SFour: A Protocol for Cryptographically Secure Record Linkage at Scale

by

Muhammad Basit Khurram

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

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.

Acknowledgements

I would like to thank my supervisor Florian Kerschbaum for his invaluable counsel, unwavering support, and incredible patience throughout this endeavour. I would also like to thank Alfred Menezes and Xi He for the readiness with which they agreed to serve as members of my thesis committee and ensure the quality of this work.

This work benefited from the use of the CrySP RIPPLE Facility at the University of Waterloo.

Table of Contents

List of Tables

List of Figures

Chapter 1

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 [47, 49, 51]. 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, 10, 61] aims to address this misallocation of resources. Specifically, within this field is the problem of private record linkage [57] (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 [18, 21, 26, 29, 50, 53, 54, 59], 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 [27]. 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 [57]. A specific case of private record linkage is the case of private set intersection (PSI) [26], where the intersection of two sets is determined without learning any information about elements not in the intersection. The difference between private record linkage and private set intersection is that the former allows for approximate matches or so called *fuzzy matches*, whereas private set intersection only returns elements that match exactly across the two sets. There exist efficient, secure, and accurate solutions for the private set intersection problem, but in private record linkage, the introduction of fuzzy matches adds an extra complication to the private set intersection problem which ends up requiring additional comparisons between records.

He et al. recently proposed a protocol with a subquadratic runtime and high accuracy [26]. 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 $O(n \log(n))$ 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 [26] 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 [28] for private set intersection.

The rest of this paper is divided so that we provide the necessary background knowledge in Chapter 2, some related work in Chapter 3, the details of our protocol in Chapter 4, a display of our experimental results in Chapter 5, and lastly, we conclude our paper in Chapter 6.

Chapter 2

Background

In this chapter we provide a brief understanding of secure multiparty computation (SMC), we explain the use of circuits in performing secure computations while touching upon SPDZ [15] and TinyTables [16] (the SMC protocols that we utilize in our protocol), and we explain the threat model in which our protocol operates. We conclude this chapter with the formalization of the problem that our protocol solves, the private record linkage (PRL) problem.

2.1 Secure multiparty computation

Secure multiparty computation (SMC) is an area in cryptography that attempts to allow different parties to jointly evaluate a function using their respective inputs, without requiring any party to reveal its inputs, except any that might be inferred from the output. A common approach to performing secure multiparty computations is through secure evaluation of *circuits*, a process we explain next.

2.1.1 Securely evaluating circuits with secret shares

A circuit is a structure composed of *gates*. A gate is a structure that accepts some number of inputs, and after performing some gate specific computations on the inputs (such as computing the sum of the inputs), returns the result of the computation as an output (such as the sum of the inputs). The output of one gate can be fed as an input to a second gate, which in turn may feed its output to a third gate, such compositions can continue to grow until the desired functionality is obtained. The resulting composition of gates is termed a circuit.

Circuits can be jointly computed securely through the use of *secret sharing* schemes, i.e., two parties can provide private inputs to a circuit to jointly compute the circuit, without directly revealing to one another their respective private inputs. A protocol that uses secret shares to evaluate a circuit between two parties requires each party to split each of its inputs into two parts, or *shares*. A party secret sharing one of its inputs keeps exactly one of the two shares *secret* and sends to the other party the other share. To evaluate their circuit, rather than computing directly on the private inputs, the parties evaluate their circuit by computing on the secret shares instead. Secret sharing allows a sort of proxy to allow the two parties to perform computations on private inputs without revealing to one another the private inputs. After the computations are performed on the secret shares, the secret shares can be used to reconstruct the results of the computations on the actual inputs. As an example, party P_1 may have a number, n, with a value that P_1 may wish to keep private. To secret share n with party P_2, party P_1 could draw a random number, r and send to party P_2 the value of $n' = n + r$. Here, r and n' are secret shares of n and the value of n can be reconstructed by a simple subtraction, $n = n' - r$. Using secret shares, any function that can be represented by a circuit can be evaluated securely. There are several kinds of protocols that use secret sharing to securely evaluate circuits. We refer the reader to Beimel's survey [7] for a comprehensive discussion of secret sharing protocols. For the purposes of our work, we emphasize the distinction between a boolean circuit and an arithmetic circuit, and describe the protocols that we use to securely evaluate each of these types of circuits.

2.1.2 Secure evaluation of boolean circuits

A boolean circuit is a set of logic gates that each take inputs that are 0s or 1s and give an output of a 0 or a 1, the output being the result of a logical operation (AND, OR, XOR, etc.). While a boolean circuit can compute arithmetic operations, such as addition and multiplication, boolean circuits are not as efficient as arithmetic circuits at computing arithmetic operations.

A comparator [37] is a composition of boolean operations that takes two bits and returns a result indicating whether one bit is smaller than the other. Comparators are fundamental in performing sorting operations, and since comparators make use of logical operations, sorting is generally much faster when using boolean circuits than when using arithmetic circuits. As later described in Chapter 4 our protocol requires use of a sorting circuit and indeed, we use a boolean circuit to perform this sorting.

The SMC protocol that we use to securely compute boolean circuits is TinyTables [16], which is a two party linear secret sharing based SMC protocol for computing boolean circuits. Linear secret shares for boolean circuits are bits that are closed under addition modulo 2, and which allow the reconstruction of a secret value through the summation of the secret shared bits. These sort of secret shares allow participants to compute linear boolean functions (XOR and NOT) locally. Jointly computing an AND gate with linear secret shared values from two parties requires communication between the two parties. This is not specific to TinyTables and is simply a limitation when using linear secret sharing schemes. To aid with the computation of AND gates, the execution of the TinyTables protocol is split into two stages: an offline stage and an online stage. During the offline stage the two parties communicate correlated randomness with one another to generate *preprocessed data* that they later use during the online stage to compute AND gates. This preprocessed data is stored in lookup tables which in turn inspire the protocol's name, *TinyTables*.

2.1.3 Secure evaluation of arithmetic circuits

An arithmetic circuit is a generalization of a boolean circuit over a field with more than two elements, i.e., an arithmetic circuit is a set of gates that each can take inputs that are elements of the underlying field, with the limitation that arithmetic circuits can only compute arithmetic operations (addition and multiplication). While arithmetic circuits can simulate the computation of boolean operations, boolean circuits are more efficient than arithmetic circuits at computing boolean operations.

The SMC protocol that we use to securely compute arithmetic circuits is SPDZ [15], a linear secret sharing based SMC protocol for computing arithmetic circuits with two or more participants. The arithmetic shares used in SPDZ are additive so that protocol participants are able to use these shares to perform linear arithmetic computations (such as addition and multiplication with a constant) locally. Similar to computing AND gates in boolean circuits, when using an arithmetic circuit, multiplications of secret shared values from multiple parties require communication between the different parties. To aid in computing multiplications, the execution of the SPDZ protocol is split into two stages, an offline stage and an online stage, just as is done in TinyTables. This preprocessing step in SPDZ produces what are known as *Beaver Triples* [6] to allow relatively easy computation of multiplications during the online stage.

2.1.4 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.

Different protocols have different ways to compute preprocessed data, such as lookup tables in TinyTables and Beaver Triples in SPDZ. However, since this preprocessed data can be precomputed far in advance of the online stage (in fact, the offline stage can occur before either party has any data on which to compute), analysis of a protocol's runtime performance is often focused on the protocol's online stage [3, 9, 14, 35, 56], sometimes with no regard to the offline stage. Consequently, in our analysis of our protocol's runtime in Chapter 5, we omit a thorough discussion of our protocol's offline runtime.

