Encryption-based sub-string matching for privacy-preserving record linkage

Sirintra Vaiwsri[a,*], Thilina Ranbaduge[b], Peter Christen[c]

[a]*Faculty of Industrial Technology and Management, King Mongkut's University of Technology North Bangkok, , Prachinburi, 25230, Thailand*
[b]*Data61, Canberra, 2600, ACT, Australia*
[c]*School of Computing, The Australian National University, Canberra, 2600, ACT, Australia*

Abstract

Accurate and secure string matching in record linkage is increasingly important in application domains such as bioinformatics, healthcare, and crime detection. Most existing privacy-preserving string matching techniques provide an overall similarity between a pair of strings. As a result, these techniques cannot identify the longest common sub-string between the strings in a pair leading to lower linkage quality, while existing techniques that can identify the longest common sub-string from a pair of strings have long runtimes. While blocking techniques that can be used in the record linkage pipeline improve the time complexity, each string is generally inserted into several blocks making it vulnerable to frequency based attacks. In this paper, we propose two encryption-based approaches to improve the effectiveness and efficiency of string matching in record linkage. Our approaches compare strings based on their lengths of sub-strings. In the first approach, we encrypt the sub-string lengths into individual ciphertexts and compare a pair of ciphertexts based on the corresponding sub-string. In the second approach, we encrypt multiple lengths of sub-strings into a single ciphertext that allows efficient comparison of ciphertexts. We evaluate our approaches on real-world datasets and validate the accuracy, complexity, and privacy compared to four baselines, showing that our approaches outperform all baselines in terms of complexity and privacy while providing higher linkage quality than a standard privacy-preserving record linkage technique.

Keywords:
Privacy-preserving record linkage, String matching, Homomorphic encryption.

1. Introduction

Various application domains collect large amounts of data. These data often need to be integrated between databases to facilitate efficient data analysis [1]. Record linkage (RL) aims to link records that refer to the same entities in different databases [2]. However, such data to be linked often contain sensitive information about individuals such as patients, taxpayers, or customers [1].

Privacy-preserving record linkage (PPRL) aims to link records without the need to share sensitive information between organisations [3, 4]. PPRL techniques generally encode values of records to be linked before sending them from the organisations that own data to the organisation that conducts the linkage [4]. Only limited information about the record pairs classified as matches is being revealed at the end of a PPRL protocol, where no organisation can learn the sensitive values of these records [2]. However, secure string matching that considers the information of position or the order of characters in strings is required for applications such as the linking of financial data, individual identification numbers, and telephone numbers.

A widely used PPRL technique is Bloom filter (BF) encoding [5]. A BF is a bit vector that encodes a set of sub-strings of length q (called q-grams) in an attribute value by using a set of hash functions to set the bit position in a BF to 1. A pair of BFs can be compared by using set-based similarity functions such as the Dice-coefficient [1]. BF encoding has several drawbacks, including (1) a loss of positional information of q-grams in attribute values, (2) they return only an overall similarity between a pair of encoded strings, and (3) BFs are susceptible to privacy attacks [6, 7].

Secure multi-party computation (SMC) [8, 9] based PPRL techniques encrypt sensitive values and exchange them between organisations, where each organisation that participates in a protocol could not learn any sensitive information from each other. However, existing SMC based techniques cannot provide a high quality of linkage for data of low quality, and they consume long runtimes and computation costs.

Blocking techniques (such as q-gram based blocking [1]) are commonly used in a PPRL pipeline. These techniques aim to reduce the comparison space by grouping similar records into blocks and comparing only records in the same blocks [1]. As string matching often has long runtimes, blocking techniques can be used to speed up the runtime of the strings comparison process [2, 10].

Existing techniques such as BF encodings [5] cause lower linkage quality because of the loss of positional information and overall similarity calculation, while SMC based PPRL techniques often require high resources (especially runtimes) but cannot provide a high quality linkage, especially data with er-

*Corresponding Author
Email addresses: sirintra.v@itm.kmutnb.ac.th (Sirintra Vaiwsri), thilina.ranbaduge@data61.csiro.au (Thilina Ranbaduge), peter.christen@anu.edu.au (Peter Christen)

rors, such as missing values, variations, etc. Accurate and secure string matching was proposed by Vaiwsri et al. [11], however, their approach has long runtimes in the comparison process. Although blocking techniques have been used in various PPRL techniques to address the problem on long runtimes by reducing the number of required record comparisons, it is possible that similar strings will be added into different blocks which results in these similar strings not being compared, thus lower linkage quality.

In this paper, we propose two encryption-based PPRL approaches to provide high linkage quality and use lower runtimes of string matching between databases. Our proposed approaches are based on the lengths of sub-strings that correspond to the sub-strings of strings to be compared. In the first approach, named *one-to-one*, we encrypt the length of a sub-string into one ciphertext and encode this length into hash values of 1− or 0−encodings [12] which will be used for comparison. In the second approach, named *many-to-one*, we encrypt multiple lengths of sub-strings into a single ciphertext [13] for comparison [14]. In both approaches, the ciphertexts are sent to a semi-trusted third party for conducting the comparison. We then identify the lengths of the longest common sub-string of string pairs based on these ciphertexts. We analyse our approaches in terms of linkage quality, complexity, and privacy, and evaluate them using real-world datasets.

Based on our evaluation results, our one-to-one approach is more suitable where high accuracy is required (such as financial service and healthcare applications), while our many-to-one approach is more suitable for linking large databases that require fast, high linkage quality, and high degrees of privacy such as bioinformatics and crime detection applications.

2. Related Work

As we reviewed the literature related to string matching and secure string matching in the context of privacy-preserving record linkage (PPRL), the string matching is increasingly important in many domains. Ukkonen [15] showed that suffix trees can be used efficiently for string matching. Wang and Li [16] used a suffix tree to find similar sub-strings between groups of sub-strings. Wang et al. [17] proposed a string matching protocol based on suffix trees and edit distance constraints. Yu et al. [18] proposed an approach to search similar strings based on an index tree. However, string matching based on trees can result in high memory and time consumption.

