret sharing* of the value x. That is, (x_0, x_1, x_2) are sampled uniformly s.t. $x = x_0 \oplus x_1 \oplus x_2$. Party P_i holds the shares $x_i, x_{i+1 \mod 3}$. We use the notation $[\![x]\!]_i$ to refer to share x_i. $[\![x]\!]$ can locally be converted to a 2-out-of-2 sharing $\langle\!\langle x \rangle\!\rangle$ where P_i holds x'_0 and P_j holds x'_1 s.t. $x = x'_0 \oplus x'_1$, e.g. $i = 0, j = 1$. $\langle\!\langle x \rangle\!\rangle_k$ refers to x'_k. $\langle\!\langle x \rangle\!\rangle$ can also be converted back to $[\![x]\!]$ using one round of communication.

2.4 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. Typically the required table size is $m \approx 1.6n$ for $\lambda = 40$ bits of statistical security, see [DRRT18].

3 Our Construction

3.1 Overview

First we describe our join algorithm without any privacy and then we will discuss how this translates to the secret shared setting. Figure 1 depicts our algorithm with the following phases:

1. Y is inserted into a cuckoo hash table T based on the join-key(s). That is, let us assume the columns Y_1 and X_1 are the join keys. Then row $Y[i]$ is inserted at $T[j]$ for some $j \in \{h_0(Y_1[i]), h_1(Y_1[i])\}$.

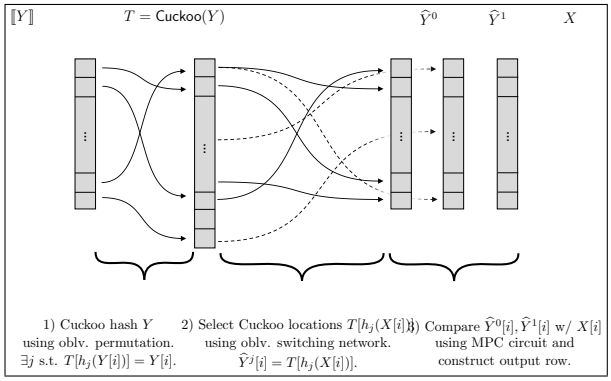

Figure 1: Overview of the join protocol using oblivious switching network.

2. Each row $X[i]$ needs to be compared with the rows $T[j]$ for $j \in \{h_0(X_1[i]), h_1(X_1[i])\}$. As such, $T[h_0(X_1[i])]$ is mapped to a new row $\widehat{Y}^0[i]$ and $T[h_1(X_1[i])$ to $\widehat{Y}^1[i]$.

3. It is now the case that if row $X[i]$ has a machining key in Y, then this row will be located at $\widehat{Y}^0[i]$ or $\widehat{Y}^1[i]$. As such, these rows can directly be compared to determine if there is a match on the join keys and the where clause evaluates to true. Let $b_i = 1$ if there is such a match and 0 otherwise.

4. Various types of joins can then be constructed from locally comparing row i from these tables, i.e. $X[i], \widehat{Y}^0[i], \widehat{Y}^1[i]$. For example, an inner join is constructed from all the rows where $b_i = 1$ by selecting the values from $X[i]$ and either $\widehat{Y}^0[i]$ or $\widehat{Y}^1[i]$ depending on which one matches. If there is no match, then that output row is set to `NULL`.

The main challenge in bringing the described algorithm to the secret shared setting is constructing the cuckoo hash table T and selecting rows from T without leaking sensitive information. We achieve this with the use an MPC friendly *randomized encoding* and a new three-party protocol called an *oblivious switching network*.

Let us continue to assume that the columns X_1 and Y_1 are the join-keys. Our protocol begins by generating a *randomized encoding* for each of the secret shared join-key $[\![x_i]\!] \in [\![X_1]\!]$ and $[\![y_i]\!] \in [\![Y_1]\!]$. Figure 2 contains the ideal functionality for this encoding which takes secret shares from the parties, apply a PRF F_k to the reconstructed value using a internally sampled key k, and returns the resulting value to one of the three parties. For $[\![x_i]\!] := [\![X_1]\!][i]$, P_0 will learn $F_k(x_i)$ while P_1 will learn $F_k(y_i)$ for $[\![y_i]\!] := [\![Y_1]\!][i]$. Since the join-keys x_i (resp. y_i) are unique and k is not known, this can be simulated by sending random values to P_0 (resp. P_1).

Party P_1 proceeds by constructing a *secret shared* cuckoo hash table $\langle\!\langle T \rangle\!\rangle$ from the rows of $[\![Y]\!]$ where the hash function values for row i are defined as $h_j(y_i) = H(j||F_k(y_i))$. Note that P_1 knows only the randomized encodings $F_k(y_i)$ of each row $Y[i]$, and not the contents of the row itself. The goal in this step is to construct a secret shared cuckoo table $\langle\!\langle T \rangle\!\rangle$ such that row $Y[i]$ is located at $T[h_j(y_i)]$ for some

j. We construct $\langle\langle T \rangle\rangle$ using a three-party *oblivious permutation protocol* where P_1 inputs a permutation π, all parties input secret shares of Y, and the result is secret shares of "Y permuted according to π" which forms T (details follow later). This completes Step 1 and is the first transformation shown in Figure 1.

It is now the case that $\langle\langle T \rangle\rangle$ is a valid cuckoo hash table of $[\![Y]\!]$ which is secret shared between P_0 and P_1. Party P_0, who knows the randomized encodings $F_k(x_i)$ for all $[\![x_i]\!] := [\![X_1]\!][i]$, now must compare the rows of $\langle\langle T \rangle\rangle$ indexed by $h_j(x_i) = H(j||F_k(x_i))$ with the row $[\![X]\!][i]$. In particular, assuming we use two cuckoo hash functions, then P_0 constructs two *oblivious switching networks* that maps the shares $\langle\langle T[h_0(x_i)] \rangle\rangle$ and $\langle\langle T[h_1(x_i)] \rangle\rangle$ to be "aligned" with $[\![X]\!][i]$. Exactly how such a network operates is discussed later but the result is two new tables $\langle\langle \widehat{Y}^0 \rangle\rangle, \langle\langle \widehat{Y}^1 \rangle\rangle$ such that $T[h_j(x_i)] = \widehat{Y}^j[i]$. This completes Step 2 and is the second transformation shown in Figure 1.

Once the shares of $\widehat{Y}^0[i] = T[h_0(x_i)], \widehat{Y}^1[i] = T[h_1(x_i)]$ are obtained using the switching network, the parties employ an MPC protocol to directly compare these rows with $[\![X]\!][i]$. That is, they compute a bit $[\![b]\!]$ which equals one if the join-keys are equal and the `where` clause $P(\langle\langle \widehat{Y}^j \rangle\rangle[i], [\![X]\!][i])$ outputs one for some j. For each row, the output row for an inner join is constructed as $S(\langle\langle \widehat{Y}^j \rangle\rangle[i], [\![X]\!][i])$ using MPC where S is the user defined selection circuit. In addition, the MPC circuit outputs the secret shared flag $[\![b]\!]$ indicating whether this row is set to `NULL`.

Left joins work in a similar way except that all rows of X are output and marked not `NULL`. Finally, unions can be computed by including all of Y in the output and all of the rows of X where the comparison bit $[\![b]\!]$ is zero. Regardless of the type of join, the protocols do not reveal any information about the tables. In particular, not even the cardinality of the join is revealed due to the use of `NULL` rows.

3.2 Randomized Encodings

The randomized encoding functionality $\mathcal{F}_{\text{ENCODE}}$ of Figure 2 enables the parties to coordinate their secret shares without revealing the underlying values. In particular, the parties will construct a cuckoo hash table using these encodings. The functionality takes as input several tuples $([\![B_i]\!], [\![X_i]\!], P_i)$ where $B_i \in \{0,1\}^d$ is an array of d bits, $X_i \in (\{0,1\}^\sigma)^d$ is a array of d strings and P_i that denotes that party P_i should be output the encodings for this tuple. The functionality assigns a random ℓ bit encoding for each input $x \in \{0,1\}^\sigma$. For $j \in [d]$, if the bit $B_i[j] = 0$ then the functionality outputs the encoding for $X_i[j]$ and otherwise a random ℓ bit string. Looking forward, $B_i[j] = 1$ will mean that the key $X_i[j]$ is actually set to `NULL` and a random encoding should be returned.

LowMC Encodings. We realize this functionality using the LowMC block cipher[ARS+15]. When implemented with the honest majority MPC protocols[AFL+16], this approach results in extremely high throughput, computing up to one million encodings per second. Once the parties have their secret shared inputs, they sample a secret

Parameters: Input string size of σ bits and output encoding size of ℓ bits.
[Encode] Upon receiving command (ENCODE, $\{([\![B_i]\!], [\![X_i]\!], P_i)\}$) from all parties where $X_i \in (\{0,1\}^\sigma)^{d_i}, B_i \in \{0,1\}^{d_i}$ for some $d_i \in \mathbb{Z}^*$.

1. Sample a uniformly random $F : \{0,1\}^\sigma \to \{0,1\}^\ell$. Define $F' : \{0,1\} \times \{0,1\}^\sigma \to \{0,1\}^\ell$ as $F'(b,x) = \bar{b}F(x) + br$ where $r \leftarrow \{0,1\}^\ell$ is sampled each call.
2. For each $([\![B_i]\!], [\![X_i]\!], P_i)$, send $\{F'(b,x) \mid (b,x) \in \text{ZIP}(B_i, X_i)\}$ to P_i.

Figure 2: The Randomized Encoding ideal functionality $\mathcal{F}_{\text{ENCODE}}$

shared LowMC key uniformly and encrypt each input under that key using the MPC protocol. These encryptions are revealed as the encodings to the appropriate party.

The LowMC cipher is parameterized by a block size ℓ, keys size κ, s-boxes per layer m and the desired data complexity d. To set these parameters, observe that the adversary only sees a bounded number of block cipher outputs (encodings) per key. As such, the data complexity can be bounded by this value. For our implementation we upper bound the number of outputs by $d = 2^{30}$. The remaining parameters are set to be $\ell \in \{80, 100\}$ and $m = 14$ which results in $r = 13$ rounds and computational security of $\kappa = 128$ bits[ARS+15]. The circuit for $\ell = 80$ contains 546 AND gates (meaning each party will send only 546 bits per encoding).

One issue with the LowMC approach alone is that the input size is fixed to be at most $\ell \in \{80, 100\}$ bits. However, we will see that the larger join protocol requires an arbitrary input size σ. This is accommodated by applying a universal hash function to the input shares. Specifically, the parties jointly pick a random matrix $E \leftarrow \{0,1\}^{\sigma \times \ell}$. The parties can then locally multiply each secret shared input before it is sent into the LowMC block cipher.

The security of this transformation follows from $xE \neq x'E$ with overwhelming probability if $x \neq x'$. In particular, $f(x) = xE$ is a universal hash function given that E is independent of x. As such the probability that $f(x) = f(x')$ for any $x \neq x'$ is $2^{-\ell}$. Applying the birthday bound we obtain that probability of any collisions among the tuples is $2^{-\ell+p}$ where $p = \log_2 D^2/2 = 2\log_2(D) - 1$ and $D = \sum_i d_i$ is the total number of encodings.

Conditioned on the inputs to the block cipher being unique, the outputs of the block cipher is also distinct and indistinguishable from random ℓ bit strings. As such, in the simulation the real outputs can be replaced with that of the ideal functionality so long as $2^{-\ell+p}$ is statistically negligible, i.e. $\ell - p \geq \lambda$.

3.3 Oblivious Switching Network

The ideal functionality of a switching network was introduced by Mohassel and Sadeghian[MS13]. It obliviously transform a vector $A = (A_1, ..., A_n)$ such that the output is $A' = (A_{\pi(1)}, ..., A_{\pi(m)})$ for an arbitrary function $\pi : [m] \to [n]$. The accompanying protocol of [MS13] was designed in the two party setting where the first party inputs A while the second party inputs a description of π. We introduce a

Parameters: Input, output size of σ, ℓ bits (respectively). Computational security parameter κ.
[Encode] Upon receiving command $(\textsc{Encode}, \{([\![B_i]\!], [\![X_i]\!], P_i)\})$ from all parties where each $X_i \in (\{0,1\}^\sigma)^{d_i}$. Let $d = \max_i(d_i)$.

1. If $\sigma > \ell$, the parties jointly sample a matrix $E \in \{0,1\}^{\sigma \times \ell}$. Otherwise E is the $\sigma \times \ell$ identity matrix.
2. The parties have $\mathcal{F}_{\text{MPC}}$ evaluate the following circuit:
   (a) Uniformly sample a key k for a LowMC cipher with block size ℓ, security κ and data complexity at least d blocks.
   (b) For each $([\![B_i]\!], [\![X_i]\!], P_i)$ input pair, reveal $\{F'(b,x) \mid (b,x) \in \textsc{zip}(B_i, X_i)\}$ to P_i where $F'(b,x) = \mathsf{LowMC}_k(xE) \oplus br$ and $r \leftarrow \{0,1\}^\ell$ is sampled for each call.

Figure 3: The randomized encoding LowMC protocol.

Parameters: 3 parties denoted as the P_p, P_s and P_r. Elements are strings in $\Sigma := \{0,1\}^\sigma$. An input vector size of n and output size of m.
[Switch] Upon the command $(\textsc{switch}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from the P_p and $(\textsc{switch}, \langle\!\langle A \rangle\!\rangle_1)$ from the P_s:

1. Interpret $\pi : [m] \to [n]$ and $A \in \Sigma^n$.
2. Compute $A' \in \Sigma^m$ s.t. $\forall i \in [m], A_{\pi(i)} = A'_i$.
3. Generate $\langle\!\langle A' \rangle\!\rangle$ and send $\langle\!\langle A' \rangle\!\rangle_0$ to P_p and $\langle\!\langle A' \rangle\!\rangle_1$ to P_r.