2.2 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 $f(x, y)$) [43].

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 [43]. 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 [22], or when making use of other software attestation methods that can detect deviations from a specified protocol [46]. 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. Furthermore, Goldreich et al. [24] show that any protocol that is secure in the semi-honest security model can be made secure in the malicious security model, albeit inefficiently, through the use of what are known as zero-knowledge protocols. When we describe our protocol in Chapter 4, we prove that the SFour protocol is secure in the semi-honest model. We delegate the (efficient) adaptation of our protocol within a malicious security model to future work.

2.3 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* $m : \mathcal{R} \times \mathcal{R} \to \{0, 1\}$ 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 $\Pi : \mathcal{D} \times \mathcal{D} \to \mathcal{D}^2$, 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.

Although the problem can be generalized to any number of parties, this work focuses only on the case of two parties. Furthermore, the problem is trivially solved when all databases utilize consistent universally unique identifiers, i.e., when an entity has the same identifier across all databases in which that entity possesses a record. This of course is difficult to accomplish and we assume that such an identifier scheme typically does not exist in practice. In case such a scheme were to exist, a solution would then be to run a private set intersection protocol (a protocol that computes the intersection of two sets without revealing items not in both sets) to find exact matches from amongst the unique identifiers. This approach would fail if there were errors while entering the unique identifiers. Given a PRL protocol's ability to find approximate matches, PRL protocols have a robustness that allows them to use multiple fields in a manner that enables them to succeed even while in the presence of erroneous unique identifiers.

2.3.1 The example of the Taxi datasets

To allow for a fair comparison with the Laplace Protocol [26], we implement our protocol to specifically perform record linkage across datasets from the Taxi and Limousine Commission [55], these being the datasets that He et al. used to evaluate the Laplace Protocol. As such, our protocol requires that all participants ensure that their databases have the following required columns: a column for the time (at which a taxi picks up a passenger), a column for the longitude (from which a passenger is picked up), and a column for the latitude (from which a passenger is picked up).

Chapter 3

Related Work

Several avenues of research have been pursued in finding solutions to the PRL problem [18, 21, 26, 29, 50, 53, 54, 59], 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 [26]. 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 [26]. With APC as a benchmark, 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 [26]. The SCS protocol proposed by Huang et al. [28] 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 [18, 21, 50], offer no formal security guarantee and some of these indeed get broken soon after publishment [11, 12, 13, 39]. Wen and Dong propose the concept of garbled Bloom filters, Bloom filters that make use of secret shares to offer some cryptographic protection, and using these structures develop a new PRL protocol [59]. However, the authors make no claims of cryptographic security, and indeed, their protocol leaks significant information through the presence of false positive matches (a party may

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

send a record that the party believes has a matching record belonging to the other party, but if the match is a false positive, the party may reveal a non-matching record to the other party). There are protocols that work efficiently through the use of locality sensitive hashing (LSH) [19, 31, 32], however, He et al. show that use of an LSH can leak information.

Some authors attempt to prove the security of their protocols by (unsuccessfully) attacking their protocols themselves [13, 58]. 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 [47, 49, 51]. There are other PRL protocols that rely on the use of a trusted third party [20, 25, 30, 31, 45], however, given the difficulty of finding a trusted third party, these protocols are not practical in the real world.

Authors of previous works [26, 29] 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 [29]. More recently, He et al. [26] 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. These authors 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 [57].

Chapter 4

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 4.4.1), 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 4.1 summarizes our overall protocol.

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

4.0 Step 0: Setting parameters and identifiers

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

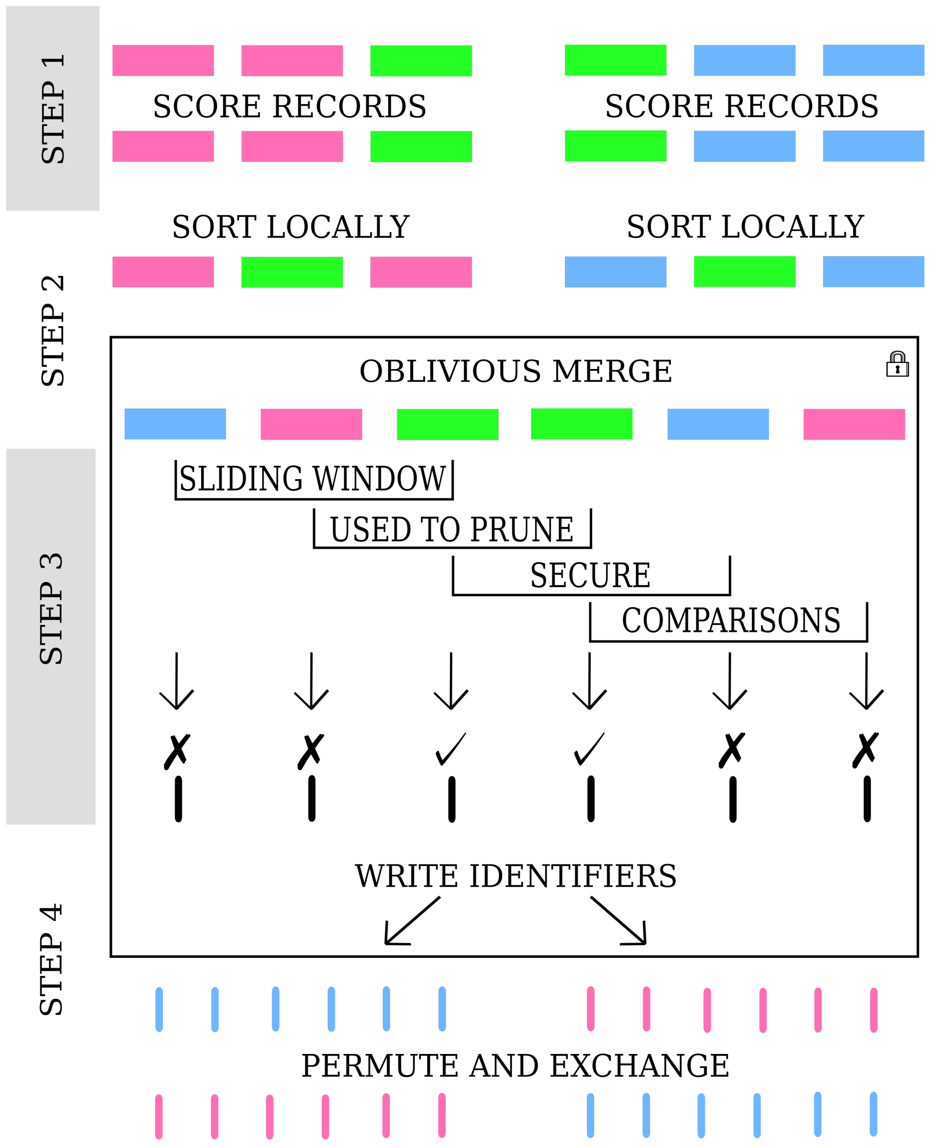

Figure 4.1: Overview of the SFour protocol. The locked box indicates use of secure multiparty computations. The small bars at the bottom of the figure indicate identifiers. We colour the identifiers with two different colours to reflect the different parties that generated the identifiers. At the end 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.

- the total number of records in the protocol;
- the scoring function to score their records (Section 4.1);
- the matching function to determine matches (Section 4.3.2).

