SFour: A Protocol for Cryptographically Secure Record Linkage at Scale

Basit Khurram
University of Waterloo
basit.khurram@uwaterloo.ca

Florian Kerschbaum
University of Waterloo
fkerschb@uwaterloo.ca

***Abstract*—The prevalence of various (and increasingly large) datasets presents the challenging problem of discovering common entities dispersed across disparate datasets. Solutions to the private record linkage problem (PRL) aim to enable such explorations of datasets in a secure manner. A two-party PRL protocol allows two parties to determine for which entities they each possess a record (either an exact matching record or a fuzzy matching record) in their respective datasets — without revealing to one another information about any entities for which they do not both possess records. Although several solutions have been proposed to solve the PRL problem, no current solution offers a fully cryptographic security guarantee while maintaining both high accuracy of output and subquadratic runtime efficiency. To this end, we propose the first known efficient PRL protocol that runs in subquadratic time, provides high accuracy, and guarantees cryptographic security in the semi-honest security model.**

***Index Terms*—Secure Multiparty Computation, Private Record Linkage, Entity Matching, Private Permutation, Deduplication**

I. Introduction

With greater computation prowesses come greater abundances of data with which come the natural responsibilities of protecting such data stores. However, such protections need not be entirely suffocating: it is fully feasible to protect privacy of data while still allowing the data to be utilized in meaningful ways. Yet, it is common to see organizations keeping their data in silos instead of collaborating with one another in ways that could stand to benefit all parties involved, including the individuals whose data the organizations possess. Consider the case of two hospitals [30, 33, 35]. Consider now the potential advantages of allowing the two hospitals to aggregate their data on patients that frequent both hospitals. Such a collaboration could result in not only richer datasets that reveal quicker treatment options, but could also prevent the misallocation of resources from having the two hospitals duplicating each other's findings. Understandably, privacy laws often prevent hospitals from sharing such data with others, even if doing so could be highly advantageous for all those that are involved. The field of secure multiparty computation [2, 6, 42] (SMC) aims to address this misallocation of resources. Specifically, within this field is the problem of private record linkage [38] (PRL): the problem of finding all records in one database that also belong in another database, without revealing any information about any other record in the two databases. In the example of our two hospitals, rather than having each hospital duplicate the other's work, the two hospitals could run a private record linkage protocol to aggregate data for the patients that they have in common. Now, the two hospitals could continue their efforts in helping the patients that they were already helping, without learning anything about patients that they were not already helping.

There are different solutions to the private record linkage problem [11, 12, 16, 19, 34], but there is currently no solution that is accepted as satisfactorily solving the PRL problem in its entirety. Each solution comes with a tradeoff in its accuracy, computational complexity, or security. There is a trivial solution, the *All Pairwise Comparison* approach, which guarantees perfect accuracy and complete security but it runs in quadratic time. A quadratic runtime complexity can easily become unscalable when dealing with datasets containing billions of records. Further, there are datasets where matching records are rare, for example, the North American Association of Central Cancer Registries maintains a standard for ensuring cancer registries contain fewer than 0.1% duplicated records [17]. Intuitively speaking, given the rarity of matches, a protocol that runs in quadratic time is somewhat excessive, and so, a good solution to the private record linkage problem must run in subquadratic time. Current solutions that run in subquadratic time tend to either lack security or provide inaccurate results [38].

He et al. recently proposed a protocol with a subquadratic runtime and high accuracy [16]. In terms of security, the authors note that they are not aware of any subquadratic protocol with high accuracy that also guarantees cryptographic security — and so, the authors consider the use of differential privacy to propose the DPRL security model for the record linkage problem, instead of exploring the possibility of a cryptographically secure solution. While differential privacy limits the leakage of information, this leakage is still significantly large when compared to the protections offered by cryptographic security. Furthermore, while the protocol proposed by He et al. runs in subquadratic time, it is nevertheless limited by the cost of expensive cryptographic computations, resulting in a runtime of around 80 hours for a pair of datasets with each having only $5,000$ records.

To this end, we propose a new protocol, SFour, which aims to address the shortfalls in existing protocols. Our contributions are summarized as follows:

- we create a PRL protocol which 1) runs in $O(n \operatorname{polylog}(n))$ time, 2) provides guaranteed cryptographic security in the semi-honest security model, and 3) has high recall and perfect precision;

- we evaluate our protocol's runtime on real and synthetic data to show that we outperform the differentially private Laplace Protocol [16] by 1-2 orders of magnitude;
- we develop a new efficient private permutation technique that can greatly improve upon other schemes that use a similar approach as us, such as the Sort-Compare-Shuffle protocol [18] for private set intersection.

The rest of this paper is divided so that we provide the necessary background knowledge in Section II, some related work in Section III, the details of our protocol in Section IV, a display of our experimental results in Section V, and lastly, we conclude our paper in Section VI.

II. Problem Definition

In this section we explain the threat model in which our protocol operates and the formalization of the problem that our protocol solves, the private record linkage (PRL) problem.

A. Threat model

This work considers only the *semi-honest* security model, where an adversarial party attempts to gain as much information about another party's inputs as possible, while using a transcript of the entire protocol, but without deviating from the protocol's instructions. A two-party protocol is secure in the semi-honest model when neither party is able to gain any information from the execution of the protocol, other than the information gained from the protocol's output and the size of the other party's input. More formally, a two-party protocol Π computing $f(x,y)$ is secure in the semi-honest security model when each party's view $VIEW^{\Pi}(x,y)$, where the party's input is x and the other party's input is y, is computationally indistinguishable from the view generated by the party while using a polynomial-time simulator to simulate messages received from the other party (after supplying to the simulator the party's own input x and its output from computing $f(x,y)$) [27].

The semi-honest security model is contrasted to the *malicious security* model, the latter allowing adversaries to arbitrarily deviate from the specified protocol while attempting to nonconsensually gain information from the protocol's execution [27]. There are cases where a protocol that is secure in the semi-honest model is sufficient to meet the needs of protocol participants, such as when meeting legal requirements related to the sharing of private data [13], or when making use of other software attestation methods that can detect deviations from a specified protocol [29]. While our work currently does not address the malicious security model, an efficient protocol in the semi-honest model is an important prerequisite towards constructing efficient protocols that are secure in the malicious security model. When we describe our protocol in Section IV, we prove that the SFour protocol is secure in the semi-honest model.

Goldreich et al. [15] show that any protocol that is secure in the semi-honest security model can be made secure in the malicious security model, through the use of what are known as zero-knowledge proofs. We emphasize that simply applying generic constructions for zero-knowledge proofs would destroy the performance gain of our protocol: optimized zero-knowledge proofs need to be constructed, e.g., set completeness checks for our free shuffling technique, and white-box conversions between our secure computation protocols. We delegate this (efficient) adaptation of our protocol for the malicious security model to future work.

B. The private record linkage problem

Consider two semi-honest parties P_1 and P_2, which own database D_1 and database D_2, respectively, wanting to discover all entities for which they each have a corresponding record. Before proceeding however, we require that both parties first *deduplicate* their records, i.e., ensure that no two records from any one database D_i match one another. Now, let D_1 and D_2 each come from some domain $\mathcal{D}$ and let each database consist of records from some domain $\mathcal{R}$. Suppose further that the parties agree in advance on a collection of required columns that each database must contain. Having columns in common helps the parties determine whether a record in one database matches a record in the other database. To designate any two records as matching records, the two parties additionally agree on a *matching function* $m : \mathcal{R} \times \mathcal{R} \rightarrow \{0,1\}$ which when given two records outputs a 1 if the given records are a match for each other, or a 0 otherwise. Within our setting, the matching function is a commutative function, i.e., $m(r,s) = m(s,r), \forall (r,s) \in \mathcal{R}^2$. Having agreed on a matching function, the parties now need a protocol to solve the private record linkage problem.