Figure 4: The Oblivious Switching Network ideal functionality $\mathcal{F}_{\textsc{Switch}}$. See Figure 12, 14 for $\mathcal{F}_{\textsc{Permute}}$ and $\mathcal{F}_{\textsc{Duplicate}}$.

new oblivious switching network protocol tailored for the honest majority setting with significantly efficiency improves. Our protocol has $O(n)$ overhead and is constant round. [MS13] requires $O(n \log n \kappa)$ communication/computation and $O(\log n)$ rounds.

The ideal functionality of our protocol is given in Figure 4 with three parties, a programmer P_p, a sender P_s and a receiver P_r. P_p has a description of π while P_p,P_s have a secret sharing of a vector $A \in \Sigma^n$ where $\Sigma = \{0,1\}^\sigma$. P_p and P_r are each output a share of $\langle\!\langle A' \rangle\!\rangle$ s.t. $A' = (A_{\pi(1)}, ..., A_{\pi(m)})$. For ease of presentation, we will initially assume A is the private input of P_s.

Permutation Network. We begin with a restricted class of switching networks where the programming function π is injective. That is, each input element A_i will be mapped to a maximum[1] of one location in the output A'. As we will see later, this property will simplify the implementation since we do not need to duplicate any element. Intuitively, the Permute protocol of Figure 5 instructs P_s to first shuffled A in a random order (as specified by π_0) and then secret share it between P_p & P_r.

[1]Strictly speaking, this protocol implementation a generalization of a permutation network since it allows some elements to not appear in the output, i.e. $m < n$ and $\pi : [m] \to [n]$.

Then P_{p} & P_{r} will reorder these shares (as specified by π_1) to be in the desired order (i.e. π). This is done as follows, P_{p} samples two random functions π_0, π_1 such that $\pi_1 \circ \pi_0 = \pi$, $\pi_0 : [n] \to [n]$ is bijective and $\pi_1 : [m] \to [n]$ is injective. P_{p} sends π_1 to P_{r} and $\pi_0, S \leftarrow \Sigma^n$ to P_{s} who sends $B := (A_{\pi_0(1)} \oplus S_0, ..., A_{\pi_0(n)} \oplus S_n)$ to P_{r}. The final shares of $A' = \pi(A)$ are defined as P_{p} holding $\langle\!\langle A' \rangle\!\rangle_0 := (S_{\pi_1(1)}, ..., S_{\pi_1(m)})$ and the P_{r} holding $\langle\!\langle A' \rangle\!\rangle_1 := (B_{\pi_1(1)}, ..., B_{\pi_1(m)})$.

The simulation of this protocol is perfect. The view of P_{s} contains a uniform permutation π_0 and vector S. Similarly, the view of P_{r} contains π_1 which is uniformly distributed (when π_0 is unobserved) and the uniform vector B. See Section B.1 for details. In our computational secure setting, π_0, S can be generated locally by P_{p} and P_{s} using a common source of randomness, e.g. a seeded PRG. This reduces the rounds to 1.

Shared Inputs. As presented here in the text our protocols assume the input vector A being transformed is the private input of the P_{s}. However, the full protocols will require the input A to secret shared. Let us assume we have some switching network protocol Π which takes input A from P_{s}, π from P_{p} and output shares of $\pi(A)$. Then this can be transform to shared input $\langle\!\langle A \rangle\!\rangle$. The parties invoke Π where P_{s} input their share $\langle\!\langle A \rangle\!\rangle_1$ and P_{p} inputs π. P_{p} and P_{r} receive $\langle\!\langle B \rangle\!\rangle$ from the functionality. The final result can then be computed as P_{r} holding $\langle\!\langle A' \rangle\!\rangle_1 := \langle\!\langle B \rangle\!\rangle_1$ while P_{p} locally defines $\langle\!\langle A' \rangle\!\rangle_0 := \langle\!\langle B \rangle\!\rangle_0 \oplus \pi(\langle\!\langle A \rangle\!\rangle_0)$. It is easy to verify that $A' = \pi(\langle\!\langle A \rangle\!\rangle_1) \oplus \pi(\langle\!\langle A \rangle\!\rangle_0) = \pi(A)$. The protocol descriptions in Figure 5 include this shared input modification. However, here in the text we will continue to assume A is the sole input of P_{s}.

Duplication Network.

The Duplication protocol of Figure 5 considers a second type of restricted network where $\pi : [n] \to [n]$, s.t. $\pi(1) = 1$ and $\pi(i) \in \{i, \pi(i-1)\}$ for $i = 2, ..., n$. That is, each output position is either a copy of the same input position (i.e. $\pi(i) = i$) or is a duplicate of the previous output position (i.e. $\pi(i) = \pi(i-1)$). For example, let the truth table of π be $(\pi(1), ..., \pi(6)) = (1, 1, 3, 4, 4, 4)$ and therefore $A' = (A_1, A_1, A_3, A_4, A_4, A_4)$. Note the only change is that A_1, A_4 were duplicated into the next position(s). This transformation can be characterized by a vector $b \in \{0,1\}^n$ where $b_i = 1$ denotes that the output position i should be a copy of output position $i-1$, i.e. $b = (0, 1, 0, 0, 1, 1)$ for the example above. Therefore we get the relation $A'_i = \overline{b_i} A_i \oplus b_i A'_{i-1}$ for $i \in [2, n]$.

As a warm-up, let us fix some index i and consider the simpler relation where

$$A'_i = \overline{b_i} A_i \oplus b_i A_{i-1},$$

i.e. A'_i is either A_i or A_{i-1} and not A'_{i-1} as described before. Conceptually, we will implement this using an OT-like protocol with OT messages (A_i, A_{i-1}) and select-bit b_i. P_{s} samples three uniform strings $\langle\!\langle A'_i \rangle\!\rangle_1, w_0, w_1 \leftarrow \Sigma$ and a uniform bit $\phi \leftarrow \{0,1\}$. P_{s} constructs two messages $m_0 = A_i \oplus \langle\!\langle A'_i \rangle\!\rangle_1 \oplus w_\phi$ and $m_1 = A_{i-1} \oplus \langle\!\langle A'_i \rangle\!\rangle_1 \oplus w_{\phi \oplus 1}$. P_{s} sends w_0, w_1 to P_{r} and sends m_0, m_1, ϕ to P_{p} who sends

Parameters: 3 parties denoted as P_{p}, P_{s} and P_{r}. Elements are strings in $\Sigma := \{0,1\}^{\sigma}$. An input, output vector size of n, m.

[Permute] Upon the command $(\textsc{Permute}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from P_{p} and $(\textsc{Permute}, \langle\!\langle A \rangle\!\rangle_1)$ from P_{s}. Require that $\pi : [m] \to [n]$ is *injective* and $\langle\!\langle A \rangle\!\rangle_0, \langle\!\langle A \rangle\!\rangle_1 \in \Sigma^n$. Then:

1. P_{p} uniformly samples a bijection $\pi_0 : [n] \to [n]$ and let $\pi_1 : [n] \to [m]$ s.t. $\pi_1 \circ \pi_0 = \pi$. P_{p} sends π_0 and $S \leftarrow \Sigma^n$ to P_{s}.
2. P_{s} sends $B := (\langle\!\langle A_{\pi_0(1)} \rangle\!\rangle_1 \oplus S_1, ..., \langle\!\langle A_{\pi_0(n)} \rangle\!\rangle_1 \oplus S_n)$ to P_{r}.
3. P_{p} sends π_1 and $T \leftarrow \Sigma^m$ to P_{r} who outputs $\langle\!\langle A' \rangle\!\rangle_0 := \{B_{\pi_1(1)} \oplus T_1, ..., B_{\pi_1(m)} \oplus T_m\}$. P_{p} outputs $\langle\!\langle A' \rangle\!\rangle_1 := \{S_{\pi_1(1)} \oplus T_1 \oplus \langle\!\langle A_{\pi(1)} \rangle\!\rangle_0, ..., S_{\pi_1(m)} \oplus T_m \oplus \langle\!\langle A_{\pi(m)} \rangle\!\rangle_0\}$.

[Duplicate] Upon the command $(\textsc{Duplicate}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from P_{p} and $(\textsc{Duplicate}, \langle\!\langle A \rangle\!\rangle_1)$ from P_{s}. Require that $\pi : [n] \to [n]$ s.t $\pi(1) = 1$ and $\pi(i) \in \{i, \pi(i-1)\}$ for $i \in [2, n]$ and $A \in \Sigma^n$. Then:

1. P_{p} computes the vector $b \in \{0,1\}^m$ such that $b_1 = 0$ and for $i \in [2, n]$, $b_i = 1$ if $\pi(i) = \pi(i-1)$ and 0 otherwise.
2. P_{s} samples $\langle\!\langle B \rangle\!\rangle_1, W^0, W^1 \leftarrow \Sigma^n, \langle\!\langle B_1 \rangle\!\rangle_0 \leftarrow \Sigma$ and $\phi \leftarrow \{0,1\}^n$. P_{s} redefine $\langle\!\langle B_1 \rangle\!\rangle_1 := \langle\!\langle A_1 \rangle\!\rangle_1 \oplus \langle\!\langle B_1 \rangle\!\rangle_0$. For $i \in [2, n]$, P_{s} sends
$$M_i^0 := \langle\!\langle A_i \rangle\!\rangle_1 \quad \oplus \langle\!\langle B_i \rangle\!\rangle_1 \oplus W_i^{\phi_i}$$
$$M_i^1 := \langle\!\langle B_{i-1} \rangle\!\rangle_1 \oplus \langle\!\langle B_i \rangle\!\rangle_1 \oplus W_i^{\overline{\phi_i}}$$
and $\langle\!\langle B_1 \rangle\!\rangle_0, \phi$ to P_{p}. P_{s} sends $\langle\!\langle B \rangle\!\rangle_1, W^0, W^1$ to P_{r}.
3. P_{p} sends $\rho := \phi \oplus b, R \leftarrow \Sigma^n$ to P_{r} who responds with $\{W_i^{\rho_i} : i \in [2, n]\}$. For $i \in [2, n]$, P_{p} defines
$$\langle\!\langle B_i \rangle\!\rangle_0 := M_i^{b_i} \oplus W_i^{\rho_i} \oplus b_i \langle\!\langle B_{i-1} \rangle\!\rangle_0$$
P_{p} outputs $\langle\!\langle A' \rangle\!\rangle_0 := \langle\!\langle B \rangle\!\rangle_0 \oplus R \oplus \pi(\langle\!\langle A \rangle\!\rangle_0)$ and P_{r} outputs $\langle\!\langle A' \rangle\!\rangle_1 := \langle\!\langle B \rangle\!\rangle_1 \oplus R$.

[Switch] Upon the command $(\textsc{Switch}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from P_{p} and $(\textsc{Switch}, \langle\!\langle A \rangle\!\rangle_1)$ from P_{s} where $\pi : [m] \to [n]$ and $\langle\!\langle A \rangle\!\rangle_0, \langle\!\langle A \rangle\!\rangle_1 \in \Sigma^n$.

1. P_{p} samples an injection $\pi_1 : [m] \to [n]$ s.t. for $i \in \mathit{image}(\pi)$ and $k = |\mathit{preimage}(\pi, i)|$, $\exists j$ where $\pi_1(j) = i$ and $\{\pi_1(j+1), ..., \pi_1(j+k)\} \cap \mathit{image}(\pi) = \emptyset$. P_{p} sends $(\textsc{Permute}, \pi_1, \langle\!\langle A \rangle\!\rangle_0)$ to $\mathcal{F}_{\textsc{Permute}}$ and P_{s} sends $(\textsc{Permute}, \langle\!\langle A \rangle\!\rangle_1)$. P_{p} receives $\langle\!\langle B \rangle\!\rangle_0 \in \Sigma^m$ in response and P_{r} receives $\langle\!\langle B \rangle\!\rangle_1 \in \Sigma^m$.
2. P_{p} defines $\pi_2 : [m] \to [m]$ s.t. for $i \in \mathit{image}(\pi)$ and $k := |\mathit{preimage}(\pi, i)|$ and j where $\pi_1(j) = i$, then $\pi_2(j) = ... = \pi_2(j+k) = j$. P_{p} and P_{r} respectively send $(\textsc{Duplicate}, \pi_2, \langle\!\langle B \rangle\!\rangle_0)$ and $(\textsc{Duplicate}, \langle\!\langle B \rangle\!\rangle_1)$ to $\mathcal{F}_{\textsc{Duplicate}}$. As a result P_{p} obtains $\langle\!\langle C \rangle\!\rangle_0 \in \Sigma^m$ from $\mathcal{F}_{\textsc{Duplicate}}$ and P_{s} obtains $\langle\!\langle C \rangle\!\rangle_1 \in \Sigma^m$.
3. P_{p} computes the permutation $\pi_3 : [m] \to [m]$ such that for $i \in \mathit{image}(\pi)$ and $k = |\mathit{preimage}(\pi, i)|$, $\{\pi_3(\ell) : \ell \in \mathit{preimage}(\pi, i)\} = \{j, ..., j+k\}$ where $i = \pi_1(j)$. P_{p} sends $(\textsc{Permute}, \pi_3, \langle\!\langle C \rangle\!\rangle_0)$ to $\mathcal{F}_{\textsc{Permute}}$ and P_{s} sends $(\textsc{Permute}, \langle\!\langle C \rangle\!\rangle_1)$. P_{p} receives $S \in \Sigma^m$ in response. P_{p} and P_{r} respectively receives and outputs $\langle\!\langle A' \rangle\!\rangle_0, \langle\!\langle A' \rangle\!\rangle_1 \in \Sigma^m$.

Figure 5: The Oblivious Switching Network protocols $\Pi_{\textsc{Permute}}$, $\Pi_{\textsc{Duplicate}}$, $\Pi_{\textsc{Switch}}$.