4.0.1 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 4.2.1, and the third is in Section 4.4. 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 4.4). 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.

The reader will 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 4.2), 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 4.4.

4.1 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) [62], 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.

For medical databases containing patient data, an example of a simple scoring function is a function that returns a person's date of birth in Unix time. When two hospitals agree on this scoring function, they can proceed to sort records for people according to date of birth and then run secure comparisons only on records of patients born around the same time as one another. Of course, a simple scoring function such as this can fail if hospitals don't have reliable date of birth data, or even if one hospital has more precise data than the other hospital (for example, a person's hospital of birth may have the time in seconds that that person was born). Based on our experiments on the Taxi dataset [55] however, we are able to achieve significantly high accuracy results by using a scoring function that simply returns a timestamp.

4.1.1 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 [60]. Using SFCs to project a high dimensional vector into a single dimension has proven to be beneficial in load balancing tasks [42], performing cryptographic transformations [36], processing spatial data [4], and carrying out multidimensional similarity searches [40]. 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).

Recalling that the Taxi datasets (Section 2.3.1) in our example problem require columns for a Unix timestamp, a longitudinal coordinate, and a latitudinal coordinate, one scoring function that we used in our experiments was a function that returned the integer produced by mapping the 3-dimensional (timestamp, longitude, latitude) tuple under a three dimensional Hilbert curve.

4.2 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.

4.2.1 Identifier tags (part two)

Before we sort the records however, we recall the pseudo-identifiers mentioned in Section 4.0.1 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 4.4.3, 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.

4.2.2 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 4.1. This simple preprocessing significantly improves our protocol's performance, as we explain next.

4.2.3 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 $O(n \log(n))$ [5]. 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. FRESCO's implementation of the TinyTables protocol allows us to construct the boolean circuit for the Odd-Even 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 TinyTables secret shares of the sorted records.

Algorithm 1 summarizes the computations involved in this subsection (FRESCO's API differs from what is illustrated). 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.

4.3 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.

4.3.1 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*

Algorithm 1: Step2(partyId, records)

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

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 4.3.4).

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.

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,

Algorithm 2: Step3(sortedRecordsShares)

```
/* Convert the boolean shares to arithmetic shares.                    */
for idx ← 1 to sortedRecordsShares.size() do
    boolShare ← sortedRecordsShares[idx]
    spdzShare ← BOOL2SPDZ(boolShare)
    sortedRecordsShares[idx] ← spdzShare
end

/* Run the sliding window over the record shares.                      */
windowSize ← $\lceil 1 + \log(sortedRecordsShares.size()) \rceil$
results ← [FALSE] * sortedRecordsShares.size()
windowLeftIdx ← 0
windowRightIdx ← windowSize
while windowLeftIdx < sortedRecordsShares.size() - 1 do
    leftRecord ← sortedRecordsShares[windowLeftIdx]
    rightRecordIdx ← windowLeftIdx +1
    while rightRecordIdx < windowRightIdx + 1 AND rightRecordIdx <
     sortedRecordsShares.size() do
        rightRecord ← sortedRecordsShares[rightRecordIdx]
        matchResult ← ISMATCH(leftRecord, rightRecord)
        results[leftIdx] ← FRESCO.OR(results[leftIdx], matchResult)
        results[rightIdx] ← FRESCO.OR(results[rightIdx], matchResult)
        rightRecordIdx++
    end
    windowLeftIdx++
    windowRightIdx++
end
return results
```

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 Chapter 5 show that our protocol's accuracy with such a window size is comparable with the results obtained from the Laplace Protocol experiments [26]. Note that setting the window size to n essentially results in the All Pairwise Comparison protocol, the trivial solution that offers 100% accuracy at the cost of a quadratic runtime.

4.3.2 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 for the Taxi dataset problem is the same as the one that was used in the Laplace Protocol's evaluation [26]:

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

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.

4.3.3 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.

4.3.4 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.

4.3.5 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 a call to FRESCO.OR, which is the 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`. By using a logical OR operation, we are able to ensure that a `TRUE` result is not overwritten. 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.

4.4 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 4.0.1 are used in solving this problem efficiently.

4.4.1 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 [28] 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.

4.4.2 Writing identifiers

It is simple to iterate over the sorted records from Step 2 and replace the bits in the results array from the previous section with an identifier for the corresponding record. Doing so would allow the parties to know which records have matches. If we were to look at the results array and write a record's `matchID` identifier to an output array, whenever the results array indicated a match for that record, we would have an array that could eventually reveal which records have matches. Similarly, if we were to write `mismatchID` identifiers for records, we would know which records do not have matches. However, if we are not careful of how we select identifiers, we may unintentionally leak positional information about how the records were sorted in Step 2. To avoid this from happening, we need a data independent protocol that writes an identifier to an array, regardless of whether or not there is a match. This is why we need both `matchID` identifiers and `mismatchID` identifiers. Ideal Functionality 1 describes how we write identifiers, with both parties executing the code as a secure multiparty computation over boolean shares (using TinyTables).

Understanding the logic behind how we write these identifiers lies in understanding the output of this final SMC. 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 not identifiers that P_i generated. Recall from Section 4.0.1 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).

4.4.3 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. If the parties were able to exchange their output arrays, they would be able to derive meaning from the identifiers and be able to recognize any `matchID` identifiers in the output arrays. The parties know which of their individual records are associated with which `matchID` identifiers and so they would be able to determine which of their records have a matching record in the other party's dataset. 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 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

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

```
output_p1 ← []
output_p2 ← []
for $idx$ ← 1 to $resultBits.size()$ do
    resultBit ← resultBits[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 $resultBit == 1$ then
                output_p1.append(matchID)
            else
                output_p1.append(mismatchID)
            end
        end
    else
        if $ownerBit == 0$ then
            output_p2.append(otherID)
        else
            if $resultBit == 1$ 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
```

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.

4.5 Analysis

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

4.5.1 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.
- 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 $O(\log(n))$.

4.5.2 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 [23] 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.

Chapter 5

Experimental Results

We performed experiments to empirically evaluate the SFour protocol's performance and accuracy. As such, this chapter 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. [26], and
- achieves perfect precision (no false positive matches) and high recall.

5.1 Performance

As mentioned in Section 2.3.1, we used public datasets published by the Taxi and Limousine Commission (TLC) [55] 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.

5.1.1 Online phase

We ran our protocol with both parties on the same machine. 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 5.1 shows the online runtimes from our experiments, with Figure 5.1 offering a graphical representation.

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

The authors of the Laplace Protocol [26] 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.