Formally, the semi-honest parties P_1 and P_2 need a protocol $\Pi : \mathcal{D} \times \mathcal{D} \rightarrow \mathcal{D}^2$, such that $\Pi(D_1, D_2) = (O_1, O_2)$, where $O_i = \{r | r \in D_i \wedge (\exists s \in D_{3-i} | m(r,s) = 1)\}$. Apart from this output, the only other information the parties are allowed to learn is the size of the other party's database. The aim of this work is to describe a PRL protocol Π that 1) runs in subquadratic time, 2) offers guaranteed cryptographic security in the semi-honest security model, and, 3) achieves high accuracy.

III. Related Work

Several avenues of research have been pursued in finding solutions to the PRL problem [11, 12, 16, 19, 34], each with varying tradeoffs. The trivial solution to PRL is the *All Pairwise Comparison* (APC) approach which uses secure computations to compare every single record in one dataset with every single record in the other dataset [16]. While this approach has guaranteed security and perfect accuracy, it also has a quadratic runtime complexity which is untenable at scale. Indeed, He et al. recently used a moderately powerful consumer machine [1] to estimate that it would take three weeks to perform APC across two databases, with each database having only 5,000 records [16]. With APC as a benchmark,

[1] A 3.1 GHz Intel Core i7 machine with 16 GB RAM.

other solutions often sacrifice security or accuracy for better asymptotic runtimes. Some approaches attempt to run in linear time by performing exact matches on records or by using protocols for private set intersection, however these approaches suffer from low accuracy [16]. The SCS protocol proposed by Huang et al. [18] is not a PRL protocol but rather is a private set intersection protocol, however, our work in some sense generalizes this approach and adapts it to the PRL problem. Further, our work offers contributions that may significantly improve the SCS protocol's runtime performance.

Some other PRL approaches, such as those that use Bloom filters to try to speed up their computations [11, 12, 34], offer no formal security guarantee and some of these indeed get broken soon after publishment [7, 8, 9, 23]. Wen and Dong propose the concept of garbled Bloom filters,

Some authors attempt to prove the security of their protocols by (unsuccessfully) attacking their protocols themselves [9, 39]. We stress that cryptographic security offers a much more satisfying guarantee of security. Perhaps most worrying is that studies investigating the practicality of using private record linkage on real world data tend to suggest that these protocols with no formal security guarantees may be ready for use in hospitals and other national databases [30, 33, 35].

Authors of previous works [16, 19, 31] note that they know of no subquadratic PRL protocol with high accuracy and cryptographic security. Inan et al. settle for producing one of the first subquadratic protocols that leaks only differentially private noise [19]. More recently, He et al. [16] however, show that this earlier work by Inan et al. overlooks some information leakage and that the protocol is insecure. He et al. go on to produce a corrected subquadratic PRL protocol with high accuracy that leaks only differentially private noise. Rao et al.'s protocols provides the best performance [31] in the differentially private setting, but additionally relies on a semi-trusted third party. However, our strictly two-party protocol still outperforms theirs by an order of magnitude. All these works state that no cryptographically secure PRL protocol that runs in subquadratic time and with high accuracy has yet been formalized. It is our intention to change this view with the work we present in this paper. For further details and comparisons of other PRL protocols, we refer the reader to the PRL taxonomy [38].

IV. SFour Protocol

Our protocol is conceptually split into four phases, however we add a "zero-th step" to aid in our exposition. Briefly, we aim to score records in such a way that if two records are similar to one another, then each of these two records has a score similar to the other record's score. Once both parties have scored their records, they can perform a secure computation to sort the union of their scored records and then compare only those records that are close to each other, as determined by their scores. The parties run a sliding window over the sorted records and compare the records in a window with only other records in the same window. This reduces the total number of secure comparisons needed in the protocol, as parameterized by the window size.

The order in which the comparison results are revealed could leak positional information about the records (Section IV-E1), and so each party performs a private permutation on the results before the results are revealed. In order to optimize the permutation, both parties generate identifiers, with each identifier carrying with it a semantic meaning known only to the party that generated the identifier. At the end of the protocol, each party possesses a collection of identifiers that it generated. These identifiers convey to each party which records possess a matching record in the other party's dataset. Figure 1 summarizes our overall protocol.

In this section we describe the steps of our protocol in further detail.

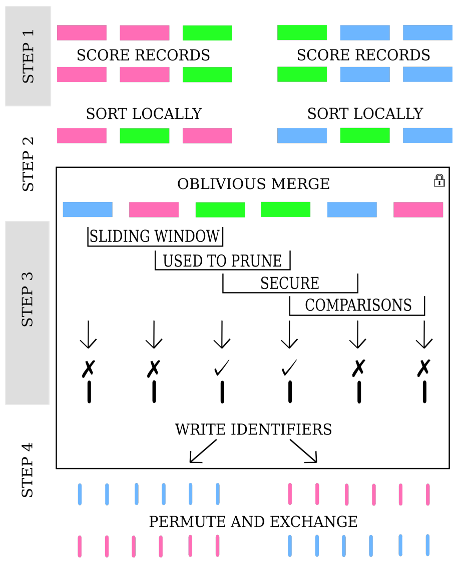


Fig. 1: Overview of the SFour protocol. The locked box indicates use of secure multiparty computations. The small bars at the bottom of the figure indicate identifiers. We colour the identifiers with two different colours to reflect the different parties that generated the identifiers. At the end of Step 4, each party receives a set of identifiers that that party generated. Each party recognizes the semantic meanings of only the identifiers that it generated.

A. Step 0: Setting parameters and identifiers

Before beginning with the protocol, both parties agree on the following:

- the total number of records in the protocol;
- the scoring function to score their records (Section IV-B);
- the matching function to determine matches (Section IV-D2).

1) Identifier tags (part one): In order to optimize the final step of the protocol, both parties privately generate pseudo-identifiers for the records. This optimization technique is a new way to efficiently perform private permutations and is one of our contributions in this work. Given the subtle nuances involved in the optimization, we split the discussion of these identifiers into three parts, the first of which is here, the second is in Section IV-C1, and the third is in Section IV-E. For now, it suffices to understand that each party generates and appends to each of its own records a private metadata tag in the form

```
(ownerBit||matchID||mismatchID||otherID)
```

where

- `ownerBit` is a bit indicating which of the two parties owns the record (so, Party 1 could set a 1, Party 2 a 0);
- `matchID` is a random integer which is presented to the owner of this record if it is determined that this record has a matching record in the other party's database;
- `mismatchID` is a random integer which is presented to the owner of this record if it is determined that this record does not have a matching record in the other party's database;
- `otherID` is a random integer which is presented to the party that *does not* own this record and offers no indication on as to whether this record has a matching record in either database.

At the end of the protocol each party receives one of the three identifiers tagged to a record for each record involved in the protocol (explained further in Section IV-E). To ensure correctness, however, there are some caveats to how the identifiers are generated. If $ID_{i_1}, ID_{i_2}, ID_{i_3}$ are the sets of `matchID`, `mismatchID`, `otherID` identifiers that are generated by party P_i, respectively, D_i is the set of records belonging to party P_i, and there are n records in total across both parties' datasets, then:

1) $|ID_{i_1}| = |ID_{i_2}| = |D_i|$, i.e., there is a unique `matchID` identifier and a unique `mismatchID` identifier for each record in a party's own database;
2) $|ID_{i_3}| = n - |D_i|$, i.e., P_i has as many unique `otherID` identifiers as there are records in the other party's database;
3) $ID_{i_1}, ID_{i_2}, ID_{i_3}$ are pairwise disjoint, i.e., a party has no collisions between any two of the identifiers that it generates.