Kim et al. [19] proposed an approximate string query approach based on sub-sequences and q-grams of sub-sequences that occur in both a string and a query string. However, this approach cannot provide accurate string matching because false positives can occur depending upon the lengths of strings, subsequences, and q-grams. Mahdi et al. [20] proposed secure sub-string searching and set-maximal searching techniques for genome data. The authors created a generalised suffix tree of data and encrypt each vertex using the Counter mode of Advanced Encryption Standard (AES-CTR) [21]. Although the experimental results show that the proposed techniques have high quality in searching string values in genome data, the search patterns can be inferred or learned by an attacker. Further, their proposed algorithm consumes high memory and runtime due to the use of generalised suffix trees.

In PPRL, BF encoding [5] is considered as a standard technique to link records [2]. BF encoding [5] is adapted for numerical values by Vatsalan and Christen [22] and Karapiperis et al. [23], where a binary vector [5] is generated based on a list that contains a number to be encoded and its neighbouring values [22] or its range of values [23]. Wu et al. [24] proposed two PPRL techniques based on differential privacy [25] and BF encoding [5] to consider privacy, fairness, and cost in PPRL. The experimental results show that these techniques outperform the standard differential privacy [26] in terms of privacy, fairness, and cost of the linkage protocol.

Xue et al. [27] proposed a sequential record linkage technique based on BF encoding, Laplace mechanism, and machine learning algorithms such as K-nearest Neighbour (KNN) and Support Vector Machine (SVM). The experimental results show that the proposed technique provides better linkage quality compared to the RAPPOR [28] technique. However, the results depend upon the fine-tuning of parameters to provide the best privacy protection. Recently, Yao et al. [29] proposed a PPRL technique based on BF and Siamese Neural Networks (SNN). They first encode records that are to be linked into BFs. Then the authors apply blocking on these BFs. The database owners send the corresponding BFs of the same blocking keys to a trusted third party who then compares BFs using the SNN technique. The main aim of their SNN-PPRL technique is classify encoded record pairs as matches and non-matches using BF similarities.

Secure multi-party computation (SMC) based techniques, such as homomorphic encryption, were proposed for sharing sensitive information between organisations [8, 9]. Essex [30] proposed a secure approximate string matching protocol based on the Damgård-Geisler-Krøigaard (DGK) homomorphic encryption [31] method, a private set intersection cardinality of encryptions of the occurrence (0 or 1) of q-grams in a string, and the list of all possible q-grams. However, this approach can result in false matches, and high runtimes and memory usage. Saha et al. [32] proposed techniques to address the problems of secure pattern matching (SPM) with wildcards. The first problem is SPM with repetitive wildcards. They address this problem by using symmetric homomorphic encryption based on ring learning with errors (ring-LWE). The second problem is the SPM with compound wildcards. They address this problem by using double-query with symmetric homomorphic encryption based on ring learning with errors (ring-LWE). Furthermore, they improved the computational time by proposing a packing method which allows for grouping all sub-patterns into a single polynomial which is being used for encryption.

Mullaymeri and Karakasidis [33] proposed a PPRL approach for approximate string matching based on a reference dataset and Fuzzy Vault [34] which is a cryptographic scheme that allows the decryption of a string if the decryption key is very similar to the encryption key. In order to encrypt and decrypt strings, the authors convert strings to polynomial coefficients and add noises for privacy protection where these noises do not

belong to the polynomial curve. However, to provide high linkage quality, the reference dataset must be very similar to the databases to be linked. Recently, Stammler et al. [35] proposed a PPRL technique using the EpiLink algorithm [36] and ABY framework [37] for secure two-party computation. The technique provides high privacy protection, however, it can reveal a number of records in the databases being linked. While dummy records can be added to improve the degree of privacy, it can result in increased computational complexity.

Nakagawa et al. [38] proposed a secure sub-string search approach for genome data based on the FM-index [39]. The FM-index is a compressed sub-string based on the Burrows-Wheeler transform [40] which is a popular technique used in the genomics domain. The proposed approach provides the longest prefix matches and the longest maximal exact match. The experimental results show that the proposed approach provides an efficient search time. However, the approach consumes a long time in data preparation. Vaiwsri et al. [11] recently proposed two approaches for secure and accurate string matching. The authors first generate a list of q-grams of a string to be linked. In the first approach, the authors conduct a hash encoding for each q-gram in the generated list and randomly shift these hashes. In the second approach, the authors generate a bit array of q-grams in the generated list, then pad this bit array with random bits. The authors also proposed a comparison mechanism which can compare string pairs more efficiently.

Most of the approaches discussed above cannot provide high linkage quality because they only allow for overall similarity calculations [5]. Some techniques also reveal the lengths of strings [11, 16, 17, 18], and some consume long runtimes even if blocking techniques are applied [11]. In our approaches, we aim to address the problem of high time complexity, while providing high linkage quality string matching between databases without any of the parties that participate in a protocol having to reveal their sensitive information.

3. Preliminaries

In this section, we describe two fundamental concepts that we use for encrypting and comparing the lengths of sub-strings in our proposed approaches.

3.1. 1- and 0-Encodings based Comparison

Lin and Tzeng [12] proposed an approach to compare encrypted integer numbers based on the sets of their special 1− and 0−encodings, where each encoding is a binary string, b. The two database owners (DOs) that participate in the protocol decide who will generate the set 1− or 0−encodings and make an agreement on the length of a binary string, l, to be used for generating encodings.