$\rho = \phi \oplus b_i$ to P_{r}. The final shares[2] are constructed by having P_{r} send w_ρ to P_{p} who computes $\langle\!\langle A'_i \rangle\!\rangle_0 := m_{b_i} \oplus w_\rho$. In our computationally secure setting observe that sending w_0, w_1, ϕ can be optimized away using a seed.

The protocol just described considers the setting where the parties select between A_i, A_{i-1}. However, we require that at each iteration the messages being selected is either A_i or A'_{i-1} where $\langle\!\langle A'_{i-1} \rangle\!\rangle$ was computed in the previous iteration. Fortunately this can be achieved at no overhead by leveraging that fact that P_{p} knows b_i and $\langle\!\langle A'_{i-1} \rangle\!\rangle_0$ ahead of time. At index i, P_{s} uses $\langle\!\langle A'_{i-1} \rangle\!\rangle_1$ instead of A_{i-1} while the P_{p} computes $\langle\!\langle A'_i \rangle\!\rangle_0 := m_{b_i} \oplus w_\rho \oplus b_i \langle\!\langle A'_{i-1} \rangle\!\rangle_0$. As such, P_{p} is manually adding other other share of $\langle\!\langle A'_{i-1} \rangle\!\rangle$ when it is need. The full proof of security is in Section B.2. To briefly show correctness, let us assume by induction that $\langle\!\langle A'_{i-1} \rangle\!\rangle$ is correct, then:

$$\begin{aligned} A'_i &= (m_{b_i} \qquad\qquad\qquad\qquad\qquad \oplus w_\rho \oplus b_i \langle\!\langle A'_{i-1} \rangle\!\rangle_0) \oplus (\langle\!\langle A'_i \rangle\!\rangle_1) \\ &= (\overline{b_i} A_i \oplus b_i \langle\!\langle A'_{i-1} \rangle\!\rangle_1 \oplus \langle\!\langle A'_i \rangle\!\rangle_1 \oplus w_{b_i \oplus \phi} \oplus w_\rho \oplus b_i \langle\!\langle A'_{i-1} \rangle\!\rangle_0) \oplus (\langle\!\langle A'_i \rangle\!\rangle_1) \\ &= \overline{b_i} A_i \oplus b_i A'_{i-1} \end{aligned}$$

Universal Switching Network. Our Switch protocol of Figure 5 is a universal switching network for an *arbitrary* $\pi : [m] \to [n]$ and is constructed in three phases[MS13]: $A \xrightarrow{\pi_1} B \xrightarrow{\pi_2} C \xrightarrow{\pi_3} A' = \pi(A)$.

Let us first work through an example $\pi[6] \to [8]$ where $(\pi(1), ..., \pi(6)) = (4, 3, 4, 7, 4, 7)$. Observe that 4 appears three times, 7 appears twice and $\{1, 2, 5, 6, 8\}$ are "unused". First we apply the permutation $B = \pi_1(A)$ which ensures A_4 is followed by two unused elements and A_7 is followed by one, e.g. $B = (A_3, A_4, A_1, A_2, A_7, A_8)$. In general, an element that should appear $k > 1$ times will be followed by $k - 1$ unused elements. Note that $|B| = |A'|$ and need not be the size of A. This is achieved by dropping some unused elements, e.g. A_5, A_6. The duplication network $C = \pi_2(B)$ will duplicate A_4 twice and A_7 once, e.g. $C = (A_3, A_4, A_4, A_4, A_7, A_7)$. Note that only elements in A' are left now and they have the correct multiplicity. Finally, $A' = \pi_3(C)$ permutes C to be in the desired order, e.g. $A' = (A_4, A_3, A_4, A_7, A_4, A_7)$.

More specifically, the transformations are defined as:

1. $B := \pi_1(A)$: Sample an injective $\pi_1 : [m] \to [n]$ s.t. if π maps an input position i to k outputs positions (i.e. $k = |preimage(\pi, i)| = |\{j : \pi(j) = i\}|$), then there exists a j such that $\pi_1(j) = i$ and $\{\pi_1(j+1), ..., \pi_1(j+k-1)\} \cap image(\pi) = \emptyset$.

2. $C := \pi_2(B)$: Let $\pi_2 : [m] \to [m]$ s.t. if A_i is mapped to $k > 0$ positions in $A' = \pi(A)$, then for $\pi_1(j) = i$ it holds that $C_j = ... = C_{j+k-1} = A_i = B_j$. That is, $\pi_2(j) = ... = \pi_2(j + k - 1) = j$.

3. $A' := \pi_3(C)$: Sample permutation $\pi_3 : [m] \to [m]$ s.t. C is permuted to the same ordering as $\pi(A)$. That is, for $i \in image(\pi)$ and $\pi_1(j) = i$, sample

[2]Due to technique reasons about simulating output the full protocol additionally randomizes the output shares.

permutation π_3 s.t. $\{\pi_3(\ell) \mid \ell \in preimage(\pi, i)\} = \{j, ..., j + |preimage(\pi, i)| - 1\}$.

Observe that steps π_1, π_3 can both be implemented using the oblivious permutation protocol while π_2 can be implemented with a duplication network. Figure 5 provides a formal description of the full switching network protocol. The simulation of this protocol is presented in Section B.3 and follows from the simulation of the permutation and duplication subprotocols.

Comparison. We compare with alternative constructions to illustrate the performance improvement that our switching protocol provides. The first and most traditional is for P_{s} to use additive homomorphic encryption, e.g. Paillier, to encrypt the shares $\langle\!\langle A \rangle\!\rangle_1$ and send these to P_{p}. P_{p} can apply the mapping function π to these encryptions, rerandomize them and send the result back to P_{s} who decrypts it to obtain their share of $\langle\!\langle A' \rangle\!\rangle_1$. This approach has a very high computational overhead compared to ours due to additive homomorphic encryption being an intensive process.

An alternative approach is that taken by [MS13] which can be viewed as the two party version of our protocol. In their setting the permutation network is the most expensive operation and is implemented using $O(n \log n)$ OTs[IKNP03]. Our protocol is both asymptotically more efficient by a $O(\log n)$ factor and has smaller constants since our protocol does not require the relatively more expensive OT primitive.

3.4 Join Protocols

Our join protocol can be divided into four phases:

1. Compute randomized encodings of the join-columns/keys.
2. Party P_1 constructs a cuckoo table T for table Y and arranges the secret shares using a permutation protocol.
3. For each row x in X, P_0 uses an oblivious switching network to map the corresponding location i_1, i_2 of the cuckoo hash table to a secret shared tuple $(x, T[i_1], T[i_2])$.
4. The join-key(s) of x is compared to that of $T[i_1], T[i_2]$. If one of them match then the corresponding Y' row is populated; otherwise the Y' row is set to `NULL`.
5. The various types of joins can then be constructed by comparing row i of X and Y'.

Steps 1 through 4 are performed by the Map routine of Figure 6 while step 5 is performed in the Join routine. Figure 7 contains the ideal functionality of the join protocol.

Randomized Encodings. We begin by generating randomized encodings of the columns being used for the join-keys. For example, selecting all columns of X and Y where $X_1 = Y_1$ `and` $X_2 = Y_3$. In this case there are two join-keys, X_1, X_2 from X and Y_1, Y_3 from Y. The protocol has P_0 learn the randomized encoding for each row of X and P_1 learn them for Y. Importantly, is that after a previous join operation, some (or all) of the rows being joined can be `NULL`. We require that the randomized encodings of these rows not reveal that they are `NULL`. For table X, a special column $X_{\texttt{NULL}}$ encodes if for each row is logically `NULL`. The $\mathcal{F}_{\text{ENCODE}}$ functionality will then return a random encoding for all `NULL` rows. Specifically, the parties will send $(\text{ENCODE}, \{([\![X_{\texttt{NULL}}]\!], [\![X_{j_1}||...||X_{j_l}]\!], P_0), ([\![Y_{\texttt{NULL}}]\!], [\![Y_{k_1}||...||Y_{k_l}]\!], P_1)\})$ to $\mathcal{F}_{\text{ENCODE}}$ where $j_1, ..., j_l$ and $k_1, ..., k_l$ index the join-keys of X and Y. Let $\mathbb{E}_x, \mathbb{E}_y \in (\{0,1\}^\ell)^n$ be the encodings that P_0 and P_1 respectively receive from $\mathcal{F}_{\text{ENCODE}}$.

For correctness, we require the encoding bit-length ℓ to be sufficiently large such that the probability of a collision between encodings is statistically negligible. Given that there are a total of $D = 2n$ encodings, the probability of this is at most $2^{-\ell+2\log_2 D-1}$ which we require to be less than $2^{-\lambda}$, therefore $\ell \geq \lambda + 2\log_2 D - 1$. Our implementation uses $\lambda = 40$ and $\ell \in \{80, 100\}$ depending on D.

Constructing the Cuckoo Table. The next phase of the protocol is for P_1 to construct a secret shared cuckoo table for Y where each row is inserted based on its encoding in $\mathbb{E}_y$. P_1 locally inserts the encodings $\mathbb{E}_y$ into a plain cuckoo hash table t with $m \approx 1.5n$ slots using the algorithm specified in Section 2 and [DRRT18]. In the presentation we assume two hash functions are used. P_1 samples an injective function $\pi : [m] \to [m]$ such that $t[j] = \mathbb{E}_y[i]$, then $\pi(j) = i$.

Parties P_0 and P_1 convert $[\![Y]\!]$ to $\langle\!\langle Y \rangle\!\rangle$ such that P_0 holds $\langle\!\langle Y \rangle\!\rangle_0$. P_1 sends $(\text{SWITCH}, \pi, \langle\!\langle Y \rangle\!\rangle_1)$ to $\mathcal{F}_{\text{SWITCH}}$ and P_0 sends $(\text{SWITCH}, \langle\!\langle Y \rangle\!\rangle_0)$. In response $\mathcal{F}_{\text{SWITCH}}$ sends $\langle\!\langle T \rangle\!\rangle_1$ to P_1 and $\langle\!\langle T \rangle\!\rangle_0$ to P_2. It is now the case that T is a valid secret shared cuckoo hash table of Y.

Selecting from the Cuckoo Table.. The next phase of the protocol is for each row of X, select the appropriate rows of T so the keys can be compared. P_0 knows that if the join-keys of the $X[i]$ row will match with a row from Y, then this row will be at $T[j]$ for some $j \in \{h_1(e), h_2(e)\}$ where $e = \mathbb{E}_x[i]$.

To obliviously compare these rows, P_0 will construct two switching networks with programming $\pi_1, \pi_2 : [n] \to [m]$ such that if $h_l(\mathbb{E}_x[i]) = j$ then $\pi_l(i) = j$. Each of these will be used to construct the tables $\langle\!\langle \widehat{Y}^1 \rangle\!\rangle, \langle\!\langle \widehat{Y}^2 \rangle\!\rangle$ which are the result of applying the switching networks π_1, π_2 to $\langle\!\langle T \rangle\!\rangle$. For $i \in [n]$, the parties select either $\langle\!\langle \widehat{Y}^1 \rangle\!\rangle[i]$ or $\langle\!\langle \widehat{Y}^2 \rangle\!\rangle[i]$ and assign it to $[\![Y']\!][i]$ based on which has matching joins keys with $[\![X]\!][i]$. If there is no match then $[\![Y']\!][i] = \texttt{NULL}$.

Inner Join. Given the secret shared tables $[\![X]\!], [\![Y']\!]$ as described above, the parties do a linear pass over the n rows to construct the join between X and Y. Recall that the inner join consists of all the selected columns from the rows $X[i], Y[j]$ where the join-keys of the rows $X[i]$ and $Y[j]$ are equal.

If row $X[i]$ has a matching row in Y then this row will have been mapped to $Y'[i]$. Next, the `where` clause further filters the output table as a function of $Y'[i]$ and $X[i]$. The MPC protocol sets the `NULL`-bit of the final output table Z as $Z_{\texttt{NULL}}[i] := X_{\texttt{NULL}}[i] \vee Y'_{\texttt{NULL}}[i] \vee \neg P(Y'[i], X[i])$ where P is the predicate function specified by the `where` clause. Finally, the computation specified by the `select` query is performed, e.g. copying the columns of X, Y or computing a function of them.

Left/Right Join. A left join query is similar to an inner join except that all of the rows from the left table X are included. All rows that are in the inner join are computed as before. For rows only in X, the bit $Y'_{\texttt{NULL}}[i]$ will equal one and is used to initialize the missing columns from Y to a default, typically `NULL`. A right join can be implemented symmetrically.

Union and Set Minus.. Our framework is also capable of computing the union of two tables with respect to the join-keys. Specifically, we define the union operator as taking all of the rows from the left table and all of the rows from the right table that would not be present in the inner join. First we compute $Y \backslash X$ by only including $X[i]$ if $Y'[i]$ is `NULL`, i.e. $X[i]$ has no matching row in Y. The union of X and Y is then constructed as $(Y \backslash X) || X$ where the $||$ operator denotes the row-wise concatenation of X to the end of $Y \backslash X$.

Full Join.. We construct a full join as (X left join Y) union Y. The left join merge the rows in the intersection and the union includes the missing rows of Y. The overhead of this protocol is effectually twice that of the other protocols.

We note that under some restrictions on the tables being joined, a more efficient protocol for full joins can be achieved. We defer an explanation of this technique to Section 5.2.

Security. The simulation of these protocols directly follow from the composibility of the subroutines $\mathcal{F}_{\text{ENCODE}}$, $\mathcal{F}_{\text{SWITCH}}$ and $\mathcal{F}_{\text{MPC}}$. First, the output of $\mathcal{F}_{\text{ENCODE}}$ simply outputs random strings and it is therefore straightforward to simulate. $\mathcal{F}_{\text{SWITCH}}$ and $\mathcal{F}_{\text{MPC}}$ both output secret shared values. 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 $2^{-\lambda}$. See B.4 for a full proof of security.