There are no restrictions on identifiers generated by one party colliding with identifiers generated by the other party, and so, generating identifiers that adhere to the above restrictions is not difficult. Party P_i can simply take a random permutation of the integers from 1 to $3n$, and select the first $|D_i|$ integers as `matchID` identifiers, the next $|D_i|$ integers as `mismatchID` identifiers, and the next $n - |D_i|$ integers as `otherID` identifiers.

Note that the number of `otherID` identifiers generated by a party is not necessarily the same as the number of either of the other two kinds of identifiers generated by the same party. The number of `otherID` identifiers generated by a party aligns with the number of records held by the other party because our protocol requires the parties to exchange their `otherID` identifiers with one another. However, all identifiers must remain private, including the `otherID` identifiers. This private exchange is accomplished through secret sharing. It is these exchanged `otherID` identifiers that are used in the tags that are appended to the records. While we could add an extra step to our overall protocol to specifically facilitate this exchange in `otherID` identifiers, the work involved in generating and exchanging the identifiers is relatively small. Rather, we add this private exchange of `otherID` identifiers in Algorithm 1 (Section IV-C), which we use when constructing our first SMC circuit.

The exact usage of the different kinds of identifiers is better explained with a little additional context, therefore, we leave the discussion of the precise purposes served by each kind of identifier for Section IV-E.

B. Step 1: Scoring

We wish to compute a score for each record so that when two records are similar to one another they each have a score that is similar to the other record's score. This is the essence of a locality preserving hash function (LPH) [43], a hash function H that satisfies

$$d(x, y) < d(y, z) \Rightarrow d(H(x), H(y)) < d(H(y), H(z)),$$

for some inputs $x, y,$ and z, and some metric d.

After both parties have agreed on a scoring function, each participant is able to locally score its records. By keeping these scores private, there is no risk of information leakage from the first step of our protocol.

1) Space filling curves: While finding a *perfect* scoring function, i.e., a locality preserving hash function that also satisfies the converse of the previous conditional statement, is orthogonal to our research, we ran empirical experiments to find a *satisfactory* scoring function.

A space filling curve (SFC) is a mapping that can transform a multidimensional vector into a one dimensional vector [41]. Using SFCs to project a high dimensional vector into a single dimension has proven to be beneficial in load balancing tasks [26], performing cryptographic transformations [22], processing spatial data [3], and carrying out multidimensional similarity searches [24]. Incidentally, we used a space filling curve specifically for these last two use cases. The SFC we used is called the *Hilbert curve*, and for our work it suffices to know that the Hilbert curve can 1) take an arbitrary n-dimensional vector and project it as an integer in $\mathbb{Z}_{2^{np}}$, where p is a parameter that determines the size of the curve's range, and 2) that after applying the Hilbert curve, points that are close to each other in such an n-dimensional space are projected closely to one another in $\mathbb{Z}_{2^{np}}$ (however, the converse is not always true).

C. Step 2: Sorting

Having scored their records, both parties can blindly sort the union of their scored records by using an SMC circuit. This sorting brings similar records, i.e., records that are likely to match one another, close together. We optimize the secure sorting by offloading some of the computational effort into a quicker offline preprocessing stage.

1) Identifier tags (part two): Before we sort the records however, we recall the pseudo-identifiers mentioned in Section IV-A1 and the discussion of how there is a private exchange of some identifiers. Specifically, only the `otherID` identifiers are exchanged between the two parties, as each party generates `otherID` identifiers for the other party, which is in contrast to the `matchID` identifiers and `mismatchID` identifiers, which each party generates for its own records. After the records have been securely sorted, the parties do not know which record belongs to which party and so the parties are unable to append their identifier tags to their own records. Therefore, the exchange of identifiers takes place before the joint and secure sorting occurs in Step 2, so that the parties can form their identifier tags in advance of sorting and be able to append the tags to their own records. Performing the oblivious exchange of `otherID` identifiers is as simple as each party creating secret shares for the `otherID` identifiers and sending them to the other party. Once the parties receive the shares of the `otherID` identifiers, they can each concatenate the shares to their individual partial record tags (containing only the `ownerBit`, `matchID` identifiers, and `mismatchID` identifiers) to complete their individual record tags. As we will see in Section IV-E3, it does not matter to which of its records a party appends which `otherID` identifier. In Algorithm 1, `partiallyTag` refers to a party's process of locally generating and then concatenating the `ownerBit`, and the `matchID` and `mismatchID` identifiers to its own individual records. After the records are partially tagged, the parties secret share their records and proceed. Similar to the partial tagging process, `completelyTag` refers to the process of a party generating and exchanging shares of `otherID` identifiers, and then concatenating the shares of `otherID` identifiers received from the other party to shares of its own records, and thus completing the record tags.

2) Local sorting (preprocessing): Our oblivious sorting protocol requires a preprocessing step: the parties must first locally sort their own records, using the scores generated in Section IV-B. This simple preprocessing significantly improves our protocol's performance, as we explain next.

3) Secure merge sorting: Our protocol uses a merge network, a circuit that can merge two sorted lists of input, to securely sort the union of both parties' records. Batcher's Odd-Even merge network is currently one of the fastest known merge networks, running with a time complexity of $O(n \log(n))$ [4]. The Odd-Even merge makes use of a lot of comparison operations, and since boolean circuits are significantly faster than arithmetic circuits at performing comparisons, we use a boolean circuit to perform the merge. With this circuit the two parties can securely merge their sorted records, and effectively sort the union of their records. Of course, the parties do not possess the sorted records themselves, but actually possess secret shares of the sorted records.

Algorithm 1 summarizes the computations involved in this subsection. Here, ODDEVENMERGE is the invocation of a secure multiparty computation protocol, with the inputs as two lists of secret shares of the records belonging to the two parties. Code that is not in a secure computation is executed locally by each party. Note that throughout the overall SFour protocol, both parties possess and run the same code; the presence of `if-else` statements allows the parties to play their respective roles as required.

Algorithm 1: Step2(partyId, records)

```
records ← partiallyTag(records)
if partyId == 1 then
    p1RecordShares ← createAndSend(records)
    p2RecordShares ← receive()
else
    p1RecordShares ← receive()
    p2RecordShares ← createAndSend(records)
end
p1RecordShares ← completelyTag(p1RecordShares)
p2RecordShares ← completelyTag(p2RecordShares)
sortedRecordShares ← ODDEVENMERGE(p1RecordShares,
  p2RecordShares)
return sortedRecordShares
```

D. Step 3: Sliding window

Having blindly merge sorted their records, so that similar records are close together, the two parties are ready to finally compare their records to search for matches. We split this section into subsections to first give an overview of how records are selected for comparison, we then provide two further subsections that explain our use of an arithmetic circuit and then a boolean circuit to optimize the comparison of records. We include one last subsection in this step of our protocol to explain how the results of the comparisons are stored.

1) Sliding window: To decide which records should be compared with one another, the parties run a *sliding window* over their sorted records. A window is just a pair of indices, a *left index* and a *right index*, which indicate the range of records currently inside it. A sliding window increments the indices of its bounds so that the window *slides* across all records, while ensuring that the window size remains constant (except for near the end, where the window may reduce in size as the number of records remaining decreases). Our protocol uses a sliding window to determine which records should be compared with one another: all records within the bounds of the window are compared with one another. The sliding window allows the parties to narrow the search space for the records that they compare while seeking matches. This step in its entirety is the most computationally expensive step in the protocol. We can optimize this step by carefully switching between boolean and arithmetic circuits to try to utilize each circuit for the types of operations each type of circuit works best. Incidentally, we switch the boolean shares of the sorted records from Step 2 into arithmetic shares, since it turns out that we will benefit from using an arithmetic circuit soon (Section IV-D4).