Each DO first converts its integer number into a binary string, b, of length l as $b = p_1 p_2 \cdots p_l$, where each p_i is a binary value at each position i, $1 \leq i \leq l$, in b. The DO then uses b to iteratively generate a special encoding where 1−encodings are inserted into a set $\mathbf{e}_x^1$ and 0−encodings are inserted into a set $\mathbf{e}_y^0$. Each $e^1 \in \mathbf{e}_x^1$ and $e^0 \in \mathbf{e}_y^0$ is generated using Eq. (1) and Eq. (2), respectively.

$$e^1 = p_1 p_2 \cdots p_i \text{ if } p_i = 1, \tag{1}$$

$$e^0 = p_1 p_2 \cdots p_i \leftarrow p_i = 1 \text{ if } p_i = 0, \tag{2}$$

where $\leftarrow$ is the assignment. The position i is first initialised to $i = l$ and it is decreased by one position for each iteration until $i = 1$. Therefore, a DO needs l iterations to generate a set of encodings. The two generated sets $\mathbf{e}_x^1$ and $\mathbf{e}_y^0$ are then used to conduct a comparison as:

$$comp(x, y) = \begin{cases} 1 : x > y & \text{if } \mathbf{e}_x^1 \cap \mathbf{e}_y^0 \neq \emptyset \quad (3a) \\ 0 : x \leq y & \text{if } \mathbf{e}_x^1 \cap \mathbf{e}_y^0 = \emptyset \quad (3b) \end{cases}$$

For example, the first DO has $x = 9$ and the set $\mathbf{e}_x^1 = \{\text{“1001”}, \text{“1”}\}$ generated using Eq. (1). The second DO has $y = 8$ and the set $\mathbf{e}_y^0 = \{\text{“1001”}, \text{“101”}, \text{“11”}\}$ generated using Eq. (2). These two sets have a common encoding which is “1001” ($\mathbf{e}_x^1 \cap \mathbf{e}_y^0 \neq \emptyset$). Therefore, the comparison result (following Eq. (3a) returns 1 which means $x > y$ ($9 > 8$). We use these special 1− and 0−encodings and the comparison in Eq. (3a) [12] in our first approach, as we describe in detail in Section 5.

3.2. Packing based Comparison

Cheon et al. [13] proposed a homomorphic encryption scheme, called Cheon-Kim-Kim-Song (CKKS), that supports arithmetic operations, such as addition, subtraction, and multiplication, to be conducted over ciphertexts. The authors also proposed a packing method that encrypts a list of multiple decimal numbers into a single ciphertext.

Recently, Cheon et al. [14] proposed an approach to find the minimum value between two ciphertexts that can result in errors within a $2^{-\alpha}$ bound, where α is the precision of a ciphertext. Their comparison function is based on the polynomial composition function ($\circ$), $z = f(z) \circ g(z) = f(g(z))$, where z is initialised as $z = E(\mathbf{x}) - E(\mathbf{y})$, where $E(\mathbf{x})$ is the encryption of a list of decimal numbers $\mathbf{x}$ from the first DO, and $E(\mathbf{y})$ is the encryption of a list of decimal numbers $\mathbf{y}$ from the second DO. The authors suggested optimal polynomial functions $g()$ and $f()$ that result in a maximum of 2^{-4} errors, where these functions can be written as [14]:

$$g(z) = \frac{46623}{2^{10}} z^9 - \frac{113492}{2^{10}} z^7 + \frac{97015}{2^{10}} z^5 - \frac{34974}{2^{10}} z^3 + \frac{5850}{2^{10}} z, \tag{4}$$

$$f(z) = \frac{35}{128} z^9 - \frac{180}{128} z^7 + \frac{378}{128} z^5 - \frac{420}{128} z^3 + \frac{315}{128} z. \tag{5}$$

The calculated z is then used to find a minimum value between ciphertexts $E(\mathbf{x})$ and $E(\mathbf{y})$ as:

$$min(E(\mathbf{x}), E(\mathbf{y})) = \frac{E(\mathbf{x}) + E(\mathbf{y})}{2} - \left(\frac{E(\mathbf{x}) - E(\mathbf{y})}{2} \times z \right). \tag{6}$$

As a result, the two lists (packs) of decimal numbers can be compared by using a single comparison. We use the packing method [13] and the calculations in Eq. (4) to Eq. (6) [14] in our second approach, as we describe in detail in Section 6.

Table 1: Common notation used in our approaches.

$\mathbf{D}, \mathbf{D}_A, \mathbf{D}_B$	Database, database A, and database B	$\mathbf{G}$	Publicly available global database
$\mathbf{B}$	Inverted index of blocks	$\mathbf{R}$	List of references
$\mathbf{T}$	Data to be encrypted	$\mathbf{E}$	Encrypted database
$\mathbf{M}$	An inverted index of compared results	$q, \mathbf{q}$	A length of q-gram and a list of q-grams
v, x, y	String value, string value in $\mathbf{D}_A$, and string value in $\mathbf{D}_B$	s	Sub-string value
r, *rid*	A reference value and reference identifier	*bkv*	A blocking key value
t	Threshold for generating blocking key and reference values	s_t	Similarity threshold
q_m	Minimum number of q-grams for generating *bkv* and reference values	m	Minimum length of the LCS
b	Binary string	l	Length of a binary string
d	Number of decimal places	l_c	Number of the longest common characters
l_v, l_s, l_v^s	Length of a string, sub-string, and a sub-string in a string	$\mathbf{l}_v^s$	List of l_v^s values
n_v^s	Integer form of l_v^s	*lcs*	Length of the longest common sub-string
e, $\mathbf{e}$, $\mathbf{eid}$	Encoding, list of encodings, and list of encoding identifiers	*sid*, $\mathbf{sid}$	Sub-string identifier and a set of *sid*
E	A ciphertext	*min*	Minimum value in a pair of l_v^s or ciphertexts
$\mathbf{H}, h$	List of hash values and a hash value	s_v	Secret salt value
hid	A hash value of reference identifier *rid*	*pid*, $\mathbf{pid}$	A pair of identifiers and a list of identifier pairs.
$\|...\|$	Size or number of values in a database or list		

4. Protocol Overview

In our approaches, we use notation as listed in Table 1. Our approaches involve three parties, the two DOs and a linkage unit (LU), where we assume all participants can follow the honest-but-curious [41] or malicious [42] model. The DOs want to find matches between pairs of sensitive strings in their databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, based on the lengths of the longest common sub-strings (LCS) of string pairs. The DOs do not communicate with each other except to agree on the parameters to be used in the protocol, and they do not want to reveal any sensitive information to any other parties, such as the positions of sub-strings occurring in strings, the frequencies of sub-strings, and the string values in their databases. Therefore, the DOs encrypt their sensitive values and send these encryptions to the LU to conduct the comparison.