3.5 Non-unique Join on Column

When values in the join-column are not unique within a single table, the security guarantees begin to erode. Recall that the randomized encodings for X, Y are revealed to P_0, P_1 respectively. Repeated values in the join-columns will lead to duplicate randomized encodings and therefore reveal their location. Learning the distribution of these duplicates reveals that the underlying table has the same distribution. In the event that only one of the tables contains duplicates, the core protocol can naturally be extended to compute the various join operations subject

Parameters: Table size n. For all command, X, Y are tables and $\{X_j \mid j \in J\}$ and $\{Y_k \mid k \in K\}$ are the join-keys of X and Y respectively. S, P are resp. the `select` and `where` function.

[Map] Upon receiving $(\textsc{Map}, [\![X]\!], J, [\![Y]\!], K)$ from all parties.

1. The parties send $(\textsc{Encode}, \{([\![X_{\texttt{NULL}}]\!], [\![X_{J_1}||...||X_{J_l}]\!], P_0), ([\![Y_{\texttt{NULL}}]\!], [\![Y_{K_1}||...||Y_{K_l}]\!], P_1)\})$ to $\mathcal{F}_{\textsc{encode}}$ where $l = |J| = |K|$. P_0 receives $\mathbb{E}_x$ and P_1 receives $\mathbb{E}_y$ from $\mathcal{F}_{\textsc{encode}}$.

2. P_1 constructs a cuckoo hash table t for the set $\mathbb{E}_y$. Define π_0 such that $\pi_0(j) = i$ where $\mathbb{E}_y[i] = t[j]$.

3. P_0 and P_1 convert $[\![Y]\!]$ to $\langle\!\langle Y \rangle\!\rangle$. P_1 sends $(\textsc{Permute}, \pi_0, \langle\!\langle Y \rangle\!\rangle_1)$ to $\mathcal{F}_{\textsc{Permute}}$ and P_0 sends $(\textsc{Permute}, \langle\!\langle Y \rangle\!\rangle_0)$. P_1 receives $\langle\!\langle T \rangle\!\rangle_1$ and P_2 receives $\langle\!\langle T \rangle\!\rangle_0$ from $\mathcal{F}_{\textsc{Permute}}$.

4. Let $h_1, ..., h_w$ be the cuckoo hash functions. P_1 defines $\pi_1, ..., \pi_w$ such that $\pi_l(i) = j$ where $h_l(\mathbb{E}_x[i]) = j$.

5. For $l \in [w]$, P_1 sends $(\textsc{Switch}, \pi_l, \langle\!\langle T \rangle\!\rangle_1)$ to $\mathcal{F}_{\textsc{Switch}}$ and P_2 sends $(\textsc{Switch}, \langle\!\langle T \rangle\!\rangle_0)$. P_0 receives $\langle\!\langle \widehat{Y}^l \rangle\!\rangle_0$ and P_1 receives $\langle\!\langle \widehat{Y}^l \rangle\!\rangle_1$ from $\mathcal{F}_{\textsc{Switch}}$.

6. For $i \in n$, if $\exists j \in [w]$ s.t. $\widehat{Y}^j_{\texttt{NULL}}[i] = 0 \vee [\![X_{J_1}||...||X_{J_l}]\!][i] = [\![\widehat{Y}^j_{K_1}||...||\widehat{Y}^j_{K_l}]\!][i]$ then $[\![\widehat{Y}^*]\!][i] := \langle\!\langle \widehat{Y}^j \rangle\!\rangle[i]$. Otherwise $[\![\widehat{Y}^*]\!][i] := (\texttt{NULL}, 0, ...)$. Output $[\![\widehat{Y}^*]\!]$.

[Join] Upon receiving command $(\textsc{Join}, \mathit{type}, [\![X]\!], J, [\![Y]\!], K, S, P)$ from all parties.

1. The parties send $(\textsc{Map}, [\![X]\!], J, [\![Y]\!], K)$ to $\Pi_{\textsc{map}}$ and receive $[\![Y']\!]$.

2. Output the table $[\![Z]\!]$ defined by the case *type*:

   - $\textsc{Inner}$: For $i \in [n]$, $\mathcal{F}_{\textsc{mpc}}$ evaluate $[\![Z_{\texttt{NULL}}]\!][i] := [\![X_{\texttt{NULL}}]\!][i] \vee [\![Y'_{\texttt{NULL}}]\!][i] \vee \neg P([\![X]\!][i], [\![Y']\!][i])$ and $[\![Z]\!][i] := S([\![X]\!][i], [\![Y']\!][i])$.
   - $\textsc{Left}$: For $i \in [n]$, $\mathcal{F}_{\textsc{mpc}}$ evaluate $[\![Z_{\texttt{NULL}}]\!][i] := [\![X_{\texttt{NULL}}]\!][i] \vee \neg P([\![X]\!][i], [\![Y']\!][i])$ and $[\![Z]\!][i] := S([\![X]\!][i], [\![Y']\!][i])$.
   - $\textsc{Union}$: For $i \in [n]$, $\mathcal{F}_{\textsc{mpc}}$ evaluate $[\![Z_{\texttt{NULL}}]\!][i] := [\![X_{\texttt{NULL}}]\!][i] \vee \neg P([\![X]\!][i], \texttt{NULL})$ and $[\![Z]\!][i] := S([\![X]\!][i], \texttt{NULL})$.
     For $i \in [n]$, $\mathcal{F}_{\textsc{mpc}}$ evaluate $[\![Z_{\texttt{NULL}}]\!][n+i] := [\![Y_{\texttt{NULL}}]\!][i] \vee \neg [\![X_{\texttt{NULL}}]\!][i] \vee \neg P(\texttt{NULL}, [\![Y']\!][i])$ and $[\![Z]\!][n+i] := S(\texttt{NULL}, [\![Y']\!][i])$.
   - $\textsc{Full}$: all parties sending $(\textsc{Join}, \textsc{Left}, [\![X]\!], J, [\![Y]\!], K, S', P)$ to $\Pi_{\textsc{join}}$ and receiving $[\![X']\!]$ in response. They then send $(\textsc{Join}, \textsc{Union}, [\![X']\!], J, [\![Y]\!], K, S'', P')$ to $\Pi_{\textsc{join}}$ and output the response, where S', S'' and P' are appropriately updated version of S, P.

Figure 6: Join protocols $\Pi_{\textsc{map}}$ and $\Pi_{\textsc{join}}$.

[Join] Upon receiving command $(\textsc{Join}, type, [\![X]\!], J, [\![Y]\!], K, S, P)$ from all parties. J, K index the join columns of X, Y respectively. Let n_X and n_Y denote the number of rows in X, Y respectively. Define $\textsc{Keys}(X, J, i) = (X_j[i])_{j \in J}$. Output the table $[\![Z]\!]$ defined by the case *type*:

- $\textsc{Inner}$: Let the rows of Z be $\{S(X[i], Y[j]) \mid \exists i, j \text{ s.t. } \neg X_{\texttt{NULL}}[i] \wedge \neg Y_{\texttt{NULL}}[j] \wedge \textsc{Keys}(X, J, i) = \textsc{Keys}(Y, K, j) \wedge P(X[i], Y[i])\}$ along with zero or more $\texttt{NULL}$ rows s.t. Z has n_X rows.

- $\textsc{Left}$: Let the rows of Z be $\{S(X[i], Y[j]) \mid \exists i, j \text{ s.t. } \neg X_{\texttt{NULL}}[i] \wedge \neg Y_{\texttt{NULL}}[j] \wedge \textsc{Keys}(X, J, i) = \textsc{Keys}(Y, K, j) \wedge P(X[i], Y[i])\} \cup \{S(X[i], \texttt{NULL}) \mid \exists i, \forall j \text{ s.t. } \neg X_{\texttt{NULL}}[i] \wedge \textsc{Keys}(X, J, i) \neq \textsc{Keys}(Y, K, j) \wedge P(X[i], \texttt{NULL})\}$ along with zero or more $\texttt{NULL}$ rows s.t. Z has n_X rows.

- $\textsc{Union}$: Let the rows of Z be $\{S(X[i], \texttt{NULL}) \mid \exists i \text{ s.t. } \neg X_{\texttt{NULL}}[i] \wedge P(X[i], \texttt{NULL})\} \cup \{S(\texttt{NULL}, Y[i]) \mid \exists i, \forall j \text{ s.t. } \neg Y_{\texttt{NULL}}[i] \wedge \textsc{Keys}(X, J, j) \neq \textsc{Keys}(Y, K, i) \wedge P(\texttt{NULL}, Y[i])\}$ along with zero or more $\texttt{NULL}$ rows s.t. Z has $n_X + n_Y$ rows.

- $\textsc{Full}$: Let the rows of Z be $\{S(X[i], Y[j]) \mid \exists i, j \text{ s.t. } \neg X_{\texttt{NULL}}[i] \wedge \neg Y_{\texttt{NULL}}[j] \wedge \textsc{Keys}(X, J, i) = \textsc{Keys}(Y, K, j) \wedge P(X[i], Y[i])\} \cup \{S(\texttt{NULL}, Y[i]) \mid \exists i, \forall j \text{ s.t. } \neg Y_{\texttt{NULL}}[i] \wedge \textsc{Keys}(X, J, j) \neq \textsc{Keys}(Y, K, i) \wedge P(\texttt{NULL}, Y[i])\} \cup \{S(X[i], \texttt{NULL}) \mid \exists i, \forall j \text{ s.t. } \neg X_{\texttt{NULL}}[i] \wedge \textsc{Keys}(X, J, i) \neq \textsc{Keys}(Y, K, j) \wedge P(X[i], \texttt{NULL})\}$

Figure 7: Join functionality $\mathcal{F}_{\text{JOIN}}$.

to P_0 learning the duplicate distribution. This is achieved by requiring the left table X contain the duplicates rows. After learning the randomized encodings for this table P_0 can program the switching networks appropriately to query the duplicate locations in the cuckoo hash table.

When both tables contain duplicates we fall back to a less secure protocol architecture. This is required due to the cuckoo table not supporting duplicates. First, P_1 samples two random permutations π_0, π_1 and computes $X' = \pi_1(X), Y' = \pi_2(Y)$ using the oblivious permutation protocol. P_0 then learns all of the randomized encodings for the permuted tables X' and Y'. Given this, P_0 can compute the size of the output table and inform the other two parties of it. Alternatively, an upper bound on the output table size can be communicated. Let n' denote this value. P_0 can then construct two switching networks which map the rows of X' and Y' to the appropriate rows of the output table. The main disadvantage of this approach is that P_0 learns the size of the output, the distribution of duplicate rows and how these duplicate rows are multiplied together. However, unlike [LTW13] which takes a conceptually similar approach, our protocol does not leak any information to P_1 and P_2, besides the value n'.

3.6 Revealing Results

Revealing a secret shared table $[\![X]\!]$ requires two operations. First observe that the data in the `NULL` rows is not cleared out by the join protocols. This is done as an optimization. As such naively reconstructing these rows would lead to significant leakage. Instead $X[i]$ is updated as $X[i] = (\neg X_{\texttt{NULL}}[i]) \cdot X[i]$. The second operation is to perform an oblivious shuffle of the rows. This operation randomly reorders all the rows without revealing the ordering to any of the parties. In general this step is necessary since the original ordering of the result table is input-dependent. For example, say X is a list of patents info, Y is patent billing status, and Z is a list of patent diseases. Say we reveal `select` $X.name, Y.balance$ `from` X, Y `on` $X.id = Y.id$ and `select` $X.gender, Z.desease$ `from` X, Z `on` $X.id = Z.id$. Without reordering you could connect $X.name, X.gender, Y.balance$ and $Z.desease$ by the row index and infer secret information. However, by randomly shuffling this connection is destroyed and the reveal can be simulated.

4 Computing a Function of a Table

In addition to join queries, our framework can perform computation on a single secret shared table. For example, selecting $X_1 + X_2$ where $X_3 > 42$. For each row i we generate the corresponding output row $Z[i]$ by computing the new `NULL`-bit as $Z_{\texttt{NULL}}[i] := X_{\texttt{NULL}}[i] \vee P(X[i])$ where $P(\cdot)$ is the `where` predicate. The new column(s), e.g. $Z_1 = X_1 + X_2$, can then be constructed in a straightforward MPC protocol, e.g. [MR18, AFL+16]. The key property is that all of the operations are with respect to

a single row of X, allowing them to be evaluated in parallel.

Our framework also considers a second class of functions on a table that allow computation between rows. For example, computing the sum of a column. We refer to this broad class of operations as an aggregation function. Depending on the exact computation, various levels of efficiencies can be achieved. Our primary approach is to employ the ABY^3 framework [MR18] to express the desired computation in an efficient way and then to evaluate the resulting circuit. Next we highlight a sampling of some important aggregation operations:

- Sum: For a column $[\![X_j]\!]$, compute $[\![s]\!] = \sum_i [\![X_j]\!][i]$ where $X_j[i] \in \mathbb{Z}_{2^\ell}$ and i indexes only non-`NULL` rows. The parties compute $[\![s]\!]^{\mathsf{A}} := \sum_{i\in[n]} \mathsf{B2A}((\neg[\![X_{\texttt{NULL}}]\!][i]) \cdot [\![X_j]\!][i])$ where $\mathsf{B2A}$ is the boolean to arithmetic share conversion of [MR18]. In total this requires $2n\ell$ binary gates and $\ell + 1$ rounds[MR18]. The parties can then convert $[\![s]\!]^{\mathsf{A}}$ back to $[\![s]\!]$ if desired.

- Count/Cardinality: Here, we consider two cases. 1) In the general case there is an arbitrary table over which the count is being computed. This is performed by computing $[\![s]\!]^{\mathsf{A}} := \sum_{i\in[n]} \mathsf{B2A}(\neg[\![X_{\texttt{NULL}}]\!][i])$

  2) Consider case where some of the parties should learn the cardinality of a join without a `where` clause. First, w.l.o.g. let us assume that P_2 should learn the cardinality. The randomized encodings $\mathbb{E}_x, \mathbb{E}_y$ are respectively revealed P_0 and P_1 as done in the standard join protocol. These encodings are then sent to P_2 in a random order. P_2 outputs $|\mathbb{E}_x \cap \mathbb{E}_y|$ as the count/cardinality. In the event that P_0 or P_1 should also learn the cardinality, P_2 sends $|\mathbb{E}_x \cap \mathbb{E}_y|$ to them.