An overview of Step 3 is provided in Algorithm 2. Here, ISMATCH is an invocation of an SMC protocol with two inputs, each a secret share of a record being checked for a match with the other record.

Algorithm 2: Step3(sortedRecordShares)

```
/* Convert the boolean shares to arithmetic
   shares.                                          */
for idx ← 0 to sortedRecordShares.size()−1 do
    boolShare ← sortedRecordShares[idx]
    spdzShare ← BOOL2SPDZ(boolShare)
    sortedRecordShares[idx] ← spdzShare
end
/* Run the sliding window over the record
   shares.                                          */
windowSize ← ⌈1 + log(sortedRecordShares.size())⌉
results[0...sortedRecordShares.size()−1] ← FALSE
windowLeftIdx ← 0
windowRightIdx ← windowSize
while windowLeftIdx < sortedRecordShares.size()−1 do
    leftRecord ← sortedRecordShares[windowLeftIdx]
    rightRecordIdx ← windowLeftIdx+1
    while rightRecordIdx < windowRightIdx+1 AND
      rightRecordIdx < sortedRecordShares.size() do
        rightRecord ← sortedRecordShares[rightRecordIdx]
        matchResult ← ISMATCH(leftRecord, rightRecord)
        results[leftRecordIdx] ← OR(results[leftRecordIdx],
          matchResult)
        results[rightRecordIdx] ← OR(results[rightRecordIdx],
          matchResult)
        rightRecordIdx++
    end
    windowLeftIdx++
    windowRightIdx++
end
return results
```

Assuming that the window size is w and that the total number of records is n, the total number of comparisons is $O(nw)$. This is because each record is compared with at most $w-1$ records to its right. Since the window is used to prune the total number of comparisons performed, using larger windows results in larger numbers of comparisons. As such, we can expect the accuracy of the protocol to increase as the window size increases, this of course comes at the cost of slower runtimes. As long as the window size is sub-linear in the number of records, our protocol has an overall subquadratic runtime. Given that the merge sort in Step 2 runs in $O(n\log(n))$, there is no gain in our asymptotic runtime complexity by using a window with size $O(\log(n))$. Consequently, we use a window with size $O(\log(n))$. Our experiments in Section V show that our protocol's accuracy with such a window size is comparable with the results obtained from the Laplace Protocol experiments [16].

2) Matching function: Algorithm 2 makes use of a function called ISMATCH, which is a function agreed upon by the two parties which returns a boolean indicating whether two given records match one another. The matching function that we use in our implementation is the same as the one that was used in the Laplace Protocol's evaluation [16]:

$$m(r,s) = \|r.time - s.time\| \leq t \wedge$$
$$(r.lon - s.lon)^2 + (r.lat - s.lat)^2 \leq d^2,$$

where $t = 1$ hour and $d = 0.001$.

To optimize the computation of ISMATCH, we note that we can break the matching function into two separate circuits. The following subsections describe this optimization.

3) Computing distances with SPDZ: The matching function performs a fair number of arithmetic operations (addition, subtraction, and squaring). Since arithmetic operations are computed faster in an arithmetic circuit than they are in a boolean circuit, we use SPDZ to compute the "arithmetic parts" of the matching function. This explains why the first thing we do in Algorithm 1 is ensure that our shares are SPDZ shares (using BOOL2SPDZ). The outputs of this arithmetic circuit are secret shares that represent the numerical values that are yet to be compared to the specified threshold values ($t = 1$ hour and $d = 0.001$). We use these arithmetic results, or "distances", in the next circuit.

4) Comparing the distances with TinyTables: Once the arithmetic computations are complete, the matching function performs two inequality checks and computes an AND gate. Since comparisons and logical operations are much more efficiently computed in boolean circuits than they are in arithmetic circuits, we use TinyTables to compute the "boolean part" of the matching function. First we must of course convert the previously computed SPDZ shares into TinyTables shares. Once we obtain the boolean shares, we use a boolean circuit to compare the arithmetic results from earlier with the desired threshold values. The output of this boolean circuit is a secret share of a bit that indicates whether the two records currently being compared with one another are a match for each other. Next, we explain how the result of this comparison is stored.

5) Writing the results: The (secret shared) output of the ISMATCH function is stored in a `results` array. This is a bit array with as many bits as there are records across both parties, where the i^{th} index in the array indicates whether the i^{th} record in the sorted list of records possesses a matching record. Since we require a data independent protocol, we are unable to exclusively write results only when the results are `TRUE`. In fact, since we are invoking a secure computation to compute ISMATCH, we do not know the result of any comparison until after the overall protocol is completed. Consequently, we write the result of each comparison to the results array. Once the result of a comparison is obtained, the `results` array is updated with an invocation of an SMC to compute the logical OR of the secret share of the current value in the `results` array and the secret share of the newly computed `matchResult`. Note that since comparisons are commutative, i.e., ISMATCH(r, s) always equals ISMATCH(s, r), we are able to write the result of a comparison for both records simultaneously. This is important as it saves us from duplicating calls to the matching function to check for a match between two records that have been previously compared, in other words, we need only perform comparisons while moving the sliding window in one direction.

Once all comparisons have been completed and all results have been written to the results array, both parties will possess secret shares of a bit array that indicates which records have matches. This array will have as many bits as there are records, i.e., if the parties have a total of n records across both their datasets, the results array will have n bits.

E. Step 4: Shuffling identifiers

We now reach the final step of our protocol. The order of the bits in the results array computed in the previous step corresponds to the sorted ordering of the records from Step 2 (if the first bit in the results array is a 1 then the bit indicates that the first record in the list of sorted records possesses a match). However, this bit array itself is insufficient in revealing to the parties which records have matches, since the parties themselves do not know the sorted ordering of the records, and as we describe next, the parties cannot know this ordering without leaking information about the databases. The identifier tags from Section IV-A1 are used in solving this problem efficiently.

1) Positional leakage: Great care needs to be taken in ensuring that when the results are revealed to the parties that the sorting does not leak positional information about the records (if a party learns that all its records are at the beginning of the sorted records, the party will learn that each one of the other party's records have scores that are not lower than its own records' scores, this can be a great source of leakage and could allow a party to infer an unacceptable degree of information about the other party's data). The most straightforward solution to hiding leakage from the sorted ordering is to simply shuffle the results before revealing them to the parties. The problem however lies in performing an efficient and oblivious (so that it cannot be reversed) permutation. Huang et al. experience a similar problem in their protocol [18] and end up developing an SMC permutation network to obliviously perform a permutation in $O(n \log(n))$ time. However, through our use of identifiers, we are able to achieve the effect of an irreversible permutation by simply performing a non-private permutation. In other words, we are able to perform an oblivious permutation for almost free within our problem setting. We provide an explanation in the upcoming subsections.

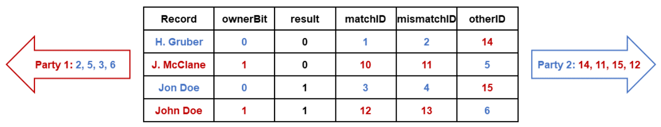
2) Writing identifiers: The goal of our protocol is to output (write) identifiers chosen by Party P_1 solely to Party P_2 and vice versa. Furthermore, P_1 should obtain a `matchID` identifier for each of P_2's matching records and a `mismatchID` identifier for each non-matching record. Ideal Functionality 1 describes how we write identifiers, with both parties executing the code as a secure multiparty computation over boolean shares. Figure 2 provides an exemplary output.