To make all pairs of strings with different lengths be classified based on a single similarity threshold, s_t, we normalise the length of the LCS, lcs, between the string pair (x, y), where $x \in \mathbf{D}_A$ and $y \in \mathbf{D}_B$, into the range [0...1]. The pair (x, y) is classified as a match if the lcs between strings is $lcs \geq s_t$. The lcs of the pair (x, y) is calculated as:

$$lcs(x, y) = \frac{l_c}{max(l_x, l_y)} = min\left(\frac{l_c}{l_x}, \frac{l_c}{l_y}\right), \tag{7}$$

where l_x is the length of the string x, l_y is the length of the string y, and l_c is the number of the longest common characters between strings x and y.

However, if the DOs send the (encrypted) lengths of their strings, l_x and l_y (as used in Eq. (7)), or the (encrypted) string values x and y (as used in other PPRL approaches [5, 11]) to the LU for comparison, then the LU can use frequency based attacks [6, 43, 44, 45] to re-identify string values x and y of the two DOs. Therefore, in our approaches, the DOs send the (encrypted) lengths of sub-strings in strings, l_v^s, in their databases to the LU, where each l_v^s is calculated as:

$$l_v^s = \frac{l_s}{l_v}, \tag{8}$$

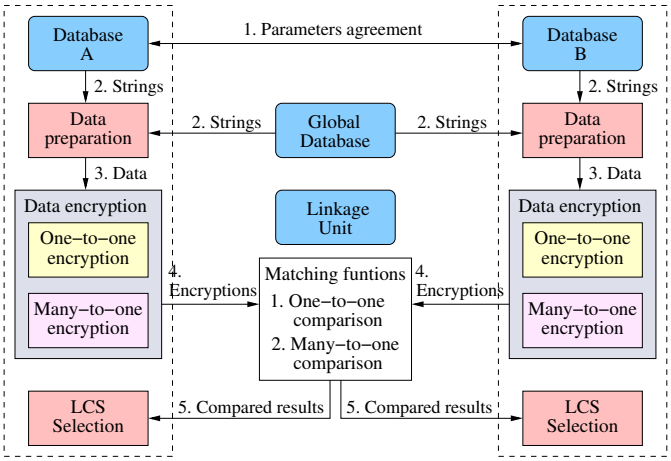

Figure 1: Overview protocol of our approaches. The rounded boxes are the DOs' databases, the global database, and the LU. The steps conducted by the DOs are shown in the dashed rectangles, where data preparation and LCS selection are common steps of our two approaches. The shaded box shows data encryption, which can be one-to-one or many-to-one. The two matching functions for comparing the ciphertext sent by the DOs are shown in the box under the LU.

where l_v is the length of the string value v and l_s is a length of a sub-string, s, that occurs in v.

Therefore, each string value v has a list of l_v^s values, $\mathbf{l}_v^s$, where each l_v^s corresponds to each sub-string s in v, such that $l_v^s \in \mathbf{l}_v^s \triangleq s \in v$, where $\triangleq$ means corresponds to. Hence, the lcs of the string pair (x, y) in Eq. (7) can be written as:

$$lcs(x, y) = \forall_{l_x^c \in \mathbf{l}_x^c,\ l_y^c \in \mathbf{l}_y^c}\ max(min(l_x^c, l_y^c)), \tag{9}$$

where l_x^c and l_y^c correspond to the length of a common sub-string c in the set of all common sub-strings $\mathbf{c}$, such that $(\mathbf{l}_x^c \equiv \mathbf{l}_y^c) \triangleq \mathbf{c}$, between strings x and y. The DOs send each of their l_v^s values ($l_x^s \triangleq l_v^s$ of x and $l_y^s \triangleq l_v^s$ of y) to the LU to find $min(l_x^s, l_y^s)$. The LU then sends the minimum length value back to the DOs which can then be used to identify the lcs of the pair (x, y) by finding the highest value ($max()$ in Eq. (9)).

As we illustrate in Fig. 1, the DOs first agree on the parameter settings to be used in the protocol. These parameters are the