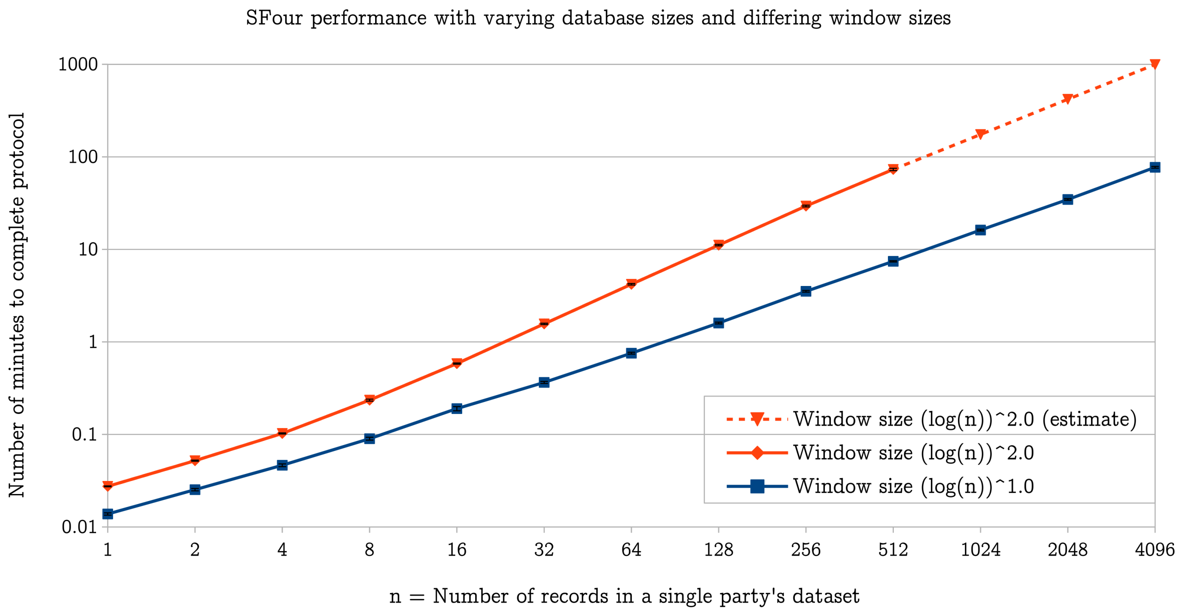

Figure 5.1: SFour online runtime.

5.1.2 Offline phase

As mentioned earlier, the preprocessing steps can be done independently of the data and far in advance of the online phase. In our experiments we reused preprocessed data, which (albeit insecure in practice) allowed us to run our experiments quicker, and so we do not offer a focused analysis of our offline runtimes. As a rough estimate however, we found that the offline stage would take around four times as long as the online stage, for any given input size. Along with the results from the previous subsection, we see that even after including our protocol's offline runtime, our protocol outperforms the online runtimes exhibited by the Laplace Protocol by around an order of magnitude. Further, the preprocessing stage in FRESCO still relies on the relatively slow MASCOT protocol [33], with work on a faster protocol, Overdrive [34], currently in progress.

5.2 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 a record is determined to have a matching record if and only if the matching function determines that the record has a matching record. The definition of the matching function is exactly the criteria used by the protocol participants in classifying any pair of records as matching records.

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 these experiments we do not explicitly define any matching functions. Since the ground truth exists, there must be some matching function that outputs booleans corresponding to the ground truth, and so we need not define any matching functions for our recall experiments. We use only the ground truth to calculate our protocol's recall rate.

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.

5.2.1 Taxi and Limousine Commission

Structure of the Taxi dataset

We reused the Taxi dataset as described in Section 2.3.1 and in the discussion of our performance experiments. Briefly, a record in the Taxi dataset contained a timestamp, a longitude coordinate and a latitude coordinate. 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.

Scoring function

We used a simple scoring function that returned only the timestamp of a record.

Results

Figure 5.2 shows that our protocol located all records that had a matching record, even as the database sizes scaled to the order of millions. This result is in line with the results obtained by the Laplace Protocol [26].

One caveat to these results lies in noticing that the duplicate synthesis process perturbs only the longitude coordinate and latitude coordinate values in a record. Given that our scoring function used only a record's unperturbed timestamp, our protocol essentially reduced to a private set intersection protocol in this case. While this allows us to perform a fair comparison with the results from the Laplace Protocol experiments, we observed that extending the scoring function to account for longitude and latitude coordinates resulted in approximately a 25% drop in recall. To allow for a better assessment of our record linkage protocol, rather than arbitrarily perturbing timestamps in the synthetically generated duplicate records, we rely on other datasets with naturally occurring matches. These datasets are described in the forthcoming sections.

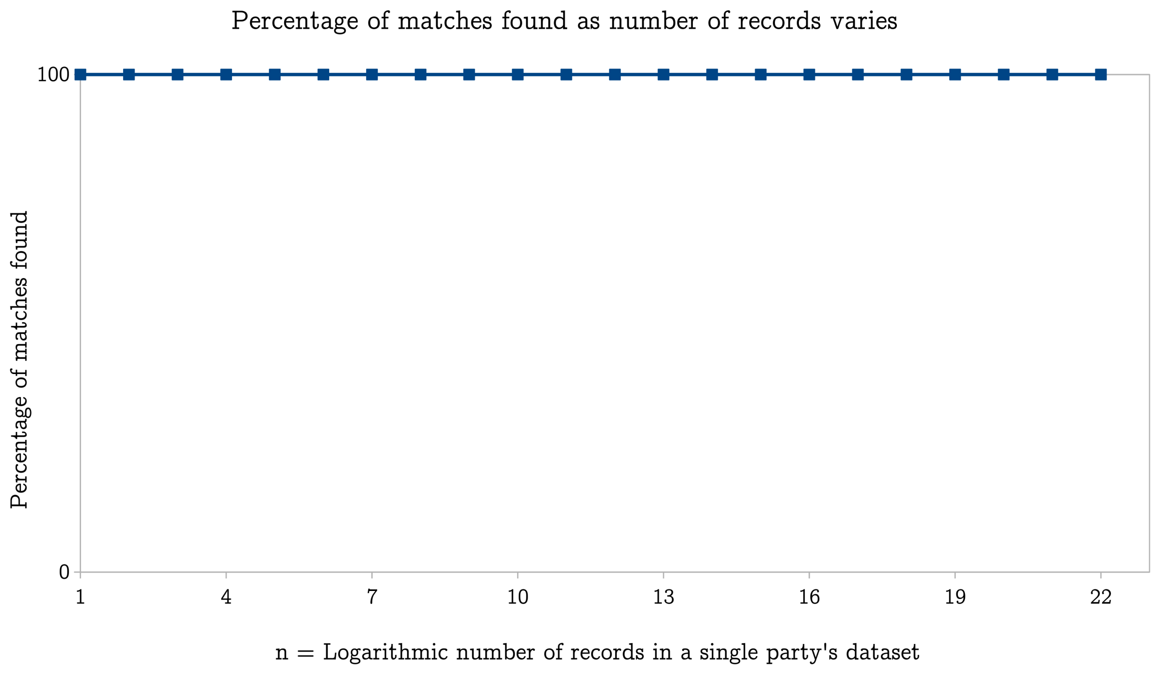

Figure 5.2: Our protocol correctly located 100% of all matches in the Taxi datasets.

5.2.2 OpenStreetMap/Yelp

Structure of the OpenStreetMap dataset

OpenStreetMap (OSM) is a project to aggregate crowdsourced geographic data [44]. A user contributing to OSM is able to use a personal GPS device to enter geographic information for places of interest. We used the publicly available OpenStreetMap API to build a dataset of restaurants. A record in our OSM dataset contained the following:

- the restaurant's phone number (used as a primary key),
- the restaurant's name,
- the restaurant's longitude coordinate, and,
- the restaurant's latitude coordinate.

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.

Structure of the Yelp dataset

Yelp is a business that hosts crowdsourced reviews about other businesses. While reviews form a large part of Yelp's crowdsourced data, this crowdsourced data also includes information such as the phone numbers and geographic locations of businesses. Using Yelp's open database [17] for research purposes and Yelp's publicly available API we formed a dataset of restaurants where each record contained the following:

- the restaurant's phone number (used as a primary key),
- the restaurant's name,
- the restaurant's longitude coordinate, and,
- the restaurant's latitude coordinate.