Ideal Functionality 1: writeIDs(results, partyId, sortedRecordShares)

```
output_p1 ← []
output_p2 ← []
for idx ← 1 to results.size() do
    result ← results[idx]
    recordTag ← sortedRecordShares[idx].recordTag
    ownerBit ← recordTag.ownerBit
    matchID ← recordTag.matchID
    mismatchID ← recordTag.mismatchID
    otherID ← recordTag.otherID
    if partyId == 1 then
        if ownerBit == 1 then
            /* This record belongs to party $P_1$.
               */
            /* Recall that otherID was generated
               by the other party, $P_2$ here.     */
            output_p1.append(otherID)
        else
            /* This record belongs to party $P_2$.
               */
            /* Recall that the remaining two
               identifiers were generated by the
               same party that owns the record,
               $P_2$ here.                          */
            if result == 1 then
                output_p1.append(matchID)
            else
                output_p1.append(mismatchID)
            end
        end
    else
        if ownerBit == 0 then
            output_p2.append(otherID)
        else
            if result == 1 then
                output_p2.append(matchID)
            else
                output_p2.append(mismatchID)
            end
        end
    end
end
if partyId == 1 then
    /* This array contains only identifiers that
       were generated by party $P_2$.              */
    return output_p1
else
    return output_p2
end
```

When we complete this computation, we return to party P_i an array of identifiers. Note however that the identifiers that are returned to party P_i are solely identifiers that P_{3-i} generated. Recall from Section IV-A1 that parties generate and append `matchID` and `mismatchID` identifiers to their own records and that the `otherID` identifiers that are appended to a party's records are actually generated by the other party. Reviewing Ideal Functionality 1 carefully, the reader will note that the identifiers in `output_p1` are all identifiers that were generated by P_2 (and vice versa, for the identifiers in `output_p2`). Since the identifiers that the parties receive in the outputs were not generated by themselves, the parties are unable to gain any information from the identifiers. Therefore, it is safe to complete our secure computation and open the secret shares of these individual output arrays to the respective parties (i.e., open `output_p1` to P_1 and `output_p2` to P_2).

3) Free shuffling: We have not yet shuffled the sorted ordering of the records. However, when we open the output arrays to the respective parties, we leak no information since the parties are unable to gain any information from the identifiers that were generated by the other party. Of course, the two parties cannot yet exchange their output arrays, since the arrays are in a sorted order — so we simply require that the parties perform a "free" (non-private) shuffle on the output arrays before they exchange their output arrays with one another. After the output



Record	ownerBit	result	matchID	mismatchID	otherID
H. Gruber	0	0	1	2	14
J. McClane	1	0	10	11	5
Jon Doe	0	1	3	4	15
John Doe	1	1	12	13	6

Fig. 2: Example output of `writeIDs`. Text in red is input by Party 1, text in blue is input by Party 2.

arrays are exchanged, the parties will each possess an array of `matchID`, `mismatchID`, and `otherID` identifiers, and having generated these identifiers themselves, the parties will be capable of recognizing to which class of identifiers each identifier belongs. The `matchID` identifiers will indicate to P_i which of its records have a match in the other party's database. Party P_i can ignore the other identifiers as they offer no additional information. The `mismatchID` identifiers will indicate to P_i which of its records do not have a match and the `otherID` identifiers will correspond to a record that belongs to the other party. Reviewing Ideal Functionality 1, we note that the `otherID` identifier is assigned regardless of whether the record has a match. Since the identifiers were shuffled before they were exchanged, the parties are unable to learn any positional information that may have been gained from the sorted ordering obtained in Step 2.

F. Analysis

In this section we analyze the theoretical runtime complexity of our protocol. We then show that our protocol offers cryptographically guaranteed security.

1) Runtime complexity: Let us assume that the total number of records across both parties' datasets is n. Then, analyzing the steps of our protocol sequentially, we see that:

- Step 0 of the protocol runs in linear time, in the number of records, given that we generate $3n$ identifiers.
- Step 1 of the protocol runs in linear time, in the number of records, given that we compute a score in constant time for each one of the records within a party's dataset.
- Step 2 of the protocol runs in $O(n \log(n))$ time, in the number of records, given that that is the runtime complexity of our merge network.
- Step 3 of the protocol runs in $O(nw)$ time, where w is the size of the sliding window. If we set $w = \log(n)$, we see that Step 3 has the same asymptotic complexity as Step 2, i.e., Step 3 runs in $O(n \log(n))$ time, in the number of records. If we set $w = \text{polylog}(n)$, Step 3 runs in $O(n\,\text{polylog}(n))$ time, in the number of records, and dominates the overall complexity.
- Step 4 involves writing an identifier for each record, which is linear in the number of records, and the free shuffle performed at the end of the protocol also runs in linear time, meaning that Step 4 runs linearly in time.

Each step of our protocol is run once, and so overall, our protocol has $O(n \log(n))$ runtime complexity, when we set the window size to $O(\log(n))$ and $O(n\,\text{polylog}(n))$ runtime complexity, when we set the window size to $O(\text{polylog}(n))$.

2) Security: We give a proof by constructing a simulator for either, since our protocol is symmetric, party's view of the protocol. (Almost) each step of our protocol entails either secret sharing an input or sharing the results of a secure two-party computation using binary or arithmetic secret shares. Secret shared inputs are reconstructed in the secure two-party computation circuits. Goldreich provides a composition theorem [14] which states that when given a protocol Π, that invokes provably secure subprotocols Π_i, it suffices to use oracles O_i, replacing the subprotocols Π_i, in the proof of security of the overall protocol Π (in the semi-honest model). Hence, we can substitute each invocation of a secure two-party computation in our protocol with a call to an oracle. The outputs of the oracle can be simulated using independent, uniformly chosen random numbers, since the outputs of the subprotocols are secret shares.

In Step 0 (setting parameters and identifiers), each party locally generates and stores `matchID` identifiers and `mismatchID` identifiers for its own records. Later, each party generates and stores `otherID` identifiers for the other party's records. These `otherID` identifiers are exchanged and received as secret shares, a process which we can simulate with uniform random numbers. Step 1 (scoring) of our protocol is done locally. Step 2 (sorting) and Step 3 (sliding window) are simulated as above. In Step 4 (shuffling identifiers), we start similarly by securely computing Ideal Functionality 1 to write record identifiers to output arrays for each party. From the output of this step each party receives an identifier, for each record in the protocol, which has been chosen uniformly at random and without replacement by the other party (in Step 0). Hence, these identifiers can be simulated easily. Each party locally permutes its set of identifiers and sends the shuffled identifiers to the other party. The identifiers received by each party are the following: 1) all the `otherID` identifiers that this party generated for each record belonging to the other party's records, 2) a `matchID` identifier for each record that belongs to this party that also has a matching record belonging to the other party, and 3) a `mismatchID` identifier for each record that belongs to this party but that does not have a matching record belonging to the other party. This set of identifiers can be simulated from the output of the protocol given to the simulator and the stored identifiers from Step 0. Due to the permutation, the order in which identifiers are received is random.

This completes the simulator of either party's view in our protocol. We conclude that our protocol is cryptographically secure in the semi-honest model.