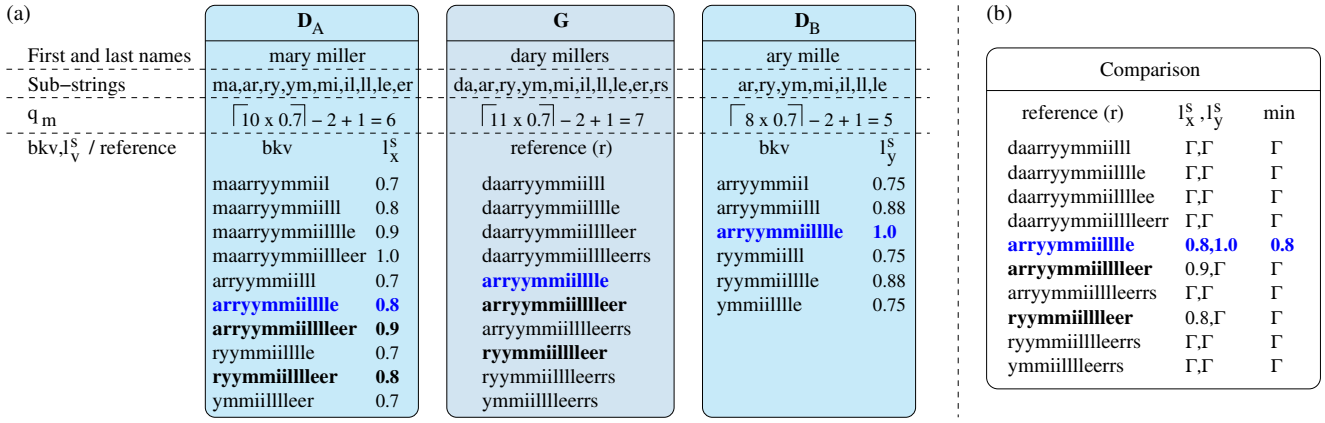

Figure 2: Example of sub-strings and their comparison results of first and last name strings. (a) shows *bkv* and l_v^s generated from strings in $\mathbf{D}_A$ and $\mathbf{D}_B$, and reference r values are extracted from the string values in $\mathbf{G}$, where these *bkv* and r values are generated by using value $q = 2$ and $t = 0.7$. (b) shows comparison results of these strings, where $s_t = 0.8$. Bold shows *bkv* values that are common with r values and the blue bold shows the sub-string that is the longest common sub-string (*lcs*). Γ is a random value in the range $[t \ldots s_t]$.

length of q-grams q, the minimum length of the LCS m where $m \leq q$, the similarity threshold s_t, the threshold t for generating blocking key values where $0 < t < s_t$, global database $\mathbf{G}$ which needs to be from the same domain as the databases to be linked, the agreement on which DO has to generate a set of 1- or 0-encodings, the number of decimal places d, the length l of the binary string of the number 10^d, the keyed hash function [2] such as HMAC(), and the secret salt value s_v.

Next, the DOs individually generate lists of sub-strings for each of their strings, where each sub-string (q-grams) is of length q. The DOs then use their lists of q-grams to generate blocks, where each blocking key value, *bkv*, represents a sub-string occurring in the strings in their databases. Therefore, if the DOs have the same *bkv*, then they have common sub-string(s), thus, the LU will be able to compare l_v^s values that correspond to the common sub-string(s). However, the frequencies of *bkv*s can reveal the frequencies of sub-strings in the DOs' databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, to the LU. Therefore, each DO uses the agreed global database, $\mathbf{G}$, to generate a list of references [46], $\mathbf{R}$, to hide the frequencies of sub-strings in its database where this $\mathbf{R}$ must be the same for the two DOs.

If the *bkv* of a block of a DO is the same as $r \in \mathbf{R}$ ($\mathbf{D} \cap \mathbf{R} \neq \emptyset$), then the DO finds the maximum value of l_v^s in its block. For any $r \in \mathbf{R}$ that is not common to any *bkv*s ($\mathbf{D} \cap \mathbf{R} = \emptyset$), each DO generates a random l_v^s value. We describe the steps of the block and reference generation process in Section 4.1. Fig. 2(a) shows example references generated from a string in $\mathbf{G}$ and example *bkv*s of $\mathbf{D}_A$ and $\mathbf{D}_B$, and Fig. 2(b) shows the corresponding comparison results.

Once the DO generated all l_v^s values for its database, in our first approach (one-to-one) each l_v^s in the inverted index is encrypted into a ciphertext, while in our second approach (many-to-one) all l_v^s values are encrypted into a single ciphertext. The DO then sends its ciphertext(s) to the LU to find the minimum (encrypted) l_v^s values between databases and return them to the DO. Finally, the DO can find the normalised LCS, *lcs*, for each string pair and check if it should be classified as a match, as we describe in Section 7.

As can be seen in Fig. 2, the pair of strings (x = "*mary miller*", y = "*ary mille*") correspond to the same reference r = *arryymmiillllle*, where the string $x \in \mathbf{D}_A$ is common with three references but the string $y \in \mathbf{D}_B$ is common with one reference. Therefore, the l_x^s, that corresponds to *bkv* in $\mathbf{D}_A$ that is common with r but not common with the *bkv* in $\mathbf{D}_B$, is compared with a random value l_y^s, $t \leq l_y^s < s_t$. This means that the pair that contains a random l_y^s value will not be classified as a match because the comparison returns a minimum l_v^s value which is $l_v^s < s_t$. Hence, the *lcs* (following Eq. (7)) between x and y is 0.8 which is the correct result because the LCS of these strings is "*ary mille*" with the sub-string length of 8. We show an example of comparison in Fig. 2(b), where Γ is a random l_v^s value.

Our approaches imply that the string comparison using the l_v^s and reference values can provide high linkage quality and the privacy of PPRL using our approaches can be improved because the DOs do not send their (encrypted) strings to the LU. Furthermore, using the reference values to represent sub-strings can reduce the computational complexity in the comparison process because no positional information is required.

4.1. Data Preparation

In this step of our protocol, each DO first generates blocks of sub-strings in its database. Algorithm 1 outlines the block generation process by a DO. In line 1, each DO first initialises an inverted index $\mathbf{B}$ to be used for storing all blocks of its database $\mathbf{D}$. In line 2, the DO initialises a set of sub-string identifiers, **sid**, to be used to ensure that all sub-strings in $\mathbf{D}$ have unique identifiers. The DO then loops over each string value $v \in \mathbf{D}$ in line 3. For each v, in lines 4 and 5 the DO checks if the length $l_v = |v|$ is at least the agreed minimum length, m.