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.

Ground truth

We assumed that using phone numbers as primary keys in the OSM and Yelp datasets would allow us to obtain a fairly reliable ground truth of the restaurants that had corresponding records in both datasets. Of course, record linkage assumes that errors are made during data entry. As such, it is very likely that our ground truth contained matches that are unlikely to be true matches. An analysis of our ground truth, using the distances between the locations of restaurants as recorded in the two datasets, suggested that less than 2.5% of the ground truth contained errors. We considered this acceptable and used the ground truth, regardless of the few incorrect matches contained in the truth. 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.

Scoring function

Our scoring function used a Hilbert curve to project a tuple in the form (longitude coordinate, latitude coordinate) into an integer.

Results

Figure 5.3 shows the result of our accuracy experiments with the OSM/Yelp datasets. We see that our protocol using a sliding window with a size of $O(\log(n))$ found just over 97% of the matching records with a single iteration of the protocol. Here we see that our protocol displays high accuracies, even when only around 0.03% of the duplicates are exact matches. A combination of increasing the window size and running repeated iterations of the protocol enabled the protocol to find all matches.

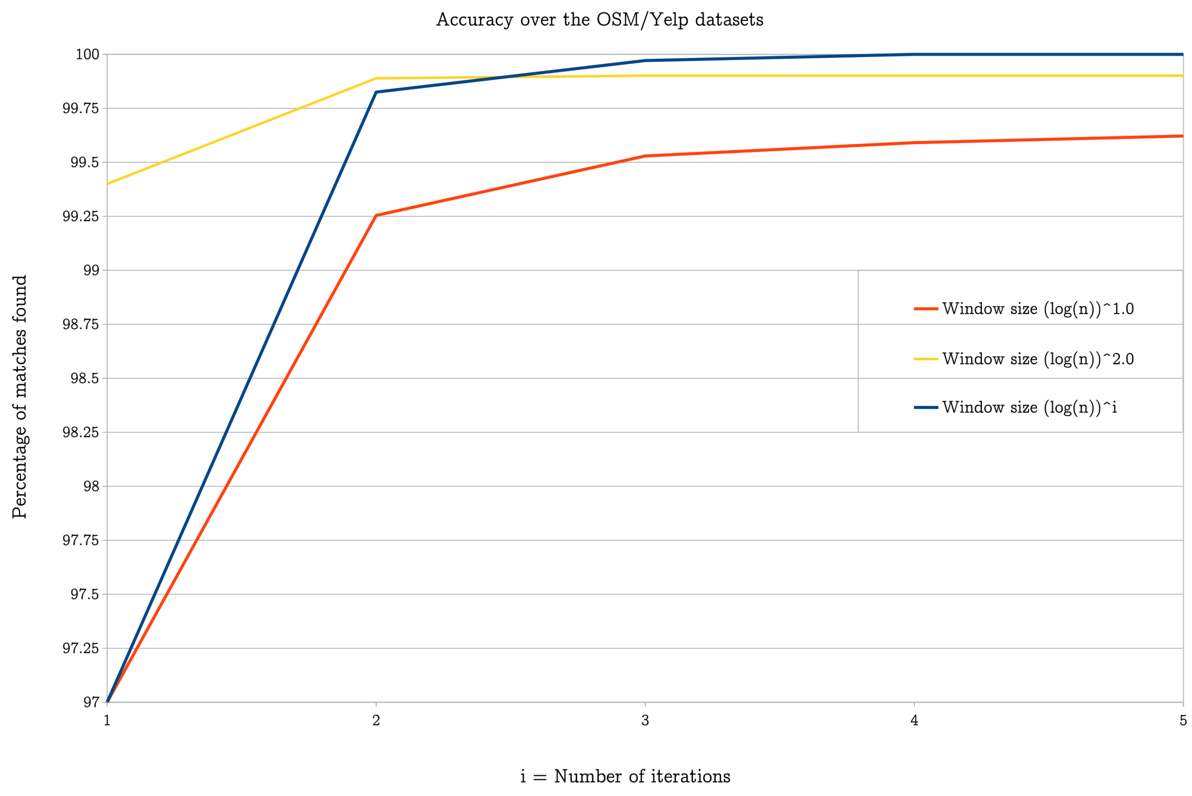

Figure 5.3: Change in recall rate with subsequent iterations of the protocol and with different window sizes.

5.2.3 British National Bibliography/Toronto Public Library

Structure of the British National Bibliography dataset

The British National Bibliography (BNB) contains a catalogue of metadata of books that have been published or distributed in the United Kingdom since the year 1950 [8]. We downloaded the catalogue from the BNB and using this data created a dataset of bibliographical information where each record in our dataset contained the following:

- the item's ISBN (used as a primary key),

- the item's title, and
- an author associated with the item.

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 of the above fields. The dataset that we obtained in the end contained 2,849,463 records.

Structure of the Toronto Public Library dataset

The Toronto Public Library (TPL) publishes a catalogue of its collection of books and media [41]. As with the British National Bibliography data, we downloaded the TPL catalogue and created a dataset of bibliographical information from the TPL catalogue. Each record in our TPL dataset contained the following:

- the item's ISBN (used as a primary key),
- the item's title, and
- an author associated with the item.

We included only those records that had a value for each of these three fields. The dataset that we obtained in the end contained 1,203,633 records.

Ground truth

Similar to how we used phone numbers to build our ground truth of matches with the OSM and Yelp datasets, we assumed here that using ISBNs would be a reliable way to form the ground truth of matches for items contained in the BNB and the TPL catalogues. To determine the quality of matches contained in the ground truth, we used a Levenshtein distance based metric [52] 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%).

Scoring function

The scoring function that we used for these bibliographical datasets differed greatly from the datasets that we have discussed thus far. The Taxi, OSM, and Yelp datasets each had numerical fields (the longitude and latitude coordinates) which allowed for a natural method to feed a record to a Hilbert curve and obtain an integer as the score from the output of the Hilbert projection, H,

$$score = H((latitude, longitude)).$$

With fields containing string values, we needed to be able to encode strings in an appropriate manner, without losing the benefits of the distance preserving properties of the Hilbert curve. We accomplished this by using two kinds of Hilbert curves. The first kind was a *field level Hilbert curve* to score a string in a single field from a single record. This field level curve accepted a numerical encoding of a string and returned an integer representing the score for the string. We numerically encoded a string by mapping each character in the string to its ASCII value ("a" was 97, "b" was 98, etc.) and returning an array of integers for that string. This numerical encoding allowed us to project a single string into a field level Hilbert curve. The intuition behind this process is that two strings that are similar to one another will have similar numerical encodings as one another. If the strings have similar numerical encodings, the field level Hilbert curve will likely generate similar scores for the numerical encodings. Ultimately, if two strings are similar to one another, they should each have similar scores as one another. Thus, we obtain a method to score a single string. Using this method, each string in a record can be scored i.e., each string can be converted to a numeric representation with some distance preserving properties inherited from the Hilbert curve.