- Min/Max: We propose a recursive algorithm where the min/max of the first and second half of the rows is recursively computed. The final result is then the min/max of these two values. Concerning `NULL` rows, the corresponding value can be initialized to a maximum or minimum sentential value which guarantee that the other value will be propagated. The overall complexity of this approach is $O(n\ell)$ binary gates and $O(\ell log n)$ rounds when using a basic comparison circuit[MR18].

More generally, any polynomial time function can generically be expressed using the ABY^3 framework[MR18]. However, the resulting efficiency may not be adequate for practical deployment.

5 Applications

5.1 Voter Registration

Improving the privacy and integrity of the United States voter registration system was a primary motivation of the developed protocols. In the United States Electoral

College, each state has the responsibility of maintaining their own list of registered citizens. A shortcoming of this distributed process is that without coordination between states it is possible for a voter to register in more than one state. If this person then went on to cast more than one vote the integrity of the system would be compromised. In the case of double registering, it is often a result of a person moving to a new state and failing to unregister from the old state. Alternatively, when a voter moves to a new state it may take them some time to register in the new state, and as such their vote may go uncast. The Pew Charitable Trust[Smi14] reported 1 in 8 voter registration records in the United States contains a serious error while 1 in 4 eligible citizens remain unregistered. The goal in this application of our framework is to improve the accuracy of the voting registration data and help register eligible voters.

A naive solution to this problem is to construct a centralized database of all the registered voters and citizen records. It is then a relatively straightforward process to identify persons with inaccurate records, attempt to double register or are simply not register at all. However, the construction of such a centralized repository of information has long had strong opposition in the United States due to concerns of data privacy and excessive government overreach. As a compromise many states have volunteered to join the Electronic Registration Information Center (ERIC)[eri18] which is a non-profit organization with the mission of assisting states to improve the accuracy of America's voter rolls and increase access to voter registration for all eligible citizens. This organization acts as a semi-trusted third party which maintains a centralized database containing hashes of the relevant information, e.g. names, addresses, drivers license number and social security number.

We propose adding another layer of security with the deployment of our secure database join framework. Within a single state, different agencies will first secret share their data to construct a join table containing the registration status of everyone within that state. This joined table can then be joined with the respective table from all of the other stated. In total, there would be 50 intra-state joins and then 50×49 inter-state joins.

We envision that the intra-state join will be perform with ERIC and the state agencies as the participating parties. The inter-state joins can then be performed by ERIC and one of the agencies from each state. This ensures that the data remains secret shared at all times. The data that each state requires can then be revealed at the end of the computation. For more details see Appendix A.

The average US state has an approximate population of 5 million with about 4 million of that being of voting age. For this set size, our protocol is capable of performing the specified query in 30 seconds and 6GB of total communication. If we consider running the same query where one of the states is California with a voting population of 30 million, our protocol can identify the relevant records in five minutes. For a more details see Section 6.

5.2 Threat Log Comparison

Another motivating application is referred to as threat log comparison where multiple organizations share data about current attacks on their computer networks. The goal of sharing this data is to allow the participating parties to identify and stop threats in a more timely manner. Facebook has a service called ThreatExchange[thr18] which provides this functionality. One drawback of the Facebook approach is that all of the data is collected on their servers and is often viewable by the other participants. This architecture inherently relies on trusting Facebook with this data.

We propose using our distributed protocol to provide a similar functionality while reducing the amount of trust in any single party, e.g. Facebook. In this setting we consider a moderate number of parties each holding a dataset containing the suspicious events on their network along with possible meta data on that event, e.g. how many times that event occurred. All of the parties input these sets into our join framework where the occurrences of each event type are counted. An example of such an event is the IP address that makes a suspicious request.

There are at least two ways to securely compute the occurrences of these events. One method is to perform a full join of all the events where the counts are added together during each join. The resulting table would contain all of the events and the number of times that each event occurred. The drawback of this approach is that each full joins require performing a left join followed by a union, twice the overhead compared to other join operations.

Now consider a different strategy for this problem. First, the parties can compute and reveal the union of the events. Given this information the parties can locally compute the number of times this event occurred on their network and secret share this information between the parties. The parties then add together this vector of secret shared counts and reveal it.

One shortcoming of this approach is no ability to limit which events are revealed. For example, it can be desirable to only reveal an event if it happens on k out of the n networks. This can be achieved by having the parties compute and reveal the randomized encodings for all of the items in the union, instead of the items themselves. Under the same encoding key, each party holding a set employs the three server parties to compute the randomized encodings for the items in their set. These encodings are revealed to the party holding the set. For each encoding in the union, the parties use the MPC protocol to compute the number of occurrences that event had and conditionally reveal the value. For example, if at least k of the networks observed the event. Other computation on meta data can also be performed as this stage.

6 Experiments

We implemented our full set of protocols and applications along with a performance evaluation of them here. They will be open source. We considered set intersection (without associated values), our various join and union operations, intersection cardinality, and intersection sum of key-value pairs along with the two application described in Section 5. We also compare to protocols that offer a similar functionality, i.e. [KKRT16, PSWW18, BA12, CGT12, IKN+17]. Our implementation is written in c++ and building on primitives provided by [Rin]. Crucial to the performance of implementation is the widespread use of SIMD instructions that allow processing 128 binary gates with a throughput of one cycle.

Experimental Setup. We performed all of our experiments on a single server acquired in 2015 which is equipped with two 18 core CPUs at 2.7 GHz and 256 GB of RAM. Despite having many cores, our implementation restricts each party to a *single thread*. We note this is a limitation of development time/resources and not of the protocols themselves. The parties communicate over a loopback device on the local area network which allows to shape the traffic flow to emulate a LAN and WAN setting. Specifically, the LAN setting allows 10 Gbps throughput with a latency of a quarter millisecond while the WAN setting allows an average 100 Mbps and 40 millisecond latency. Despite having such a fast LAN bandwidth, our protocol only utilizes a peak bandwidth of 1Gbps.

All cryptographic operations are performed with computational security parameter $\kappa = 128$ and statistical security $\lambda = 40$. We consider set/table sizes of $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$ and $n = 2^{26}$ in some cases. Times are reported as the average of several trials.

Set Intersection. We first consider set intersection. In this case the two tables of our protocol consist of a single column which is used as the join-key. We compare our protocol to [KKRT16] which is a two party set intersection protocol where the input sets each are known in the clear to one of the parties and one party learns the intersection exactly. This is contrasted by our three party protocol where the input and output sets are secret shared between the parties. That is, our protocol is composable & supports outsourced MPC while [KKRT16] does not and can not be trivially modified to do so without a large overhead. Both our protocol and [KKRT16] were benchmarked on the same hardware. We also compare to the three party protocol of [BA12] which is composable and was not benchmarked on the same hardware. Due to the code of the [BA12] protocol not being publicly available, we cite their benchmarks which were performed on three AMD Opteron computers at 2.6GHz connected on a 1Gbps LAN network. Given the relative performance of our machines, we believe this to yield a fair comparison. This protocol first sorts the two input sets/tables which in practice requires $O(n \log^2 n)$ overhead[BA12]. In contrast, our protocol and [KKRT16] has $O(n)$ overhead and $O(1)$ rounds.

This asymptotic difference also translates to a large difference in the concrete

Operation	Protocol, # Parties	LAN Time (sec.) n 2^8	2^{12}	2^{16}	2^{20}	2^{24}	WAN Time (sec.) n 2^8	2^{12}	2^{16}	2^{20}	2^{24}	Total Communication (MB) n 2^8	2^{12}	2^{16}	2^{20}	2^{24}
Intersection	This ,3	0.02	0.03	0.2	4.9	117	2.3	2.5	6.4	41.4	902	0.2	3.0	48.1	769.4	12,318
	[KKRT16] ,2	0.2	0.2	0.4	3.8	58	0.6	0.6	1.3	7.5	106	0.04	0.5	8.1	127.2	1,955
	[BA12]* ,3	2.9	23.4	374.4	*5,990.4	*95,846	–	–	–	–	–	–	–	–	–	–
Joins/Union	This ,3	0.02	0.03	0.3	9.1	192	2.6	2.9	6.6	61.4	1,337	0.3	4.9	78.1	1,249.4	19,998
	[LTW13]* ,3	2.0	8.0	128.0	*2,048.0	*32,768	–	–	–	–	–	–	–	–	–	–
Cardinality	This ,3	0.01	0.02	0.2	3.1	74	1.1	1.1	1.8	15.8	267	0.1	2.0	32.6	521.5	8,344
	[PSWW18]a ,2	*0.1	2.2	9.1	86.6	*1385	–	10.0	45.3	389.9	*6,238	–	52.7	826.1	9,971.4	*159,542
	[PSWW18]b ,2	–	–	–	–	–	–	13.0	56.2	*899.2	*14,387	–	14.3	171.3	*2,740.8	*43,852
	[CGT12] ,2	1.0	16.0	262.0	4190.0	67,100	–	–	–	–	–	0.1	0.4	6.2	99.0	1,584
Sum	This ,3	0.03	0.04	0.3	6.8	158	3.7	4.0	7.9	51.0	1,099	0.3	2.0	33.1	526.5	8,372
	[IKN+17] ,2	7.0	115.0	1,860.0	29,700.0	475,000	–	–	–	–	–	0.1	1.9	30.2	483.0	7,728
Voter Intra-state	This ,3	0.01	0.02	0.2	4.7	114	1.0	1.0	2.2	27.1	456	0.2	3.4	54.1	867.1	13,903
Voter Inter-state	This ,3	0.01	0.02	0.3	7.0	134	1.6	1.6	4.0	45.4	747	0.4	5.7	91.3	1,463.9	23,482
Threat Log $N = 2$	This ,3	0.02	0.03	0.2	5.1	121	2.4	2.5	4.8	34.6	585	0.2	3.1	50.2	804.2	12,867
Threat Log $N = 4$	This ,3	0.05	0.09	0.9	17.9	388	6.6	6.8	13.1	108.7	1,739	0.6	9.7	155.4	2,487.8	39,804
Threat Log $N = 8$	This ,3	0.10	0.19	1.7	47.1	1,021	14.9	15.3	30.0	264.3	4,228	1.4	22.8	365.7	5,854.9	93,677

Figure 8: The running time in seconds and communication overhead in MB for various join operations and application. The input tables each contain n rows. The [PSWW18] protocol has two implementation where [PSWW18]b is optimized for the WAN setting. – denotes that the running time is not available. * denotes that the running times were linearly extrapolated from the values of n provided by the publication.

running time as shown in Figure 8. Out of these three protocol [KKRT16] is the fastest requiring 3.8 seconds in the LAN setting to intersect two sets of size $n = 2^{20}$ while our protocol requires 4.9 seconds. However, our protocol is fully composable while [KKRT16] is not. Considering this we argue that a slowdown of 1.28× is acceptable. When compared to [BA12] which provides the same composable functionality, our protocol is estimated[3] to be 1220× faster.

In the WAN setting our protocol has a relative slowdown compared to [KKRT16]. This can be contributed to our protocol requiring more rounds and communication. For instance, with $n = 2^{20}$ the protocol of [KKRT16] in the WAN setting requires 7.5 seconds while our protocol requires 41 seconds, a difference of 5.5×. With respect to the communication overhead, our protocol for $n = 2^{20}$ requires 769 MB of communication and [KKRT16] requires 127 MB, a difference of 6×. The WAN running time and communication overhead of [BA12] is not known due to their code not being publicly available.

Joins/Union. The second point of comparison is performing an inner join protocol on two tables consisting of five columns of 32-bit values. We note that [BA12] is capable of this task but no performance results were available. Instead we compare with the join protocol of [LTW13]. This protocol is composable but requires that the cardinality of the intersection be revealed after each join is performed. As previously discussed, this leakage limits the suitability of the protocol in many applications. The numbers reported for [LTW13] are from their paper and the experiments were performed on three servers each with 12 CPUs at 3GHz in the LAN setting. As

[3] We linearly extrapolate the overhead of their protocol, despite having $O(n \log n)$ complexity.

can be seen in Figure 8, we estimate[4] our protocol is roughly 200× faster in the LAN setting. For example, with $n = 2^{20}$ our protocol requires a running time of 9.1 seconds while [LTW13] requires a running time of 2048 seconds. Moreover, our protocol scales quite well with the addition of these extra four columns as compared to a intersection protocol. For example, in the WAN setting an intersection with $n = 2^{20}$ requires 41 seconds while the addition of the four columns results in a running time of 61 seconds. For both protocols, operations such as left join and unions can be performed with little to no additional computation as compared to inner join.

We observed the following relative performance of the various operations of our protocol. Secret sharing the input tables tool 3% of the time, computing the randomized encodings via Π_{ENCODE} required 50%, constructing the cuckoo hash table via Π_{PERMUTE} required 6%, selecting the rows from the cuckoo table required 26%, and the final circuit computation via $\mathcal{F}_{\text{MPC}}$ required 14%. These percentages were obtained for $n = 2^{20}$ in the LAN setting and hold relatively stable regardless of n.