In line 6, the DO generates the list of q-grams, $\mathbf{q}$, of v based on the agreed length of q-gram, q, by using the function *genQgramList*(). In line 7, the DO then uses the function *calNumQgram*() to calculate the minimum number of q-grams, q_m, to be used for generating a *bkv* where the q_m is calculated

Algorithm 1: ***Block generation process by a DO***

Input:
- **D**: Database
- q: Length of q-gram
- m: Minimum length of the LCS
- t: Threshold

Output:
- **B**: Blocks of lengths of sub-strings in strings (l_v^s)

1: **B** ← {} // Initialise an inverted index
2: **sid** ← {} // Initialise a set of identifiers
3: **for** $v \in$ **D do**: // Loop over strings in a database
4: $l_v \leftarrow |v|$ // Get the length of the string value
5: **if** $l_v \geq m$ **do**: // Check the length l_v is at least m
6: **q** ← $genQgramList(v, q)$ // Generate q-gram list **q**
7: $q_m \leftarrow calNumQgram(l_v, t, q)$ // Calculate number for a *bkv*
8: **bkv** ← $genBKV(\mathbf{q}, q_m)$ // Generate list of *bkv*
9: **for** $(bkv, l_s) \in$ **bkv do**: // Loop over values in the list **bkv**
10: $l_v^s \leftarrow calLenSubstr(l_s, l_v)$ // Calculate the length l_v^s
11: $sid \leftarrow genSubstrID(\mathbf{sid})$ // Generate unique identifier *sid*
12: **sid**.$add(sid)$ // Add the *sid* into the set
13: **if** $bkv \notin$ **B do**: // Check if *bkv* not exists in **B**
14: **B**$[bkv] \leftarrow \{(sid, l_v^s)\}$ // Insert a pair into **B** as a set
15: **else**: // If *bkv* exists in **B**
16: **B**$[bkv]$.$add(sid, l_v^s)$ // Add a pair under the key *bkv*
17: return **B**

Algorithm 2: ***Data generation process by a DO***

Input:
- **B**: Blocks of l_v^s values
- **R**: List of references
- t: Threshold
- s_t: Similarity threshold

Output:
- **T**: Database to be encrypted

1: **T** ← {} // Initialise an inverted index
2: **for** $r \in$ **R do**: // Loop over references in **R**
3: **if** $r \in$ **B do**: // Check if r exists in **B**
4: $(sid, l_v^s) \leftarrow getMaxLSV(\mathbf{B}[r])$ // Get maximum l_v^s and its *sid*
5: **else**: // If r not exists in **B**
6: $(sid, l_v^s) \leftarrow genRand(t, s_t)$ // Generate fake *sid* and random l_v^s
7: **T**$[r] \leftarrow (sid, l_v^s)$ // Insert a pair into **T**
8: **T** ← $genProbRand(\mathbf{T})$ // Equally random and actual l_v^s
9: return **T**

as:

$$q_m = \lceil l_v \times t \rceil - q + 1, \tag{10}$$

where $\lceil ... \rceil$ denotes rounding to the next upper integer and t is the agreed threshold. In line 8, the DO uses q_m, **q**, and the function $genBKV()$ to generate a list **bkv** that contains pairs of (bkv, l_s), where each *bkv* represents a sub-string in the corresponding string, and l_s is the length of the sub-string that corresponds to the *bkv*. For example, as we show in Fig. 2(a), let us assume the DOs agreed on $t = 0.7$ and $q = 2$. A DO has $\mathbf{D}_A$ with the string $v =$ "*mary miller*", $l_v = 10$, and $\mathbf{q} = [ma, ar, ry, ym, mi, il, ll, le, er]$. Therefore, $q_m = \lceil 10 \times 0.7 \rceil - 2 + 1 = 6$. As a result, the list **bkv** = $[(maarryymmiil, 7), (maarryymmiilll, 8), ..., (ymmiilllleer, 7)]$ will be returned from the function $genBKV()$.

In line 9 of Algorithm 1, the DO loops over each pair (bkv, l_s) in the list **bkv**. For each pair, in line 10, the DO uses the function $calLenSubstr()$ to calculate l_v^s by using Eq. (8). We show an example of l_v^s of strings in $\mathbf{D}_A$ and $\mathbf{D}_B$ as l_x^s and l_y^s, respectively, in Fig. 2(a). In lines 11 and 12, the DO uses the function $genSubstrID()$ to generate a unique sub-string identifier, *sid*, that is not available in the set of identifiers **sid**, such that $sid \notin$ **sid**. In line 13, the DO checks if the generated *bkv* does not exist in the inverted index **B**, it then inserts the *bkv* as a key into **B** and inserts the pair (sid, l_v^s) into a set under the key *bkv* of **B** in line 14. If the *bkv* already exists in **B**, the DO adds the pair (sid, l_v^s) into the set under the corresponding key *bkv* in lines 15 and 16. The DO repeats steps in lines 3 to 16 for every $v \in$ **D**.

Next, the DO generates a list of references, **R**, to hide the frequencies of sub-strings in the DO's database from the LU and allows the LU to find common sub-strings between databases. The DO conducts a similar process as the block generation process in Algorithm 1. For each v in the agreed global database, **G**, the DO checks if l_v is at least m. The DO then generates **q** for v and calculates q_m (following Eq. (10)). After that, the DO uses **q** and q_m to generate reference values. These reference values are generated in the same way as the function $genBKV()$ in Algorithm 1 in line 8, where the length of sub-string l_s will be ignored. The DO then inserts the generated reference values into **R**.

Once the DO has generated **B** and **R**, it generates the database to be encrypted. In Algorithm 2 in line 1, the DO initialises an inverted index **T**. The DO loops over $r \in$ **R** in line 2, and checks if $r \in$ **B**, then the DO uses the function $getMaxLSV()$ to select a pair (sid, l_v^s), where the l_v^s is the maximum value in the block **B**$[r]$ in lines 3 and 4. If $r \notin$ **B**, the DO uses the function $genRand()$ to generate a pair of fake *sid* and random l_v^s, where the random l_v^s must be $t \leq l_v^s < s_t$ to ensure the random l_v^s will not be classified as matches in lines 5 and 6. The DO then inserts (sid, l_v^s) into **T** under the key r in line 7, and repeats steps in lines 2 to 7 until $|\mathbf{T}| = |\mathbf{R}|$.