V. Experimental Results

We performed experiments to empirically evaluate the SFour protocol's performance and accuracy. As such, this section is split into two parts, one dedicated to discussing the results from our performance experiments, the other for discussing the results of our accuracy experiments. Summarily, our results showed that our protocol:

- runs approximately 1-2 orders of magnitude faster than the differentially private Laplace Protocol proposed by He et al. [16], and
- achieves high accuracy.

A. *Implementation*

Our implementation of the SFour protocol uses FRESCO [1], a *framework for efficient secure computation*. FRESCO is written in Java and provides APIs to use TinyTables and SPDZ to compute boolean circuits and arithmetic circuits, respectively. One major drawback we experience while using FRESCO is the current inability to natively convert boolean circuit shares to arithmetic circuit shares (and vice versa). However, we are able to implement some custom share conversion functionality that is sufficient for our use case.

B. *Performance*

We used public datasets published by the Taxi and Limousine Commission (TLC) [37] to test our protocol's performance. A single iteration of our experiment used two databases, sizes of which we varied throughout our experiments. The TLC itself does not have data on matching records, and so we synthesized duplicated records in the same manner as did He et al. in their evaluation of the Laplace Protocol. We synthesized duplicates by adding uniformly random values drawn from the range $[-\theta^2, \theta^2]$, with $\theta = 0.001$, to the latitude and longitude values from uniformly random records selected from one database. A synthesized record was added only to the database that did not contain the original record, so a database did not contain duplicates of its own records. These duplicated records formed the ground truth of the matches to be targeted by the protocol.

For our performance experiments, we ran the secure FRESCO implementation of our protocol. Our implementation used an optimization that required each participant to have a database size of a power of two. The benefit derived from the optimization is negligible when dealing with sufficiently large datasets, and so we exclude the results obtained without use of the optimization.

1) Online phase: We ran our protocol with both parties on the same machine, since in a repeated record linkage (e.g., between two hospitals) the servers can be co-located and communication time can be neglected. Using the taskset tool we limited each party to its own single Intel Xeon E7-8860 v4 CPU @ 2.20GHz thread. Using ulimit, we were able to limit each party to using at most 108 GiB of RAM. We ran 30 trials for each database size (sufficiently many trials to obtain narrow 95% confidence intervals for our runtimes). With a window size of $O(\log(n))$ we were unable to obtain performance results for databases with more than 4096 records (i.e., when there were at most 8192 records between two parties). Since the accuracy experiments detailed in the next section varied the window sizes used in the protocol, we additionally ran performance experiments with a larger window size. With a window size of $O(\log^2(n))$ we were unable to obtain performance results for databases with more than 512 records (i.e., a total of 1024 records between two parties). However, we used a regression curve to extrapolate runtimes for larger datasets. Table I shows the online runtimes from our experiments, with Figure 3 offering a graphical representation.

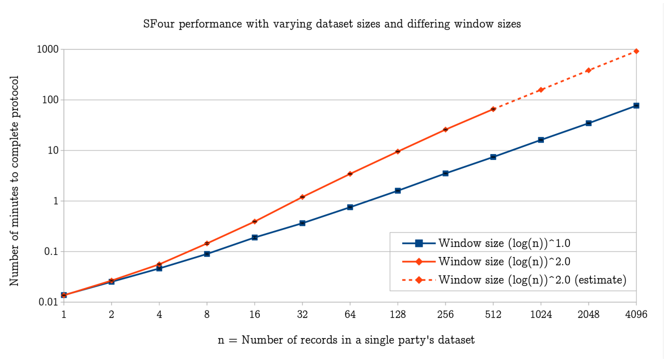


Fig. 3: SFour online runtime.

Number of records (per party)	Online runtime (95% confidence interval)	
	Window size = $\log(n)$	Window size = $\log^2(n)$
1	00m 0.8s ±0.0s	00m 0.8s ±0.0s
2	00m 1.5s ±0.0s	00m 1.6s ±0.0s
4	00m 2.8s ±0.0s	00m 3.4s ±0.0s
8	00m 5.4s ±0.1s	00m 8.7s ±0.1s
16	00m 11s ±0.2s	00m 24s ±0.2s
32	00m 22s ±0.2s	01m 12s ±0.6s
64	00m 45s ±0.4s	03m 27s ±1.7s
128	01m 36s ±0.7s	09m 32s ±4.2s
256	03m 32s ±1.6s	25m 54s ±13s
512	07m 26s ±2.5s	65m 44s ±45s
1024	16m 11s ±7.1s	†158m 23s
2048	34m 39s ±15s	†384m 54s
4096	77m 07s ±36s	†918m 36s

TABLE I: SFour online runtime. † Data representing estimated runtimes are marked.

The authors of the Laplace Protocol [16] estimate that, with two parties, each having $5,000$ records, it would take around 80 hours to complete their protocol, not including the offline encryption time required in their protocol. The results of our experiments suggest that we outperform the Laplace Protocol by over an order of magnitude, where our protocol's online phase finishes execution with two parties, each having $4,096$ records, in under 80 minutes.

2) Offline phase: Both SPDZ and TinyTables require preprocessing, but these steps can be done *independently of the data and far in advance of the online phase*. Furthermore, FRESCO's current implementation of preprocessing is incomplete and not up to the state-of-the-art and would distort the results of our experiments. As a rough estimate, we found that the offline phase would take around four times as long as the online phase, for any given input size. Hence, the combined runtime of our protocol's online and offline phases is around 400 minutes, still more than an order of magnitude faster than the Laplace protocol's online phase. Further, the preprocessing stage in FRESCO still relies on the relatively slow MASCOT

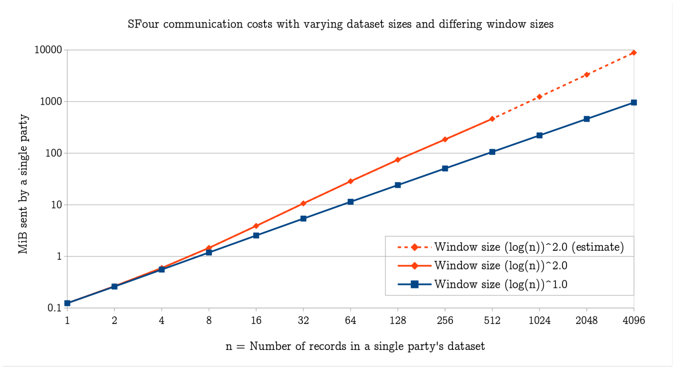


Fig. 4: SFour communication cost.

protocol [20], with work on a faster protocol, Overdrive [21], currently in progress.

3) Number of comparisons: The number of secure comparisons between records is a major factor that determines the performance of a protocol. We compare our number of secure comparisons with the numbers reported in related work [16, 31]. We perform n times the window size, e.g., $\log(n)$, secure comparisons. However, the number of comparisons in related work depends on specific locality sensitive hash functions. Hence, we compare the absolute numbers of comparisons reported for their specific datasets with our own numbers. He et al. [16] report for the Laplace protocol $9 \cdot 10^8$ comparisons for a dataset with $3 \cdot 10^6$ records at $\epsilon = 0.4$, $\delta = 10^{-5}$ and $4 \cdot 10^8$ comparisons at $\epsilon = 1.6$, $\delta = 10^{-5}$. For a dataset of the same size and a window size of $\log(n)$ we require 10^7 comparison (1.3%, 2.9% of the Laplace protocol, respectively). These savings correspond to our two orders of magnitude performance gain. Rao et al. [31] report for the Hybrid protocol $1.3 \cdot 10^6$ comparisons for a dataset with $6 \cdot 10^3$ records at $\epsilon = 1.0$, $\delta = 10^{-4}$. For this dataset size, we require $1.6 \cdot 10^5$ comparisons (12.4%).