Cardinality. The set cardinality protocol presented here also outperforms all previous protocols. As described in Section 4, our cardinality protocol allows the omission of the switching network which reduces the amount of communication and overall running time. We demonstrate the performance by comparing with the two-party protocols of Pinkas et al. [PSWW18] and De Cristofaro et al. [CGT12]. The protocol of [PSWW18] was benchmarked on two multi-core i7 machines at 3.7GHz and 16GB of RAM with similar network settings. For the protocol of [CGT12], we performed rough estimates on the time required for our machine to perform the computation without any communication overhead. For sets of size $n = 2^{20}$ our protocol requires 3.1 seconds in the LAN setting and 15.8 in the WAN setting. The next fastest protocol is [PSWW18] which requires 86.6 seconds in the LAN setting and 390 seconds in the WAN setting. In both cases this represents more than a 20× difference in running time. [PSWW18] considers a variant of their protocol optimized for the WAN setting which reduces their communication at the expense of increased running time. The protocol of [CGT12] requires the most running time by a large margin due to the protocol being based on exponentiation. Just to locally perform these public key operations requires roughly 4200 seconds of computation on our benchmark machine, a difference of 1350×. However, the protocol of [CGT12] also requires the least amount of communication, consisting of 99MB for $n = 2^{20}$ while our protocol requires 521MB followed by [PSWW18] with almost 10GB.

Sum. The last generic comparison we perform is for securely computing the weighted sum of the intersection. Our protocol for performing this task is described in Section 4. We compare to the protocol of Ion et al. [IKN+17] which is the protocol behind Google's Join-and-Compute. This protocol can be viewed as an extension of the public key based cardinality protocol of [CGT12]. In particular, [IKN+17] also

[4] Again, we linearly extrapolate the overhead of the protocol.

revealed the cardinality of the intersection and then performs a secondary computation using Paillier homomorphic encryption to compute the sum. Although this protocol reveals more information than ours, we still think it is a valuable point of comparison. Not surprisingly, the protocol of [IKN+17] requires significantly more computation time than our protocol. For a dataset size of $n = 2^{20}$, their protocol requires almost 30000 seconds to just perform the public key operations without any communication. Our protocol requires just 6.8 seconds in the LAN setting and 51 in the WAN setting. Both of these protocols also consume roughly the same amount of communication with [IKN+17] requiring 483 MB and our protocol requires 527 MB, a increase of just 9 percent.

Voter Registration. We now turn our attention to the application of auditing the voter registration data between and within the states of the United States as described in Section 5.1 & Appendix A. In summary, this application checks that a registered voter is not registered in more than one state and cross validates that their current address is correct. Only the identities of the voters which have conflicting data are revealed to the appropriate state to facilitate a process to contact the individual. In addition, the application can be extended to assist the process of enrolling unregistered citizens. This audit process is performed using two types of join queries. First, each state computes a left join between the DMV database and the list of registered voters. In Figure 9 we call this join *Voter Intra-state*. For all pairs of states, these tables are then joined to identify any registration error, e.g. double registered. This join is referred to as *Voter Inter-state*. Performance metrics are reported for each of these joins individually and then we estimate the total cost to perform the computation nation wide.

As shown in Figure 9, our protocol can perform the *Voter Intra-state* join with an input set size of 16 million voters ($n = 2^{24}$) in 115 seconds on a LAN network and in 456 seconds on a WAN network. Considering all but three states have a voting population less than $n = 2^{24}$, we consider this a realistic estimate on the running time overhead. Our protocol also achieves relatively good communication overhead of 13.9 GB, where each of the servers sends roughly one third of this. On average, that is 830 bytes for each of the n records. Given these tables, the *Voter Inter-state* join is performed between all pairs of states. For two states with $n = 2^{24}$, the benchmark machine required 135 seconds in the LAN setting and 748 seconds in the WAN setting. The added overhead in this second join protocol is an additional `where` clause which requires a moderate sized binary circuit to be securely evaluated. This join requires 23.4 GB of communication.

Given the high value and low frequency of this computation we argue that these computational overheads are very reasonable. Given the current population estimates of each state, we extrapolate that the overall running time to run the protocol between all pairs of 50 states in a LAN setting would be 53,340 seconds (14.8 hours) or 285,687 seconds (about 80 hours) in the WAN setting. However, the running time in the WAN setting could easily be reduced by running protocols in parallel

and increasing the bandwidth above the relatively low 100Mbps per party. The total communication overhead is 9,131 GB which is the main bottleneck. While this amount of communication is non-negligible, the actual dollar amount on a cloud such as AWS[aws18] is relatively low (given the importance of the computation), totaling roughly $820[Has17].

Threat Log. In this application N party secret share their data between the three computational parties and delegate the task of identifying the events that appear in at least k out of the N data sets. As described in Section 5.2, the protocol proceeds by taking the union of the sets and then the number of times each event occurred is counted and compared against k. Each event that appears more than k times is then revealed to all parties. The union protocol can only function with respect to two input sets. To compute the union of N sets we use a binary tree structure where pairs of sets are combined. As such, there are a total of $N-1$ union operations and a depth of $\log N$ protocol instances.

When benchmarking we consider $N = \{2, 4, 8\}$ input sets each of size $n \in \{2^8, 2^{12}, 2^{16}, 2^{20}, 2^{24}\}$. Since we do not reveal the size of the union, the final table will be of size nN. For $N = 2$ sets each with $n = 2^{24}$ items our protocol requires 121 seconds in the LAN setting and 586 seconds in the WAN setting. The total communication is 804MB, or approximately 24 bytes per record. If we increase the number of sets to $N = 8$ we observe that the LAN running time increases to 1,021 seconds and 4,228 seconds in the WAN setting. Given the the total input size increased by $4\times$, we observe roughly an $8\times$ increase in running time. This difference is due to each successive union operation being twice as big. Theoretically the running time and communication of this protocol is $O(nN \log N)$.

References

[AFL+16] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 805–817. ACM, 2016.

[alt18] Alienvault open threat exchange (otx). 2018.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *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, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454. Springer, 2015.

[aws18] Amazon web services. 2018.

[BA12] Marina Blanton and Everaldo Aguiar. Private and oblivious set and multiset operations. pages 40–41, 2012.

[BEE+17] Johes Bater, Gregory Elliott, Craig Eggen, Satyender Goel, Abel Kho, and Jennie Rogers. Smcql: Secure querying for federated databases. *Proc. VLDB Endow.*, 10(6):673–684, February 2017.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. Prio: Private, robust, and scalable computation of aggregate statistics. In Aditya Akella and Jon Howell, editors, *14th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2017, Boston, MA, USA, March 27-29, 2017*, pages 259–282. USENIX Association, 2017.

[CGT12] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and private computation of cardinality of set intersection and union. In Josef Pieprzyk, Ahmad-Reza Sadeghi, and Mark Manulis, editors, *Cryptology and Network Security, 11th International Conference, CANS 2012, Darmstadt, Germany, December 12-14, 2012. Proceedings*, volume 7712, pages 218–231. Springer, 2012.

[CHLR18] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. 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.

[CLR17] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1243–1255. ACM, 2017.

[DRRT18] Daniel Demmler, Peter Rindal, Mike Rosulek, and Ni Trieu. Pir-psi: Scaling private contact discovery. *Proceedings on Privacy Enhancing Technologies*, 2018(4), 2018.

[DSS12] Anupam Datta, Divya Sharma, and Arunesh Sinha. Provable de-anonymization of large datasets with sparse dimensions. In Pierpaolo Degano and Joshua D. Guttman, editors, *Principles of Security and*

Trust, pages 229–248, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[eri18] Electronic registration information center, inc. 2018.

[Fac20] Facebook. Crypten: A research tool for secure machine learning in pytorch. 2020.

[Has17] Hasham. Aws data transfer costs and how to minimize them. 2017.

[HFH99] Bernardo A. Huberman, Matt Franklin, and Tad Hogg. Enhancing privacy and trust in electronic communities. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, pages 78–86, New York, NY, USA, 1999. ACM.

[IKN+17] Mihaela Ion, Ben Kreuter, Erhan Nergiz, Sarvar Patel, Shobhit Saxena, Karn Seth, David Shanahan, and Moti Yung. Private intersection-sum protocol with applications to attributing aggregate ad conversions. Cryptology ePrint Archive, Report 2017/738, 2017. https://eprint.iacr.org/2017/738.

[IKNP03] Yuval Ishai, Joe Kilian, Kobbi Nissim, and Erez Petrank. Extending oblivious transfers efficiently. pages 145–161, 2003.

[KKRT16] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. Efficient batched oblivious PRF with applications to private set intersection. pages 818–829, 2016.

[KLS+17] Ágnes Kiss, Jian Liu, Thomas Schneider, N. Asokan, and Benny Pinkas. Private set intersection for unequal set sizes with mobile applications. *PoPETs*, 2017(4):177–197, 2017.

[KMP+17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. Practical multi-party private set intersection from symmetric-key techniques. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 1257–1272. ACM, 2017.

[LTW13] Sven Laur, Riivo Talviste, and Jan Willemson. From oblivious aes to efficient and secure database join in the multiparty setting. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security*, ACNS'13, pages 84–101, Berlin, Heidelberg, 2013. Springer-Verlag.

[MD20] Yann Dupis et. al Morten Dahl, Justin Patriquin. Tf encrypted: Encrypted deep learning in tensorflow. 2020.

[Mea86] C. Meadows. A more efficient cryptographic matchmaking protocol for use in the absence of a continuously available third party. In *1986 IEEE Symposium on Security and Privacy*, pages 134–134, April 1986.

[Mer12] Martin M. Merener. Theoretical results on de-anonymization via linkage attacks. *Trans. Data Privacy*, 5(2):377–402, August 2012.

[MR18] Payman Mohassel and Peter Rindal. ABY3: A mixed protocol framework for machine learning. *IACR Cryptology ePrint Archive*, 2018:403, 2018.

[MS13] Payman Mohassel and Seyed Saeed Sadeghian. How to hide circuits in MPC an efficient framework for private function evaluation. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 557–574. Springer, 2013.

[NS06] Arvind Narayanan and Vitaly Shmatikov. How to break anonymity of the netflix prize dataset. *CoRR*, abs/cs/0610105, 2006.

[OGE16] Efe Onaran, Siddharth Garg, and Elza Erkip. Optimal de-anonymization in random graphs with community structure. In *2016 37th IEEE Sarnoff Symposium, Newark, NJ, USA, September 19-21, 2016*, pages 1–2. IEEE, 2016.

[OOS17] Michele Orrù, Emmanuela Orsini, and Peter Scholl. Actively secure 1-out-of-n ot extension with application to private set intersection. In Helena Handschuh, editor, *Topics in Cryptology – CT-RSA 2017: The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14–17, 2017, Proceedings*, pages 381–396, Cham, 2017. Springer International Publishing.

[PSSZ15] Benny Pinkas, Thomas Schneider, Gil Segev, and Michael Zohner. Phasing: Private set intersection using permutation-based hashing. In Jaeyeon Jung and Thorsten Holz, editors, *24th USENIX Security Symposium, USENIX Security 15, Washington, D.C., USA, August 12-14, 2015.*, pages 515–530. USENIX Association, 2015.

[PSWW18] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based PSI via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT*

2018 - 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29 - May 3, 2018 Proceedings, Part III, volume 10822 of *Lecture Notes in Computer Science*, pages 125–157. Springer, 2018.

[PSZ14] Benny Pinkas, Thomas Schneider, and Michael Zohner. Faster private set intersection based on OT extension. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 797–812, San Diego, CA, 2014. USENIX Association.

[PSZ16] Benny Pinkas, Thomas Schneider, and Michael Zohner. Scalable private set intersection based on OT extension. Cryptology ePrint Archive, Report 2016/930, 2016. http://eprint.iacr.org/2016/930.

[RA18] Amanda Cristina Davi Resende and Diego F. Aranha. Faster unbalanced private set intersection. 2018.

[RH18] Eric Rescorla Robert Helmer, Anthony Miyaguchi. Testing privacy-preserving telemetry with prio. 2018.

[Rin] Peter Rindal. libOTe: an efficient, portable, and easy to use Oblivious Transfer Library. https://github.com/osu-crypto/libOTe.

[RSC+19] M. Sadegh Riazi, Mohammad Samragh, Hao Chen, Kim Laine, Kristin E. Lauter, and Farinaz Koushanfar. XONN: xnor-based oblivious deep neural network inference. In Nadia Heninger and Patrick Traynor, editors, *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1501–1518. USENIX Association, 2019.

[Smi14] Aaron Smith. 6 new facts about facebook. Pew Research Center Fact Tank, 2014. http://www.pewresearch.org/fact-tank/2014/02/03/6-new-facts-about-facebook/.

[thr18] Facebook threat exchange. 2018.

[WGC19] Sameer Wagh, Divya Gupta, and Nishanth Chandran. Securenn: 3-party secure computation for neural network training. *PoPETs*, 2019(3):26–49, 2019.

[ZW18] Yuchen Zhao and Isabel Wagner. POSTER: evaluating privacy metrics for graph anonymization and de-anonymization. In Jong Kim, Gail-Joon Ahn, Seungjoo Kim, Yongdae Kim, Javier López, and Taesoo Kim, editors, *Proceedings of the 2018 on Asia Conference on Computer and Communications Security, AsiaCCS 2018, Incheon, Republic of Korea, June 04-08, 2018*, pages 817–819. ACM, 2018.

Application	LAN						WAN					
	n						n					
	2^8	2^{12}	2^{16}	2^{20}	2^{24}	2^{26}	2^8	2^{12}	2^{16}	2^{20}	2^{24}	2^{26}
Voter Intra-state	0.01	0.02	0.2	4.7	114.7	2,190.1	1.0	1.0	2.2	27.1	456.1	7,463.9
Voter Inter-state	0.01	0.02	0.3	7.0	134.8	2,546.4	1.6	1.6	4.0	45.4	747.7	12,284.1
Threat Log $N = 2$	0.02	0.03	0.2	5.1	121.4	488.1	2.4	2.5	4.8	34.6	585.6	2,342.4
Threat Log $N = 4$	0.05	0.09	0.9	17.9	388.4	1,553.9	6.6	6.8	13.1	108.7	1,739.2	6,956.8
Threat Log $N = 8$	0.10	0.19	1.7	47.1	1,021.0	16,336.1	14.9	15.3	30.0	264.3	4,228.8	16,915.2

Figure 9: The running time in seconds for the Voter Registration and Threat Log applications. The input tables each contain n rows.