As we use the threshold t to calculate q_m in Eq. (10), no actual $l_v^s \in$ **B** will be less than t. In line 8, the DO uses the function $genProbRand()$ to ensure the numbers of actual l_v^s (in the range $[t \ ... \ s_t]$) and random l_v^s in **T** are equal. As the DOs agreed to use **G** which is in the same domain as their databases for generating **R**, the number of actual l_v^s with $t \leq l_v^s < s_t$ of each database should be more than the number of random l_v^s in **T**. Therefore, the function $genProbRand()$ makes the numbers of random and actual l_v^s (in the range $[t \ ... \ s_t]$) to be equal by replacing some actual l_v^s with random l_v^s. The total number of actual l_v^s that needs to be replaced with random l_v^s is calculated as:

$$n_r = n_t - \lceil (n_n + n_t)/2 \rceil, \tag{11}$$

where n_t is the total number of actual l_v^s value in the range $[t \ ... \ s_t]$ and n_n is the total number of random l_v^s in **T**.

We use t and l_v to calculate q_m in Eq. (10) because these t and l_v values help to reduce the number of *bkv* and reference values to be generated (especially for long string values such as the concatenation of attribute values first name, last name, and street address) by ignoring the corresponding *bkv* and reference values of sub-strings that are too short and their $l_v^s < t$. By using t and the function $genProbRand()$, there will be equal numbers of actual and random l_v^s values in **T** that are not being classified as matches which increases uncertainty and makes it more difficult for an adversary to identify the frequencies of

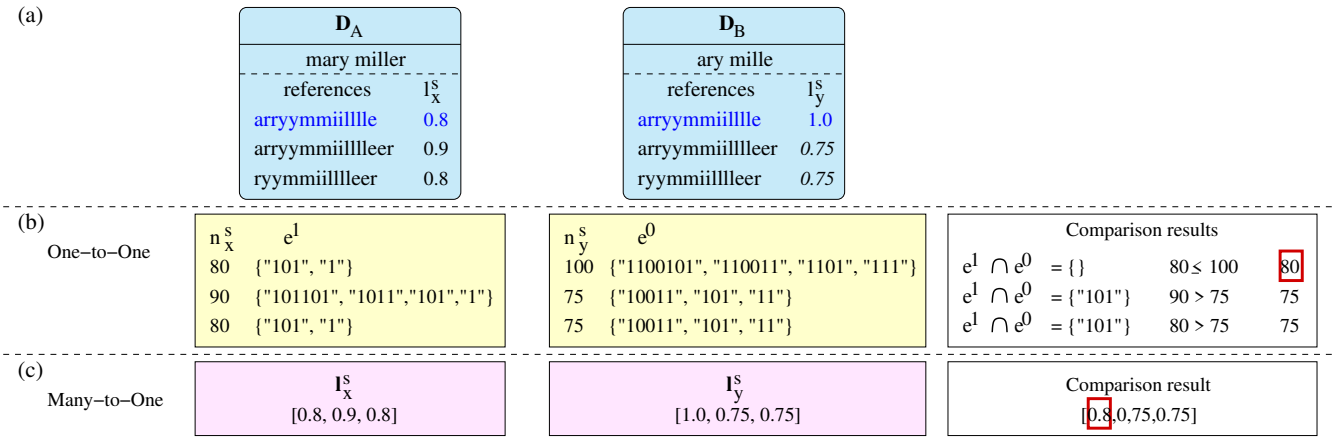

Figure 3: An example of l_v^s corresponding to references generated in Fig. 2(a) and their comparison results using our one-to-one and many-to-one approaches. Example l_v^s values are shown in (a), where the random l_v^s are shown in italic and the actual common sub-strings between $\mathbf{D}_A$ and $\mathbf{D}_B$ is shown in blue (both $\mathbf{D}_A$ and $\mathbf{D}_B$ have the same sub-string). The one-to-one approach is shown in (b), where n_v^s and their encodings of $\mathbf{D}_A$ and $\mathbf{D}_B$ are shown in yellow boxes while the comparison results are shown in the white box. The many-to-one approach is shown in (c), where the list of l_v^s of $\mathbf{D}_A$ and $\mathbf{D}_B$ are shown in pink boxes while the comparison result is shown in the white box. The selected *lcs* of the two approaches are marked with red squares in the white boxes.

sub-strings, thus, more difficult to re-identify the original string values in a database. We will discuss the privacy analysis in Section 8.3.

5. One-to-One Approach

Our one-to-one approach is based on one length of sub-string in a string (numerical value), l_v^s, encrypted into one ciphertext using the special 1- and 0-encodings [12] (as we have described in Section 3.1).

5.1. One-to-One Encryption

We apply the special 1- and 0-encodings [12] to generate the list of hash encodings for each l_v^s value. These 1- and 0-encodings will allow the LU to conduct a comparison between two encoded values in a pair. However, the comparison function in Eq. (3a) returns either 1 ($x > y$) or 0 ($x \leq y$), that the DOs will unable to select the length of the LCS. Therefore, we use homomorphic encryption functions to encrypt l_v^s into a ciphertext. The DOs then send their lists of hashes and ciphertexts to the LU. Hence, the LU can conduct the comparison using these lists, and return ciphertexts that correspond to the minimum l_v^s values of pairs to the DOs. The DOs can then decrypt the ciphertexts and select the length of the LCS of string pairs.