Once the strings in a record have been scored, the record can be fed to a *record level Hilbert curve* to obtain a score for that record, as was possible with the OSM/Yelp datasets, which already contained numerical fields (longitude and latitude coordinates). Figure 5.4 displays an example of scoring a record with two string fields.

```
score = RLH(["Dinosaurs Before Dark",
             "Mary Pope Osborne"
        ])
      = RLH([FLH([ASCII("D"), ASCII("i"), ..., ASCII("k")]),
             FLH([ASCII("M"), ASCII("a"), ..., ASCII("e")])
        ])
      = RLH([FLH([68, 105, ..., 107]),
             FLH([77, 97, ..., 101])
        ])
      = RLH([119054364887637440062323834652021509531343095,
             428798712307216120274134170937361207
        ])
```

$$= 2.9380429319614636 \times 10^{87}$$

Figure 5.4: Example scoring function used for string values in a record. Here RLH and FLH refer to record level and field level Hilbert curves, respectively. ASCII(x) returns the numerical value associated with the ASCII character x.

We arbitrarily used ASCII values while encoding strings as numerical data. It is not known whether a more methodic encoding scheme may yield better results. For strings containing non-ASCII characters, we attempted first to normalize the characters to a close ASCII representation (“ümlaut” became “umlaut”). If no close ASCII representation was determined, we arbitrarily weighed a non-ASCII character as having a score of zero. Given that the records in our datasets were mostly in English, this did not greatly affect our results.

In our experiments, we used three fields to score the records from the BNB/TPL datasets. The tuples fed to the Hilbert curves were in the form (item title, author’s first name, author’s last name).

Results

Figure 5.5 shows the result of our accuracy experiments with the BNB/TPL datasets. We see that our protocol using a sliding window with a size of $O(\log(n))$ found just over 85% of the matching records with a single iteration of the protocol. This accuracy result is weaker than the results from our earlier experiments, however, it is recognized that record linkage with datasets containing strings tends to be a more challenging problem than performing record linkage with numerical fields [38]. This difference in difficulty is reflected in the relatively more complex scoring function used here when compared with the scoring functions used for the Taxi and the OSM/Yelp datasets. A combination of increasing the window size and running repeated iterations of the protocol increased the protocol's accuracy to just under 90%.

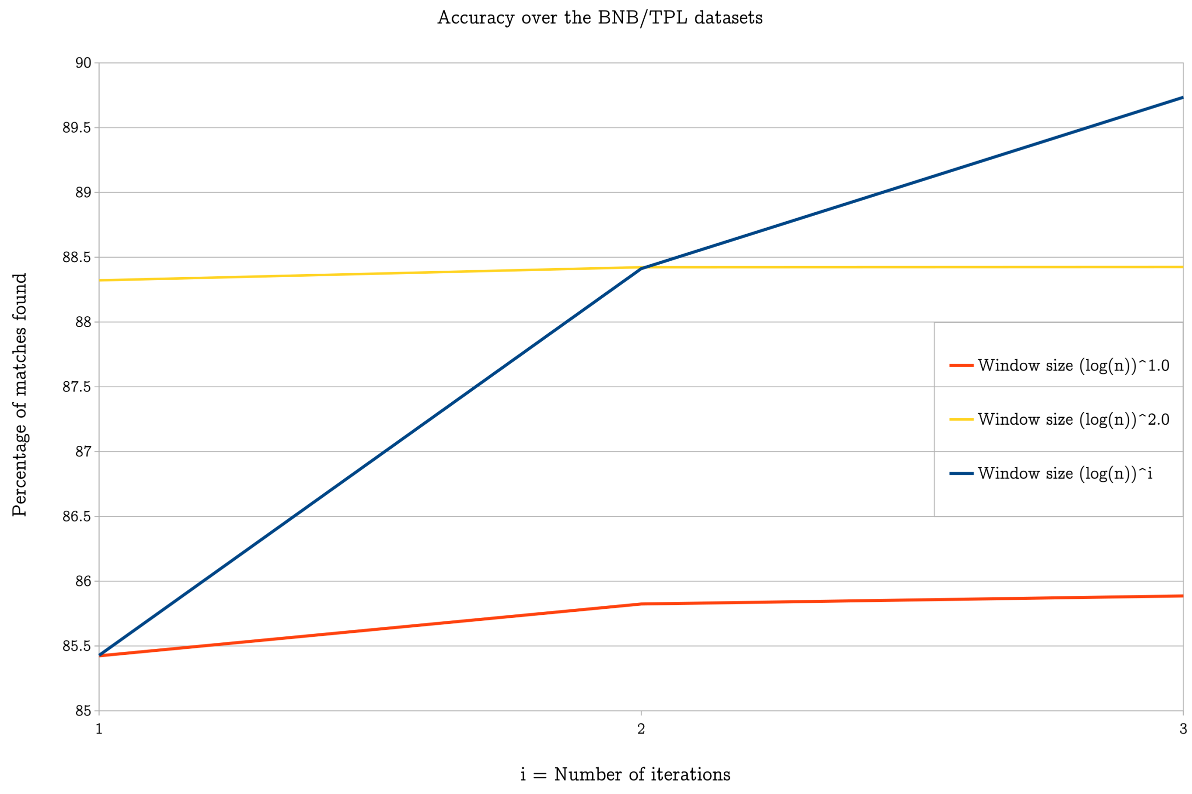

Figure 5.5: Change in recall rate with subsequent iterations of the protocol and with different window sizes.

5.2.4 North Carolina Voter Register

Structure of the North Carolina Voter Register datasets

The state of North Carolina currently hosts a state wide public register of voter details known as the North Carolina Voter Register (NCVR) [48]. Amongst other information, the NCVR contains the following voter details:

- NCID (an identifier used as a primary key),
- first name,
- middle name,
- last name,
- age,
- residential address,
- phone number,
- race,
- gender,
- place of birth, and
- political party affiliation.

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. These snapshots offered an opportunity to perform record linkage on entities that may have changed over time. 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.

Ground truth

We assumed that the NCID values would form a reliable way to generate the ground truth for the matches between the two NCVR datasets. While there were some duplicated uses of some NCIDs, it appeared that the duplicated NCIDs referred to the same entities that had been included within the same snapshot multiple times. Regardless, these repeated entities were removed from our 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).

Scoring function

The scoring function that we used for the NCVR datasets is similar to the one that we used for the bibliographical information in the BNB/TPL datasets mentioned earlier in Section 5.2.3. The difference lies only in the fields used while scoring the NCVR dataset. Here, instead of feeding (title, first name, last name) tuples to field level Hilbert curves, as was the case with the BNB/TPL datasets, we instead form (first name, middle name, last name, race, gender, place of birth) tuples for the Hilbert curves used to score the records from the NCVR datasets.

Results

Figure 5.6 shows the result of our accuracy experiments with the NCVR datasets. We see that our protocol using a sliding window with a size of $O(\log(n))$ found just under 98% of the matching records with a single iteration of the protocol. A combination of increasing the window size and running repeated iterations of the protocol marginally increased the protocol's accuracy.