A Voter Query Details

Given the problem statement from 5.1, a naive solution is to construct a centralized database of all the registered voters and citizen records. It is then a relatively straightforward process to identify persons with inaccurate records, attempt to double register or are simply not register at all. However, the construction of such a centralized repository of information has long had strong opposition in the United States due to concerns of data privacy and excessive government overreach. As a compromise many states have volunteered to join the Electronic Registration Information Center (ERIC)[eri18] which is a non-profit organization with the mission of assisting states to improve the accuracy of America's voter rolls and increase access to voter registration for all eligible citizens. This organization acts as a semi-trusted third party which maintains a centralized database containing hashes of the relevant information, e.g. names, addresses, drivers license number and social security number.

In particular, instead of storing this sensitive information in plaintext, all records are randomized using two cryptographically strong salted hash functions. Roughly speaking, before this sensitive information is sent to ERIC, each state is provided with the first salt value $salt_1$ and updates each value v as $v := H(salt_1||v)$. This hashed data is then sent to ERIC where the data is hashed a second time by ERIC which possesses the other salt value. The desired comparisons can then be applied to the hashed data inside ERIC's secure data center. When compared with existing alternative, this approach provides a moderate degree of protection. In particular, so long as the salt values remain inaccessible by the adversary, deanatomized any given record is likely non-trivial. However, a long series of works, e.g. [NS06, Mer12, DSS12, OGE16, ZW18], have shown that a significant amount of information can be extracted with sophisticated statistical techniques. Moreover, should the adversary possess the salt values a straightforward dictionary attack can be applied.

We propose adding another layer of security with the deployment of our secure database join framework. In particular, two or more of the states and ERIC will participate in the MPC protocol. From here we consider two possible solutions. The first option is to maintain the existing repository but now have it secret shared between the computational parties. Alternatively, each state could be the long-term holder of their own data and the states perform all pairwise comparison amongst

themselves. For reason of preferring the more distributed setting we further explore the pairwise comparison approach.

The situation is further complicated by how this data is distributed within and between states. In the typical setting no single state organization has sufficient information to identify individuals which are incorrectly or double registered. For example, typical voter registration forms requires a name, home address and state ID/driver's license number. If two states compared this information there would be no reliable attribute for joining the two records. The name of the voter could be used but names are far from a unique identifier. The solution taken by ERIC is to first perform a join between a state's registered voters and their Department of Motor Vehicles (DMV) records, using the state ID/driver's license number as the join-key. Since the DMV typically possesses an individual's Social Security Number (SSN), this can now be used as a unique identifier across all states. However, due to regulations within some states this join is only allowed to be performed on the hashed data or, presumably, on secret shared data.

In addition to identifying individuals that are double registered, the mission of ERIC is to generally improve the accuracy of all voter records. This includes identifying individuals that have moved and not yet registered in their new state or that have simply moved within a state and not updated their current address. In this case the joins between/within states should also include an indicator denoting that an individual has updated their address at a DMV which is different than the voter registration record. There are likely other scenarios which ERIC also identifies but we leave the exploration of them to future work.

Given the building blocks of Section 3 it is a relatively straightforward task to perform the required joins. First a state performs a left join between their DMV data and the voter registration data. Within this join the addresses in the inner join are compared. In the event of a discrepancy, the date of when these addresses were obtained can be compared to identify the most up to date address. Moreover, the agency with the older address can be notified and initial a procedure to determine which, if any, of the addresses should be updated.

Once this join is performed, each state holds a secret shared table of all their citizens that possess a state ID and their current registration status. Each pair of states can then run an inner join protocol using the social security number as the key. There are several cases that a result record can be in. First it is possible for a person to have a DMV record in two states and be registered in neither. The identity of these persons should not be revealed as this does not effect the voting process. The next case is that a person is registered in both states. We wish to reveal this group to both states so that the appropriate action can be taken. The final case that we are interested in is when a person is registered in state A and has a newer DMV address in state B. In this case we want to reveal the identity of the person to the state that they are registered to. This state can then contact the person to inquire whether they wish to switch their registration to the new state.

This approach has additional advantages over the hashing technique of ERIC. First, all of the highly sensitive information such as a persons address, state ID number and SSN can still be hashed before being added to the database[5]. However, now that the data is secret shared less sensitive information such as dates need not be hashed. This allows for the more expressive query described above which uses a numerical comparison. To achieve the the same functionality using the current ERIC approach these dates would have to be stored in plaintext which leaks significant information. In addition, when the ERIC approach performs these comparison the truth value for each party of the predicate is revealed. Our approach reveals no information about any intermediate value.

```
stateA = select DMV.name,
                DMV.ID,
                DMV.SSN,
                DVM.date > Voter.date ?
                  DMV.date : Voter.date as date,
                DVM.date > Voter.date ?
                  DMV.address : Voter.address as address,
                DVM.address ≠ Voter.address as mixedAddress,
                Voter.name ≠ NULL as registered
           from DMV left join Voter
             on DMV.ID = Voter.ID
stateB = select ...
resultA = select stateA.SSN
                 stateA.address as addressA
                 stateB.address as addressB
                 stateA.registered
                 stateB.registered
           from stateA inner join stateB
             on stateA.SSN = stateB.SSN
          where (stateA.date < stateB.date and stateA.registered)
             or (stateA.registered          and stateB.registered)
resultB = select ...
```

Figure 10: SQL styled join query for the ERIC voter registration application.

Once the parties construct the tables in Figure 10, state A can query the table *stateA* to reveal all IDs and addresses where the *mixedAddress* attribute is set to `true`. This reveals exactly the people who have conflicting addresses between that state's voter and DMV databases. When comparing voter registration data between

[5]The hashing originally performed by ERIC can be replaced with the randomized encoding protocol.

Parameters: 3 parties denoted as P_{p}, P_{s} and P_{r}. Elements are strings in $\Sigma := \{0,1\}^{\sigma}$. An input, output vector size of n, m.

[Permute] Upon the command $(\textsc{Permute}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from P_{p} and $(\textsc{Permute}, \langle\!\langle A \rangle\!\rangle_1)$ from P_{s}. Require that $\pi : [m] \to [n]$ is *injective* and $\langle\!\langle A \rangle\!\rangle_0, \langle\!\langle A \rangle\!\rangle_1 \in \Sigma^n$. Then:

1. P_{p} uniformly samples a bijection $\pi_0 : [n] \to [n]$ and let $\pi_1 : [n] \to [m]$ s.t. $\pi_1 \circ \pi_0 = \pi$. P_{p} sends π_0 and $S \leftarrow \Sigma^n$ to P_{s}.
2. P_{s} sends $B := (\langle\!\langle A_{\pi_0(1)} \rangle\!\rangle_1 \oplus S_1, ..., \langle\!\langle A_{\pi_0(n)} \rangle\!\rangle_1 \oplus S_n)$ to P_{r}.
3. P_{p} sends π_1 and $T \leftarrow \Sigma^m$ to P_{r} who outputs $\langle\!\langle A' \rangle\!\rangle_0 := \{B_{\pi_1(1)} \oplus T_1, ..., B_{\pi_1(m)} \oplus T_m\}$. P_{p} outputs $\langle\!\langle A' \rangle\!\rangle_1 := \{S_{\pi_1(1)} \oplus T_1 \oplus \langle\!\langle A_{\pi(1)} \rangle\!\rangle_0, ..., S_{\pi_1(m)} \oplus T_m \oplus \langle\!\langle A_{\pi(m)} \rangle\!\rangle_0\}$.

Figure 11: The Oblivious Permutation Network protocol $\Pi_{\textsc{Permute}}$ repeated.

Parameters: 3 parties denoted as the P_{p}, P_{s} and P_{r}. Elements are strings in $\Sigma := \{0,1\}^{\sigma}$. An input vector size of n and output size of m.
[Permute] Upon the command $(\textsc{Permute}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from the P_{p} and $(\textsc{Permute}, \langle\!\langle A \rangle\!\rangle_1)$ from the P_{s}:

1. Interpret $\pi : [m] \to [n]$ as an injective function and $A \in \Sigma^n$.
2. Compute $A' \in \Sigma^m$ s.t. $\forall i \in [m], A_{\pi(i)} = A'_i$.
3. Generate $\langle\!\langle A' \rangle\!\rangle$ and send $\langle\!\langle A' \rangle\!\rangle_0$ to P_{p} and $\langle\!\langle A' \rangle\!\rangle_1$ to P_{r}.

Figure 12: The Oblivious Permutation Network ideal functionality $\mathcal{F}_{\textsc{Permute}}$.

states, state B should define *stateB* in a symmetric manner as *stateA*. The table *resultA* contains all of the records which are revealed to state A and *resultB*, which is symmetrically defined, contains the results for state B. We note that *resultA* and *resultB* can be constructed with only one join.

Both types of these queries can easily be performed in our secure framework. All of the conditional logic for the select and where clauses are implemented using a binary circuit immediately after the primary join protocol is performed. This has the effect that overhead of these operation is simply the size of the circuit which implements the logic times the number of potential rows contained in the output.

B Omitted Proofs

B.1 Permutation Network

We now formally prove that the oblivious permutation network protocol in Figure 5 and repeated in Figure 11 is secure with respect to the $\mathcal{F}_{\textsc{Permute}}$ functionality of Figure 12.

Theorem 1. *Protocol* Π_{PERMUTE} *of Figure 11 securely realized the ideal functionality* $\mathcal{F}_{\text{PERMUTE}}$ *of Figure 12 given at most one party is corrupted in the semi-honest model.*

Proof. Correctness follows directly from $\pi_1 \circ \pi_0 = \pi$ and that the masks cancel out. With respect to simulation, consider the following three cases:

1. *Corrupt* P_p: The view of P_p contains no messages and therefore is trivial to simulation.

2. *Corrupt* P_s: The view of P_p contains π_1, S which are sent by P_p. The simulator can uniformly sample $\pi_1 : [m] \to [n]$ from all such injective functions and uniformly sample $S \leftarrow \Sigma^n$. Clearly S has the same distribution.

   With respect to π_1, observe if π_1 if first fixed uniformly at random then there are exactly $(n-m)!$ ways to choose π_0. Moreover, for each choice of π_1 there is a disjoint set of possible π_0 values. Therefore, P_p sampling π_0 uniformly at random results in the distribution of π_1 also being uniform.

3. *Corrupt* P_r: The view of P_r contains $B := (A_{\pi_0(1)} \oplus S_1, ..., A_{\pi_0(n)} \oplus S_n)$ and $\pi_1, T \in \Sigma^m$. π_1, T are sampled uniformly and therefore trivial to simulation. similarly, each $B_i = A_{\pi_0(i)} \oplus S_i$ where S_i is uniformly distributed in their view. Therefore B_i is similarly distributed.

□

B.2 Duplication Network

We now formally prove that the oblivious duplication network protocol in Figure 5 and repeated in Figure 11 is secure with respect to the $\mathcal{F}_{\text{DUP}}$ functionality of Figure 14.

Theorem 2. *Protocol* $\Pi_{\text{DUPLICATE}}$ *of Figure 13 securely realized the ideal functionality* $\mathcal{F}_{\text{DUPLICATE}}$ *of Figure 14 given at most one party is corrupted in the semi-honest model.*

Proof. Correctness follows an inductive argument. It is easy to verify $B_1 = \langle\!\langle A_1 \rangle\!\rangle_1$ and that this is correct since $\pi(1) = 1$ by definition. Inductively let us assume that $B_{i-1} = \langle\!\langle A_{\pi(i-1)} \rangle\!\rangle_1$ and we will show that $B_i = \langle\!\langle A_{\pi(i)} \rangle\!\rangle_1$. Observe that for $i \in [2, n]$

$$\begin{aligned}
\langle\!\langle B_i \rangle\!\rangle_0 &= M_i^{b_i} \oplus W_i^{\rho_i} \oplus b_i \langle\!\langle B_{i-1} \rangle\!\rangle_0 \\
&= \overline{b_i} \langle\!\langle A_i \rangle\!\rangle_0 \oplus b_i \langle\!\langle B_{i-1} \rangle\!\rangle_1 \oplus \langle\!\langle B_i \rangle\!\rangle_1 \oplus W_i^{b_i \oplus \phi_i} \oplus W_i^{\rho_i} \oplus b_i \langle\!\langle B_{i-1} \rangle\!\rangle_0 \\
&= \overline{b_i} \langle\!\langle A_i \rangle\!\rangle_0 \oplus b_i B_{i-1} \oplus \langle\!\langle B_i \rangle\!\rangle_1
\end{aligned}$$

And therefore $B = \pi(\langle\!\langle A \rangle\!\rangle_1)$ and

Parameters: 3 parties denoted as P_{p}, P_{s} and P_{r}. Elements are strings in $\Sigma := \{0,1\}^\sigma$. An input, output vector size of n.