4) Communication: We measured the number of bytes sent per party in the FRESCO framework. The numbers range from 127KiB for 2 elements in total and a window size of $\log(n)$ to an estimated 8.66GiB for 8192 elements in total and a window size of $\log^2(n)$. The numbers scale superlinearly with the circuit size. As can be seen from the high numbers for low number of elements, communication is heavily dependent on the scoring function, i.e., the number of bits per record. We used the taxi dataset with 109 to 145 bits per record depending on the dataset size. Figure 4 depicts our results.

C. Accuracy

For our accuracy experiments, we simulated the functionality of our protocol without the use of secure multiparty computations. This allowed us to test our protocol's ability to detect matches without needing to construct large SMC circuits for databases with millions of records. By construction, our protocol guarantees perfect precision, i.e., our protocol does not incorrectly determine that some record from one database has a matching record in another database, since SMC can implement any, no matter how complex, matching function. In the ground truth a record is determined to have a matching record if and only if the matching function determines that the record has a matching record. Replacing the matching function in SMC by a simpler function that only approximates the matching function may improve performance, but is also likely to introduce false positive matches. We performed various experiments to test our protocol's recall rate, i.e., the proportion of correct matches found with respect to the total number of matches in the ground truth.

In addition to testing the accuracy of our protocol with the Taxi datasets from our performance experiments, our accuracy analysis makes use of other large and publicly available datasets. While our protocol exhibits high recall rates with these datasets, it does not generally achieve perfect recall. In order to improve the protocol's accuracy, our experiments investigated the impact of performing the following:

- increasing the size of the sliding window used in the protocol, and
- running subsequent iterations of the protocol after removing matches found during earlier iterations.

We split the discussion of our accuracy experiments to focus individually on each of the differing datasets.

1) Taxi and Limousine Commission: We reused the Taxi dataset as described in the discussion of our performance experiments. Using the Taxi dataset, we synthesized duplicate records by perturbing the longitude and latitude values. The Taxi datasets combined contained six million original records with an additional 1% of synthetically generated duplicate records.

2) OpenStreetMap/Yelp: OpenStreetMap (OSM) is a project to aggregate crowdsourced geographic data [28]. We used the publicly available OpenStreetMap API to build a dataset of restaurants. The dataset that we obtained in the end contained 98,100 records. While OSM has many more restaurants in its records, our dataset included only records for which OSM had a value for each of the fields that we included in our dataset.

Yelp is a business that hosts crowdsourced reviews about other businesses. Using Yelp's open database [10] for research purposes and Yelp's publicly available API we formed a dataset of restaurants. While Yelp possesses entries for millions of businesses, the database it makes available for research purposes contains only a relatively small subset of its overall data. We augmented this dataset with the use of Yelp's API to include records for the restaurants found in the OSM dataset (using phone numbers as the primary key in both datasets). The dataset that we obtained in the end contained 239,727 records.

There were 48,669 records in one dataset that had a corresponding record in the other dataset according to our established ground truth. Of these duplicate records, only 17 were exact duplicates (around 0.03%). Across the 337,827 records from both the OSM and Yelp datasets, we have around 29% duplication of records.

3) British National Bibliography/Toronto Public Library: The British National Bibliography (BNB) contains a catalogue

Datasets	Total number of records across both datasets	Duplication (%)	Accuracy (%)							
			Window size							
			$O\log(n)$			$O\log^2(n)$			$O\log^i(n)$	
			Iteration			Iteration			i = Iteration	
			1	2	3	1	2	3	2	3
Taxi	6,060,000	1	100.0							
OSM/Yelp	337,827	29	97.0	99.3	99.5	99.4	99.9	99.9	99.8	100.0
BNB/TPL	4,053,096	17	85.4	85.8	85.9	88.3	88.4	88.4	88.4	89.7
NCVR	21,007,514	93	97.9	98.1	98.1	98.1	98.3	98.3	98.3	98.6

TABLE II: Summary of our accuracy experiments. Increasing the window size or running subsequent iterations only marginally increases the accuracy. A hybrid approach, in which the window size is increased during each subsequent iteration, may offer a good compromise for applications requiring critical levels of accuracy.

of metadata of books that have been published or distributed in the United Kingdom since the year 1950 [5]. The BNB has millions of entries, but a non-significant number of entries currently do not have associated ISBN data. Our dataset included only items that had all three fields of ISBN, title and author. The dataset that we obtained in the end contained 2,849,463 records.

The Toronto Public Library (TPL) publishes a catalogue of its collection of books and media [25]. We included only those records that had a value for ISBN, title and author. The dataset that we obtained in the end contained 1,203,633 records.

To determine the ground truth, we used the ISBN in a combination with a Levenshtein distance based metric [36] to determine the similarity between the two records in a matching pair. Only around 52% of the matching pairs were exact matches. Approximately an additional 37% of the records with matches had a similarity of at least 90% with the other record in the match pair (based on title and author name). Of the 4,053,096 records across both datasets, there were 353,434 records in one dataset that also existed in the other dataset (a duplication of around 17%).

4) North Carolina Voter Register: The state of North Carolina currently hosts a state wide public register of voter details known as the North Carolina Voter Register (NCVR) [32]. Our linkage experiments were performed across snapshots of the NCVR dataset from different time periods, specifically, we used the November 2014 (NCVR-2014) and November 2017 (NCVR-2017) snapshots. Some NCID values were repeated within a single snapshot. We removed records where such repeated usage of NCID values occurred to attempt to deduplicate each individual snapshot before performing the record linkage. In the end, we had 10,212,202 records in the NCVR-2014 snapshot and 10,795,312 records in the NCVR-2017 snapshot, for a total of 21,007,514 records in total across both datasets.

Ultimately, of the 21,007,514 records across both datasets, there were 9,792,601 records in one dataset that had a matching record in the other dataset (a 93% duplication rate). Of these duplicated records, approximately 35% of the records were exact matches (after removing age related fields from the records, otherwise none of the records would possess an exact match, given the temporal nature of the datasets).

D. Results

Table II summarizes the results of our accuracy experiments. Our protocol experiences an accuracy-performance tradeoff. We can increase the protocol's accuracy — at the cost of decreasing the protocol's performance. Accuracy can be increased by either increasing the window size or by running additional iterations of the protocol. However, our experiments suggest that performing either of these options in moderation only results in a marginal improvement in accuracy. Running a second iteration of the protocol at worse doubles the overall linkage time when the datasets do not have a large number of matching records. When there is a large number of duplicates however (perhaps such as the 93% duplication in the NCVR datasets), the first iteration may remove a significant number of records and allow the process to finish significantly faster during a second iteration. Increasing the window size, on the other hand, could result in a significantly larger runtime than running the protocol a second time, for comparable increases in accuracy (see Figure 3). Nonetheless, the protocol is parameterized by the window size, and for cases with a critical need for high accuracy, it may be worth incurring the cost in performance for better accuracy. Our experiments show that combining both approaches by first running the experiment with a relatively small window size and then increasing the window size with subsequent iterations of the protocol may provide an acceptable compromise.

VI. Conclusion

In this paper we have introduced the first known efficient private record linkage (PRL) protocol that runs in subquadratic time, provides high accuracy, and guarantees cryptographic security in the semi-honest security model. In optimizing our protocol, we have developed a new efficient technique to perform (for almost free) an oblivious permutation on inputs provided by multiple parties. Through a secure implementation of our protocol, we have shown that our protocol's runtime outperforms that of the differentially private Laplace Protocol by 1-2 orders of magnitude. Lastly, using large and publicly available real world datasets with naturally occurring matching records, we have run accuracy experiments to confirm that our protocol maintains high levels of accuracy in its output.