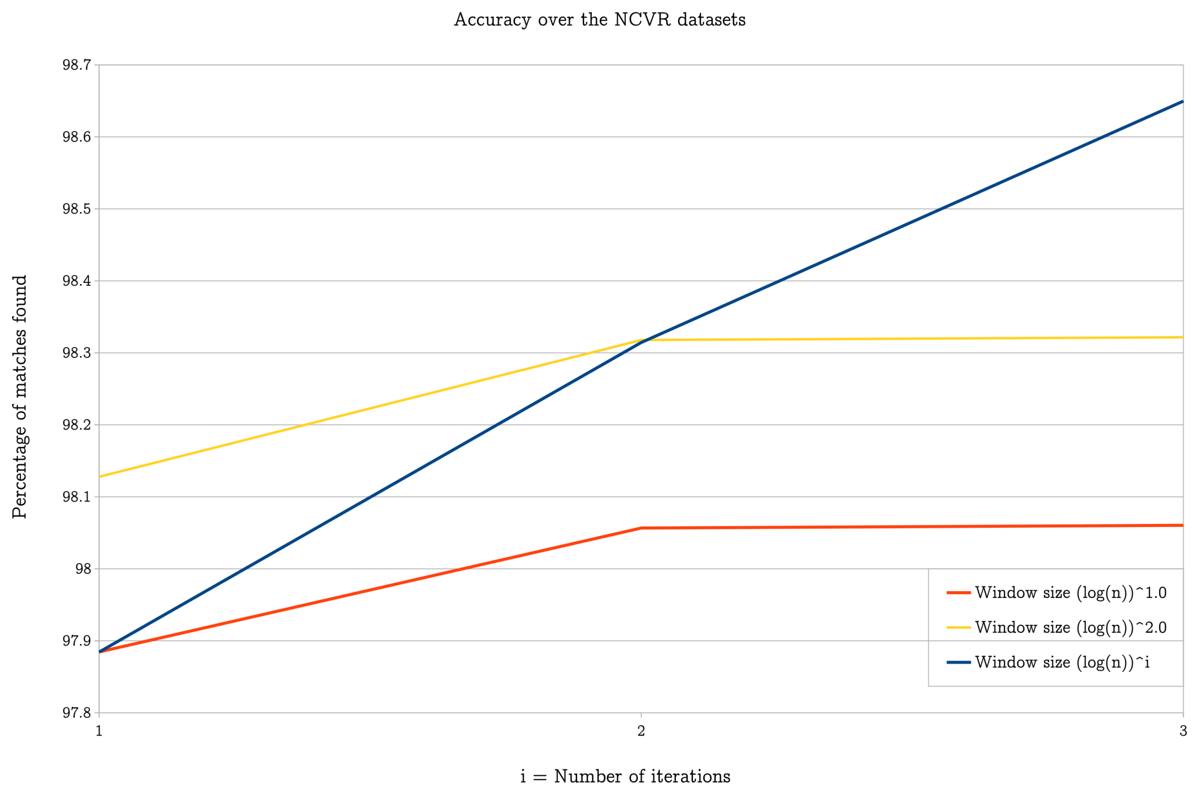

Figure 5.6: Change in recall rate with subsequent iterations of the protocol and with different window sizes.

5.3 Accuracy-Performance tradeoff

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 5.1). 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. Recall that setting the window size to the total number of records across both datasets reduces our protocol to the All Pairwise Comparison protocol and guarantees a 100% accuracy. Table 5.2 summarizes the results of our accuracy experiments.

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 5.2: 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.

Chapter 6

Conclusion

In this thesis 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.

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.

References

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

[2] David W Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P Smart, and Rebecca N Wright. From keys to databases — real-world applications of secure multi-party computation. *The Computer Journal*, 61:1749–1771, 2018.

[3] Thomas Attema, Emiliano Mancini, Gabriele Spini, Mark Abspoel, Jan de Gier, Serge Fehr, Thijs Veugen, Maran van Heesch, Daniël Worm, Andrea De Luca, Ronald Cramer, and Peter M. A. Sloot. A new approach to privacy-preserving clinical decision support systems for HIV treatment. *CoRR*, abs/1810.01107, 2018.

[4] Kevin Aydin, Mohammad Hossein Bateni, and Vahab Mirrokni. Distributed balanced partitioning via linear embedding. In *Proceedings of the Ninth ACM International Conference on Web Search and Data Mining*, WSDM '16, pages 387–396, New York, NY, USA, 2016. ACM. doi: 10.1145/2835776.2835829.

[5] Kenneth E Batcher and Sherenaz W Al-Haj Baddar. Sortnet: a program for building sorting networks. *Department of Computer Science, Kent State University, Kent, Ohio, USA, TR-KSU-CS-2008-01*, 2008.

[6] Donald Beaver. Efficient multiparty protocols using circuit randomization. In *Proceedings of the 11th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '91, pages 420–432, London, UK, 1992. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=646756.705383.

[7] Amos Beimel. Secret-sharing schemes: A survey. In *Proceedings of the Third International Conference on Coding and Cryptology*, IWCC'11, pages 11–46, Berlin, Heidelberg, 2011. Springer-Verlag. URL http://dl.acm.org/citation.cfm?id=2017916.2017918.

[8] British National Bibliography. http://www.bl.uk/bibliographic/natbib.html, Cited May 15, 2019.

[9] Charlotte Bonte, Eleftheria Makri, Amin Ardeshirdavani, Jaak Simm, Yves Moreau, and Frederik Vercauteren. Privacy-preserving genome-wide association study is practical. *IACR Cryptology ePrint Archive*, 2017:955, 2017.

[10] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 11–19, New York, NY, USA, 1988. ACM. doi: 10.1145/62212.62214.

[11] Peter Christen, Rainer Schnell, Dinusha Vatsalan, and Thilina Ranbaduge. Efficient cryptanalysis of bloom filters for privacy-preserving record linkage. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 628–640. Springer, Apr. 2017. doi: 10.1007/978-3-319-57454-7_49.

[12] Peter Christen, Thilina Ranbaduge, Dinusha Vatsalan, and Rainer Schnell. Precise and fast cryptanalysis for bloom filter based privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering*, 2018. doi: 10.1109/TKDE.2018.2874004.

[13] Peter Christen, Anushka Vidanage, Thilina Ranbaduge, and Rainer Schnell. Pattern-mining based cryptanalysis of bloom filters for privacy-preserving record linkage. In *PAKDD*, 2018.

[14] Ivan Damgård, Kasper Damgård, Kurt Nielsen, Peter Sebastian Nordholt, and Tomas Toft. Confidential benchmarking based on multiparty computation. *IACR Cryptology ePrint Archive*, 2015:1006, 2015. URL http://eprint.iacr.org/2015/1006.

[15] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P. Smart. Practical covertly secure mpc for dishonest majority - or: Breaking the SPDZ limits. *IACR Cryptology ePrint Archive*, 2012:642, 2012. URL http://dblp.uni-trier.de/db/journals/iacr/iacr2012.html#DamgardKLPSS12.

[16] Ivan Damgård, Jesper Buus Nielsen, Michael Nielsen, and Samuel Ranellucci. Gate-scrambling revisited - or: The tinytable protocol for 2-party secure computation. *IACR Cryptology ePrint Archive*, 2016:695, 2016. URL http://dblp.uni-trier.de/db/journals/iacr/iacr2016.html#DamgardNNR16.

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

[18] Timothy De Vries, Hui Ke, Sanjay Chawla, and Peter Christen. Robust record linkage blocking using suffix arrays and bloom filters. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 5, 2011.

[19] Elizabeth Durham. *A framework for accurate, efficient private record linkage*. PhD thesis, Vanderbilt University Nashville, TN, 2012.

[20] Elizabeth Durham, Yuan Xue, Murat Kantarcioglu, and Bradley Malin. Private medical record linkage with approximate matching. In *AMIA Annual Symposium Proceedings*, volume 2010. American Medical Informatics Association, 2010.

[21] Elizabeth Durham, Murat Kantarcioglu, Yuan Xue, Csaba Toth, Mehmet Kuzu, and Bradley Malin. Composite bloom filters for secure record linkage. *IEEE transactions on knowledge and data engineering*, 26:2956–2968, 2014.