[Duplicate] Upon the command $(\textsc{Duplicate}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from P_{p} and $(\textsc{Duplicate}, \langle\!\langle A \rangle\!\rangle_1)$ from P_{s}. Require that $\pi : [n] \to [n]$ s.t $\pi(1) = 1$ and $\pi(i) \in \{i, \pi(i-1)\}$ for $i \in [2,n]$ and $A \in \Sigma^n$. Then:

1. P_{p} computes the vector $b \in \{0,1\}^m$ such that $b_1 = 0$ and for $i \in [2,n]$, $b_i = 1$ if $\pi(i) = \pi(i-1)$ and 0 otherwise.
2. P_{s} samples $\langle\!\langle B \rangle\!\rangle_1, W^0, W^1 \leftarrow \Sigma^n, \langle\!\langle B_1 \rangle\!\rangle_0 \leftarrow \Sigma$ and $\phi \leftarrow \{0,1\}^n$. P_{s} redefine $\langle\!\langle B_1 \rangle\!\rangle_1 := \langle\!\langle A_1 \rangle\!\rangle_1 \oplus \langle\!\langle B_1 \rangle\!\rangle_0$. For $i \in [2,n]$, P_{s} sends
$$M_i^0 := \langle\!\langle A_i \rangle\!\rangle_1 \quad \oplus \langle\!\langle B_i \rangle\!\rangle_1 \oplus W_i^{\phi_i}$$
$$M_i^1 := \langle\!\langle B_{i-1} \rangle\!\rangle_1 \oplus \langle\!\langle B_i \rangle\!\rangle_1 \oplus W_i^{\overline{\phi_i}}$$
and $\langle\!\langle B_1 \rangle\!\rangle_0, \phi$ to P_{p}. P_{s} sends $\langle\!\langle B \rangle\!\rangle_1, W^0, W^1$ to P_{r}.
3. P_{p} sends $\rho := \phi \oplus b, R \leftarrow \Sigma^n$ to P_{r} who responds with $\{W_i^{\rho_i} : i \in [2,n]\}$. For $i \in [2,n]$, P_{p} defines
$$\langle\!\langle B_i \rangle\!\rangle_0 := M_i^{b_i} \oplus W_i^{\rho_i} \oplus b_i \langle\!\langle B_{i-1} \rangle\!\rangle_0$$
P_{p} outputs $\langle\!\langle A' \rangle\!\rangle_0 := \langle\!\langle B \rangle\!\rangle_0 \oplus R \oplus \pi(\langle\!\langle A \rangle\!\rangle_0)$ and P_{r} outputs $\langle\!\langle A' \rangle\!\rangle_1 := \langle\!\langle B \rangle\!\rangle_1 \oplus R$.

Figure 13: The Oblivious Duplication Network protocol $\Pi_{\textsc{duplicate}}$ repeated.

Parameters: 3 parties denoted as the P_{p}, P_{s} and P_{r}. Elements are strings in $\Sigma := \{0,1\}^\sigma$. An input vector size of n and output size of n.

[Duplicate] Upon the command $(\textsc{Duplicate}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from the P_{p} and $(\textsc{Duplicate}, \langle\!\langle A \rangle\!\rangle_1)$ from the P_{s}:

1. Interpret $\pi : [n] \to [n]$ as a function s.t. $\pi(1) = 1, \pi(i) \in \{i, \pi(i-1)\}$ for $i \in [2,n]$ and $A \in \Sigma^n$.
2. Compute $A' \in \Sigma^m$ s.t. $\forall i \in [n], A_{\pi(i)} = A'_i$.
3. Generate $\langle\!\langle A' \rangle\!\rangle$ and send $\langle\!\langle A' \rangle\!\rangle_0$ to P_{p} and $\langle\!\langle A' \rangle\!\rangle_1$ to P_{r}.

Figure 14: The Oblivious Duplication Network ideal functionality $\mathcal{F}_{\textsc{duplicate}}$.

$$\begin{aligned}A' =& \langle\!\langle B \rangle\!\rangle_1 \oplus R \oplus \pi(\langle\!\langle A \rangle\!\rangle_0) \oplus \langle\!\langle B \rangle\!\rangle_1 \oplus R \\ =& B \oplus \pi(\langle\!\langle A \rangle\!\rangle_0) \\ =& \pi(\langle\!\langle A \rangle\!\rangle_1) \oplus \pi(\langle\!\langle A \rangle\!\rangle_0) \\ =& \pi(A)\end{aligned}$$

With respect to simulation, consider the following three cases:

1. *Corrupt* P_{p}: The transcript of P_{p} contains $M^0, M^1 \in \Sigma^n, \langle\!\langle B_1 \rangle\!\rangle_0 \in \Sigma, \phi \in \{0,1\}^n$ from P_{s} and $W_i^{b_i \oplus \phi_i}$ from P_{r}. First observe that $\langle\!\langle B_1 \rangle\!\rangle_0, \phi$ are sampled uniformly and therefore can be simulated as the same.

   Next recall that

   $$\begin{aligned}M_i^{b_i} =& \ldots \oplus \langle\!\langle B_i \rangle\!\rangle_1 \\ M_i^{\overline{b_i}} =& \ldots \oplus W_i^{\overline{b_i \oplus \phi_i}}\end{aligned}$$

   where $\langle\!\langle B_i \rangle\!\rangle_1, W_i^{\overline{b_i \oplus \phi_i}} \in \Sigma$ are sampled uniformly can not in the view of P_{p}. Therefore M_i^0, M_i^1 are distributed uniformly.

2. *Corrupt* P_{s}: The transcript of P_{s} contains nothing and therefore is trivial to simulate. Note that the distribution of the output shares in independent of P_{s}'s random tape (view) due to $P_{\mathsf{p}}, P_{\mathsf{r}}$ re-randomizing the shares with $R \leftarrow \Sigma^n$.

3. *Corrupt* P_{r}: The transcript of P_{r} contains $\langle\!\langle B_1 \rangle\!\rangle_1, W^0, W^1$ from P_{s} and ρ from P_{p}. W^0, W^1 are sampled uniformly and therefore can be simulated as the same. $\langle\!\langle B_1 \rangle\!\rangle_1 = A_1 \oplus \langle\!\langle B_1 \rangle\!\rangle_0$ where $\langle\!\langle B_1 \rangle\!\rangle_0$ is sampled uniformly and not in the view. Therefore $\langle\!\langle B_1 \rangle\!\rangle_1$ is distributed uniformly. The same applies to ρ since ϕ is uniform and not in the view.

□

B.3 Switching Network

We now formally prove that the oblivious switching network protocol in Figure 5 and repeated in Figure 15 is secure with respect to the $\mathcal{F}_{\text{SWITCH}}$ functionality of Figure 16. In the proof we will replace calls to the Permutaiton and Duplication protocols of Π_{SWITCH} with their ideal functionalities (Figure 12, 14).

Theorem 3. *Protocol* Π_{SWITCH} *of Figure 15 securely realized the ideal functionality* $\mathcal{F}_{\text{SWITCH}}$ *of Figure 16 given at most one party is corrupted in the semi-honest model.*

Parameters: 3 parties denoted as P_{p}, P_{s} and P_{r}. Elements are strings in $\Sigma := \{0,1\}^{\sigma}$. An input, output vector size of n, m.

[Switch] Upon the command $(\textsc{Switch}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from P_{p} and $(\textsc{Switch}, \langle\!\langle A \rangle\!\rangle_1)$ from P_{s} where $\pi : [m] \to [n]$ and $\langle\!\langle A \rangle\!\rangle_0, \langle\!\langle A \rangle\!\rangle_1 \in \Sigma^n$.

1. P_{p} samples an injection $\pi_1 : [m] \to [n]$ s.t. for $i \in image(\pi)$ and $k = |preimage(\pi, i)|$, $\exists j$ where $\pi_1(j) = i$ and $\{\pi_1(j+1), ..., \pi_1(j+k)\} \cap image(\pi) = \emptyset$. P_{p} sends $(\textsc{Permute}, \pi_1, \langle\!\langle A \rangle\!\rangle_0)$ to $\mathcal{F}_{\textsc{Permute}}$ and P_{s} sends $(\textsc{Permute}, \langle\!\langle A \rangle\!\rangle_1)$. P_{p} receives $\langle\!\langle B \rangle\!\rangle_0 \in \Sigma^m$ in response and P_{r} receives $\langle\!\langle B \rangle\!\rangle_1 \in \Sigma^m$.
2. P_{p} defines $\pi_2 : [m] \to [m]$ s.t. for $i \in image(\pi)$ and $k := |preimage(\pi, i)|$ and j where $\pi_1(j) = i$, then $\pi_2(j) = ... = \pi_2(j+k) = j$. P_{p} and P_{r} respectively send $(\textsc{Duplicate}, \pi_2, \langle\!\langle B \rangle\!\rangle_0)$ and $(\textsc{Duplicate}, \langle\!\langle B \rangle\!\rangle_1)$ to $\mathcal{F}_{\textsc{Duplicate}}$. As a result P_{p} obtains $\langle\!\langle C \rangle\!\rangle_0 \in \Sigma^m$ from $\mathcal{F}_{\textsc{Duplicate}}$ and P_{s} obtains $\langle\!\langle C \rangle\!\rangle_1 \in \Sigma^m$.
3. P_{p} computes the permutation $\pi_3 : [m] \to [m]$ such that for $i \in image(\pi)$ and $k = |preimage(\pi, i)|$, $\{\pi_3(\ell) : \ell \in preimage(\pi, i)\} = \{j, ..., j+k\}$ where $i = \pi_1(j)$. P_{p} sends $(\textsc{Permute}, \pi_3, \langle\!\langle C \rangle\!\rangle_0)$ to $\mathcal{F}_{\textsc{Permute}}$ and P_{s} sends $(\textsc{Permute}, \langle\!\langle C \rangle\!\rangle_1)$. P_{p} receives $S \in \Sigma^m$ in response. P_{p} and P_{r} respectively receives and outputs $\langle\!\langle A' \rangle\!\rangle_0, \langle\!\langle A' \rangle\!\rangle_1 \in \Sigma^m$.

Figure 15: The Oblivious Switching Network protocol $\Pi_{\textsc{switch}}$ repeated.

Parameters: 3 parties denoted as the P_{p}, P_{s} and P_{r}. Elements are strings in $\Sigma := \{0,1\}^{\sigma}$. An input vector size of n and output size of m.
[Switch] Upon the command $(\textsc{switch}, \pi, \langle\!\langle A \rangle\!\rangle_0)$ from the P_{p} and $(\textsc{switch}, \langle\!\langle A \rangle\!\rangle_1)$ from the P_{s}:

1. Interpret $\pi : [m] \to [n]$ and $A \in \Sigma^n$.
2. Compute $A' \in \Sigma^m$ s.t. $\forall i \in [m], A_{\pi(i)} = A'_i$.
3. Generate $\langle\!\langle A' \rangle\!\rangle$ and send $\langle\!\langle A' \rangle\!\rangle_0$ to P_{p} and $\langle\!\langle A' \rangle\!\rangle_1$ to P_{r}.

Figure 16: The Oblivious Switching Network ideal functionality $\mathcal{F}_{\textsc{switch}}$ repeated.

Proof. Correctness follows from the first oblivious permutation call rearranges the input vector such that each output item which appears k times is followed by $k-1$ items which do not appear in the output. The duplication network then copies each of these output items into the next $k-1$ position. The final permutation places these items in the final order.

With respect to simulation, the transcript of each party contains their transcripts of three subprotocols: Permute, Shared-Duplicate and Shared-Permute. By Theorem 1 the Permute subprotocol transcript can be simulated. Similarly, Theorem 1,2 also imply that the other two transcripts can be simulated. Therefore this implies that the overall protocol can be simulated given that no other communication is performed.

□

B.4 Join Protocol

Theorem 4. *Protocol* Π_{JOIN} *of Figure 6 securely realized the ideal functionality* $\mathcal{F}_{\text{JOIN}}$ *of Figure 7 given at most one party is corrupted in the semi-honest* $\mathcal{F}_{\text{PERMUTE}}$,$\mathcal{F}_{\text{SWITCH}}$,$\mathcal{F}_{\text{ENCODE}}$*-hybrid model with statistical security parameters* λ.

Proof. First we demonstrate the correctness of the protocol. Recall that the set of non-join keys $\{(X_{J_1}||...||X_{J_l})[i] \mid i \in [n]\}$ are all distinct. The same holds true for the Y table. As such, P_0 receives n uniformly random values from $\mathcal{F}_{\text{ENCODE}}$. As discussed in 3.2, given that these encodings are of length at least $\lambda + 2\log_2(n)$ bits, then with probability $1 - 2^{-\lambda}$ all the encodings are distinct.

Recall that P_1 then constructs a cuckoo hash table using the encodings $\mathbb{E}_y$. Given that cuckoo hash table is parameterized as described in [DRRT18], this succeeds with overwhelming probability, i.e. $1 - 2^{-\lambda}$.

The correctness of the rest of the protocol is straight forward. The shared table $[\![Y]\!]$ are permuted to form a shared cuckoo hash table $[\![T]\!]$. Based on the encodings $\mathbb{E}_x$, the shares in the table T are mapped to the corresponding row of X. It is easy to verify that if Y has a matching row then it will have been mapped. Finally, $\mathcal{F}_{\text{MPC}}$ is used to compute the circuit which constructs the output table.

With respect to simulation, consider the following cases:

1. *Corrupt* P_0: The transcript of P_0 contains the encodings $\mathbb{E}_x$, the output $\langle\!\langle \widehat{Y}^l \rangle\!\rangle_0$ from $\mathcal{F}_{\text{SWITCH}}$, and the output of $\mathcal{F}_{\text{MPC}}$. Given that the inputs to $\mathcal{F}_{\text{ENCODE}}$ are either set to or all distinct values, the output $\mathbb{E}_x$ is uniformly distributed and therefore can be sample as such by the simulator. Similarly, the output of $\mathcal{F}_{\text{SWITCH}}, \mathcal{F}_{\text{MPC}}$ are both uniform.

2. *Corrupt* P_1: The transcript of P_1 contains the encodings $\mathbb{E}_y$, the output $\langle\!\langle \widehat{T} \rangle\!\rangle_1$ from $\mathcal{F}_{\text{PERMUTE}}$, the output $\langle\!\langle \widehat{Y}^l \rangle\!\rangle_1$ from $\mathcal{F}_{\text{SWITCH}}$, and the output of $\mathcal{F}_{\text{MPC}}$. All of these are distributed uniformly. The simulation of this transcript follows the same as that of P_0.

3. *Corrupt* P_2: The transcript of P_2 contains the output $\langle\!\langle \widehat{T} \rangle\!\rangle_0$ from $\mathcal{F}_{\text{PERMUTE}}$, the output $\langle\!\langle \widehat{Y}^l \rangle\!\rangle_0$ from $\mathcal{F}_{\text{SWITCH}}$, and the output of $\mathcal{F}_{\text{MPC}}$. All of these are distributed uniformly. The simulation of this transcript follows the same as that of P_0.

□