We stress that we only use generic secure computation protocols and secret shares as building blocks. Hence our protocol can be easily extended to the multiparty case (including the free shuffling technique). However, multiparty computations scale poorly in the number of participants. Yet, our construction allows the designation of a small set of computation servers to which a participant can supply its inputs as secret shares. We leave the experimental evaluation of this setup to future work.

It is our hope that the results of this work further encourage endeavours to develop cryptographically secure PRL protocols that do not significantly compromise on efficiency or accuracy.

Acknowledgements

We gratefully acknowledge the support of NSERC for grants RGPIN-05849, RGPAS-507908, CRDPJ-531191, and DGDND-00085, and the Royal Bank of Canada for funding this research. This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

References

[1] Alexandra Institute. FRESCO - a FRamework for Efficient Secure COmputation. https://github.com/aicis/fresco, Cited May 15, 2019.

[2] D. W. Archer, D. Bogdanov, Y. Lindell, L. Kamm, K. Nielsen, J. I. Pagter, N. P. Smart, and R. N. Wright. From keys to databases — real-world applications of secure multi-party computation. *The Computer Journal*, 61, 2018.

[3] K. Aydin, M. H. Bateni, and V. Mirrokni. Distributed balanced partitioning via linear embedding. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, WSDM '16, New York, NY, USA, 2016. ACM.

[4] K. E. Batcher and S. W. A.-H. Baddar. Sortnet: a program for building sorting networks. *Department of Computer Science, Kent State University, Kent, Ohio, USA, TR-KSU-CS-2008-01*, 2008.

[5] B. N. Bibliography. http://www.bl.uk/bibliographic/natbib.html, Cited May 15, 2019.

[6] D. Chaum, C. Crépeau, and I. Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, New York, NY, USA, 1988. ACM.

[7] P. Christen, R. Schnell, D. Vatsalan, and T. Ranbaduge. Efficient cryptanalysis of bloom filters for privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*. Springer, Apr. 2017.

[8] P. Christen, T. Ranbaduge, D. Vatsalan, and R. Schnell. Precise and fast cryptanalysis for bloom filter based privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering*, 2018.

[9] P. Christen, A. Vidanage, T. Ranbaduge, and R. Schnell. Pattern-mining based cryptanalysis of bloom filters for privacy-preserving record linkage. In *PAKDD*, 2018.

[10] Y. Dataset. https://www.yelp.com/dataset, Cited May 15, 2019.

[11] T. De Vries, H. Ke, S. Chawla, and P. Christen. Robust record linkage blocking using suffix arrays and bloom filters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5, 2011.

[12] E. Durham, M. Kantarcioglu, Y. Xue, C. Toth, M. Kuzu, and B. Malin. Composite bloom filters for secure record linkage. *IEEE transactions on knowledge and data engineering*, 26, 2014.

[13] D. Evans, V. Kolesnikov, M. Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2, 2018.

[14] O. Goldreich. Secure multi-party computation, 2002. Available from: http://www.wisdom.weizmann.ac.il/~oded/pp.html/.

[15] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, New York, NY, USA, 1987. ACM.

[16] X. He, A. Machanavajjhala, C. Flynn, and D. Srivastava. Composing differential privacy and secure computation: A case study on scaling private record linkage. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, New York, NY, USA, 2017. ACM.

[17] M. Hewitt and J. V. Simone, editors. *Enhancing Data Systems to Improve the Quality of Cancer Care.* National Academies Press (US), 2000. APPENDIX D, Information on Cancer Registries, by State. Available from: https://www.ncbi.nlm.nih.gov/books/NBK222926/.

[18] Y. Huang, D. Evans, and J. Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[19] A. Inan, M. Kantarcioglu, G. Ghinita, and E. Bertino. Private record matching using differential privacy. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, New York, NY, USA, 2010. ACM.

[20] M. Keller, E. Orsini, and P. Scholl. Mascot: Faster malicious arithmetic secure computation with oblivious transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, New York, NY, USA, 2016. ACM.

[21] M. Keller, V. Pastro, and D. Rotaru. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, Jan. 2018.

[22] H.-I. Kim, S. Hong, and J.-W. Chang. Hilbert curve-based cryptographic transformation scheme for spatial query processing on outsourced private data. *Data & Knowledge Engineering*, 104, 2016.

[23] M. Kroll and S. Steinmetzer. Automated cryptanalysis of bloom filter encryptions of health records. In *Proceedings of the International Joint Conference on Biomedical Engineering Systems and Technologies - Volume 5*, BIOSTEC 2015, Portugal, 2015. SCITEPRESS - Science and Technology Publications, Lda.

[24] T. Li, Y. Lin, and H. Shen. A locality-aware similar information searching scheme. *International Journal on Digital Libraries*, 17, 2016.

[25] T. P. Library. https://opendata.tpl.ca, Cited May 15, 2019.

[26] H. Liu, K. Wang, B. Yang, M. Yang, R. He, L. Shen, H. Zhong, Z. Chen, et al. Dynamic load balancing using hilbert space-filling curves for parallel reservoir simulations. In *SPE Reservoir Simulation Conference*. Society of Petroleum Engineers, 2017.

[27] G. Oded. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[28] OpenStreetMap. https://www.openstreetmap.org, Cited May 15, 2019.

[29] B. Pinkas, T. Schneider, C. Weinert, and U. Wieder. Efficient circuit-based psi via cuckoo hashing. In J. B. Nielsen and V. Rijmen, editors, *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*. Springer, 2018.

[30] S. M. Randall, A. M. Ferrante, J. H. Boyd, J. K. Bauer, and J. B. Semmens. Privacy-preserving record linkage on large real world datasets. *Journal of biomedical informatics*, 50, 2014.

[31] F.-Y. Rao, J. Cao, E. Bertino, and M. Kantarcioglu. Hybrid private record linkage: Separating differentially private synopses from matching records. *ACM Transactions on Privacy and Security*, 22(3), 2019.

[32] N. C. V. Register. https://dl.ncsbe.gov/, Cited May 15, 2019.

[33] R. Schnell and C. Borgs. Building a national perinatal data base without the use of unique personal identifiers. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 232–239. IEEE, Nov. 2015.

[34] R. Schnell, T. Bachteler, and J. Reiher. Private record linkage with bloom filters. *Social Statistics: The Interplay among Censuses, Surveys and Administrative Data*, 2010.

[35] R. Schnell, A. Richter, and C. Borgs. A comparison of statistical linkage keys with bloom filter-based encryptions for privacy-preserving record linkage using real-world mammography data. In *HEALTHINF*, 2017.

[36] SeatGeek. FuzzyWuzzy. https://github.com/seatgeek/fuzzywuzzy, Cited May 15, 2019.

[37] N. Taxi and L. Commission. Tlc trip record data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2018.

[38] D. Vatsalan, P. Christen, and V. S. Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38, 2013.

[39] A. Vidanage, T. Ranbaduge, P. Christen, and R. Schnell. Efficient pattern mining based cryptanalysis for privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[40] Z. Wen and C. Dong. Efficient protocols for private record linkage. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, New York, NY, USA, 2014. ACM.

[41] P. Xu, C. Nguyen, and S. Tirthapura. Onion curve: A space filling curve with near-optimal clustering. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, Apr. 2018.

[42] A. C.-C. Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, Washington, DC, USA, 1986. IEEE Computer Society.

[43] K. Zhao, H. Lu, and J. Mei. Locality preserving hashing. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14. AAAI Press, 2014.