[22] David Evans, Vladimir Kolesnikov, Mike Rosulek, et al. A pragmatic introduction to secure multi-party computation. *Foundations and Trends in Privacy and Security*, 2: 70–246, 2018.

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

[24] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. ACM. doi: 10.1145/28395.28420.

[25] Bilal Hawashin, Farshad Fotouhi, and Traian Marius Truta. A privacy preserving efficient protocol for semantic similarity join using long string attributes. In *Proceedings of the 4th International Workshop on Privacy and Anonymity in the Information Society*, PAIS '11, pages 6:1–6:7, New York, NY, USA, 2011. ACM. doi: 10.1145/1971690.1971696.

[26] Xi He, Ashwin Machanavajjhala, Cheryl Flynn, and Divesh 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, pages 1389–1406, New York, NY, USA, 2017. ACM. doi: 10.1145/3133956.3134030.

[27] Maria Hewitt and Joseph 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/.

[28] Yan Huang, David Evans, and Jonathan Katz. Private set intersection: Are garbled circuits better than custom protocols? In *NDSS*, 2012.

[29] Ali Inan, Murat Kantarcioglu, Gabriel Ghinita, and Elisa Bertino. Private record matching using differential privacy. In *Proceedings of the 13th International Conference on Extending Database Technology*, EDBT '10, pages 123–134, New York, NY, USA, 2010. ACM. doi: 10.1145/1739041.1739059.

[30] Alexandros Karakasidis and Vassilios S. Verykios. Reference table based k-anonymous private blocking. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 859–864, New York, NY, USA, 2012. ACM. doi: 10.1145/2245276.2245444.

[31] Dimitrios Karapiperis and Vassilios S Verykios. A distributed near-optimal lsh-based framework for privacy-preserving record linkage. *Comput. Sci. Inf. Syst.*, 11:745–763, 2014.

[32] Dimitrios Karapiperis and Vassilios S Verykios. A fast and efficient hamming lsh-based scheme for accurate linkage. *Knowledge and Information Systems*, 49:861–884, 2016.

[33] Marcel Keller, Emmanuela Orsini, and Peter 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, pages 830–842, New York, NY, USA, 2016. ACM. doi: 10.1145/2976749.2978357.

[34] Marcel Keller, Valerio Pastro, and Dragos Rotaru. Overdrive: making spdz great again. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 158–189. Springer, Jan. 2018. doi: 10.1007/978-3-319-78372-7_6.

[35] Niki Kilbertus, Adria Gascon, Matt Kusner, Michael Veale, Krishna Gummadi, and Adrian Weller. *Blind Justice: Fairness with Encrypted Sensitive Attributes*, volume 80 of *Proceedings of Machine Learning Research*. PMLR, Stockholmsmässan, Stockholm Sweden, Jul. 2018. URL http://proceedings.mlr.press/v80/kilbertus18a.html.

[36] Hyeong-Il Kim, Seungtae Hong, and Jae-Woo Chang. Hilbert curve-based cryptographic transformation scheme for spatial query processing on outsourced private data. *Data & Knowledge Engineering*, 104:32–44, 2016.

[37] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *Proceedings of the 8th International Conference on Cryptology and Network Security*, CANS '09, pages 1–20, Berlin, Heidelberg, 2009. Springer-Verlag. doi: 10.1007/978-3-642-10433-6_1.

[38] Nick Koudas, Sunita Sarawagi, and Divesh Srivastava. Record linkage: similarity measures and algorithms. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 802–803. ACM, 2006.

[39] Martin Kroll and Simone 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, pages 5–13, Portugal, 2015. SCITEPRESS - Science and Technology Publications, Lda. doi: 10.5220/0005176000050013.

[40] Ting Li, Yuhua Lin, and Haiying Shen. A locality-aware similar information searching scheme. *International Journal on Digital Libraries*, 17:79–93, 2016.

[41] Toronto Public Library. https://opendata.tpl.ca, Cited May 15, 2019.

[42] Hui Liu, Kun Wang, Bo Yang, Min Yang, Ruijian He, Lihua Shen, He Zhong, Zhangxin 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.

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

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

[45] Chaoyi Pang, Lifang Gu, David Hansen, and Anthony Maeder. Privacy-preserving fuzzy matching using a public reference table. In *Intelligent Patient Management*, volume 189, pages 71–89. Springer, Mar. 2009. doi: 10.1007/978-3-642-00179-6_5.

[46] Benny Pinkas, Thomas Schneider, Christian Weinert, and Udi Wieder. Efficient circuit-based psi via cuckoo hashing. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT (3)*, volume 10822 of *Lecture Notes in Computer Science*, pages 125–157. Springer, 2018.

[47] Sean M Randall, Anna M Ferrante, James H Boyd, Jacqueline K Bauer, and James B Semmens. Privacy-preserving record linkage on large real world datasets. *Journal of biomedical informatics*, 50:205–212, 2014.

[48] North Carolina Voter Register. https://dl.ncsbe.gov/, Cited May 15, 2019.

[49] Rainer Schnell and Christian 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. doi: 10.1109/ICDMW.2015.19.

[50] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. Private record linkage with bloom filters. *Social Statistics: The Interplay among Censuses, Surveys and Administrative Data*, pages 304–9, 2010.

[51] Rainer Schnell, Anke Richter, and Christian Borgs. A comparison of statistical linkage keys with bloom filter-based encryptions for privacy-preserving record linkage using real-world mammography data. In *HEALTHINF*, pages 276–283, 2017.

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

[53] Ziad Sehili and Erhard Rahm. Speeding up privacy preserving record linkage for metric space similarity measures. *Datenbank-Spektrum*, 16:227–236, 2016.

[54] Ziad Sehili, Lars Kolb, Christian Borgs, Rainer Schnell, and Erhard Rahm. Privacy preserving record linkage with ppjoin. *Datenbanksysteme für Business, Technologie und Web (BTW), 16. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 4.-6.3.2015 in Hamburg, Germany. Proceedings*, pages 85–104, 2015.

[55] NYC Taxi and Limousine Commission. Tlc trip record data. https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page, 2018.

[56] Maksim Tsikhanovich, Malik Magdon-Ismail, Muhammad Ishaq, and Vassilis Zikas. Pd-ml-lite: Private distributed machine learning from lighweight cryptography. *CoRR*, abs/1901.07986, 2019.

[57] Dinusha Vatsalan, Peter Christen, and Vassilios S Verykios. A taxonomy of privacy-preserving record linkage techniques. *Information Systems*, 38:946–969, 2013.

[58] Anushka Vidanage, Thilina Ranbaduge, Peter Christen, and Rainer Schnell. Efficient pattern mining based cryptanalysis for privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering*, 2019.

[59] Zikai Wen and Changyu Dong. Efficient protocols for private record linkage. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1688–1694, New York, NY, USA, 2014. ACM. doi: 10.1145/2554850.2555001.

[60] Pan Xu, Cuong Nguyen, and Srikanta Tirthapura. Onion curve: A space filling curve with near-optimal clustering. *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, Apr. 2018. doi: 10.1109/icde.2018.00119.

[61] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *Proceedings of the 27th Annual Symposium on Foundations of Computer Science*, SFCS '86, pages 162–167, Washington, DC, USA, 1986. IEEE Computer Society. doi: 10.1109/SFCS.1986.25.

[62] Kang Zhao, Hongtao Lu, and Jincheng Mei. Locality preserving hashing. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 2874–2880. AAAI Press, 2014. URL http://dl.acm.org/citation.cfm?id=2892753.2892950.