Algorithm 3 outlines the one-to-one encryption by a DO. In line 1, each DO initialises the inverted index $\mathbf{E}$ to be sent to the LU. In line 2, the DO then loops over $\mathbf{T}$ generated in Algorithm 2 and uses the function *genRefID*() to generate an identifier *rid* for each reference value r in line 3, where r is used as a seed to a pseudo-random number generator (PRNG) [2]. As a result, the two DOs will generate the same *rid* for the same r. In line 4, the DO converts the l_v^s to an integer value, n_v^s, by using the function *convert*(). This function first rounds the l_v^s value into the agreed d decimal places resulting in $l_v'^s$ and then converts it to n_v^s by calculating:

$$n_v^s = l_v'^s \times 10^d. \quad (12)$$

Algorithm 3: ***One-to-One encryption process by a DO***

Input:
- $\mathbf{T}$: Data to be encrypted
- d: Number of decimal places
- l: Length of binary string
- HMAC(): Keyed hash function
- *enc*: Agreement of encodings

Output:
- $\mathbf{E}$: Inverted index of encryptions

1: $\mathbf{E} \leftarrow \{\}$ // Initialise an inverted index
2: **for** $(r, (sid, l_v^s)) \in \mathbf{T}$ **do**: // Loop over data in $\mathbf{T}$
3: $rid \leftarrow genRefID(r)$ // Generate reference identifier
4: $n_v^s \leftarrow convert(l_v^s, d)$ // Convert l_v^s to an integer n_v^s
5: $E \leftarrow encrypt(n_v^s)$ // Encrypt the integer n_v^s into a ciphertext
6: $b \leftarrow genBinary(n_v^s, l)$ // Generate binary string of n_v^s
7: **if** $enc = 1$ **do**: // Check if the DO must do 1-encoding
8: $\mathbf{e}^1 \leftarrow genOneEnc(b)$ // Generate a set of special 1-encodings
9: $\mathbf{H} \leftarrow genHashList(\mathbf{e}^1, r, \text{HMAC}())$ // Generate list of hashes
10: **else**: // If the DO must do 0-encoding
11: $\mathbf{e}^0 \leftarrow genZeroEnc(b)$ // Generate a set of special 0-encodings
12: $\mathbf{H} \leftarrow genHashList(\mathbf{e}^0, r, \text{HMAC}())$ // Generate list of hashes
13: $eid \leftarrow encrypt(sid)$ // Encrypt the identifier of sub-string *sid*
14: $hid \leftarrow encrypt(rid)$ // Encrypt the identifier of reference *rid*
15: $\mathbf{E}[hid] \leftarrow (eid, E, \mathbf{H})$ // Insert tuple into $\mathbf{E}[hid]$
16: return $\mathbf{E}$

Function *genHashList*($\mathbf{e}$, r, HMAC()):

17: $\mathbf{H} \leftarrow [\,]$ // Initialise a list of hashes
18: **for** $e \in \mathbf{e}$ **do**: // Loop over encodings in $\mathbf{e}$
19: $h \leftarrow \text{HMAC(e, r)}$ // Hash encodes the encoding
20: $\mathbf{H}.append(h)$ // Add hash value into the list $\mathbf{H}$
21: return $\mathbf{H}$

The number of decimal places d is used to ensure the two DOs will generate their n_v^s in the same range $[0 \ldots 10^d]$. Fig. 3(a) and (b) show the examples of l_v^s and their corresponding n_v^s values, respectively, where the l_v^s of x is l_x^s and l_v^s of y is l_y^s. In line 5, the DO then encrypts n_v^s into a ciphertext E by using the function *encrypt*(). In this approach, we need to convert the l_v^s to n_v^s because we will generate a binary string of n_v^s for the special 1- and 0-encodings [12], as we describe next. Therefore, integer values are more suitable for this purpose.

Algorithm 4: *One-to-One comparison by the LU*

Input:
- $\mathbf{E}_1$: Inverted index of encryptions from the first DO
- $\mathbf{E}_2$: Inverted index of encryptions from the second DO

Output:
- $\mathbf{M}$: Inverted index of compared results

1: $\mathbf{M} \leftarrow \{\}$ // Initialise an inverted index
2: $\mathbf{E}_C \leftarrow \mathbf{E}_1 \cap \mathbf{E}_2$ // Find common identifiers
3: **for** $hid \in \mathbf{E}_C$ **do**: // Loop over common identifiers
4: $(eid_1, E_1, \mathbf{H}_1) \leftarrow \mathbf{E}_1[hid]$ // Get values from the first DO
5: $(eid_2, E_2, \mathbf{H}_2) \leftarrow \mathbf{E}_2[hid]$ // Get values from the second DO
6: **if** $\mathbf{H}_1 \cap \mathbf{H}_2 = \emptyset$ **do**: // Check if common hashes not exist
7: $min \leftarrow E_1$ // Select E_1 as a minimum value
8: **else**: // If common hashes exist
9: $min \leftarrow E_2$ // Select E_2 as a minimum value
10: $\mathbf{M}[(eid_1, eid_2)] \leftarrow min$ // Insert compared result to $\mathbf{M}$
11: return $\mathbf{M}$

Algorithm 5: *Many-to-One encryption process by a DO*

Input:
- $\mathbf{T}$: Data to be encrypted
- s_v: Secret salt value

Output:
- **eid**: List of encrypted identifiers
- E: Ciphertext

1: **eid** $\leftarrow$ [ ] // Initialise the list of identifiers
2: $\mathbf{l}_v^s \leftarrow$ [ ] // Initialise the list of l_v^s
3: $\mathbf{T}' \leftarrow sort(\mathbf{T})$ // Sort $\mathbf{T}$ based on reference values
4: **for** $(r, (sid, l_v^s)) \in \mathbf{T}'$ **do**: // Loop over data in $\mathbf{T}'$
5: $eid \leftarrow encrypt(sid)$ // Encrypt the identifier *sid*
6: **eid**$.append(eid)$ // Add *eid* to the list **eid**
7: $\mathbf{l}_v^s.append(l_v^s)$ // Add l_v^s to the list $\mathbf{l}_v^s$
8: **eid**, $\mathbf{l}_v^s \leftarrow permute($**eid**, $\mathbf{l}_v^s, s_v)$ // Permute the lists **eid** and $\mathbf{l}_v^s$
9: $E \leftarrow packEncrypt(\mathbf{l}_v^s)$ // Encrypt the list $\mathbf{l}_v^s$ into a ciphertext
10: return **eid**, E

In line 6, the DO converts n_v^s to a binary string b of length l by using the function *genBinary*(). Then, in lines 7 to 12, if the DO has agreed to do the 1-encoding, it uses the function *genOneEnc*() to encode b into a set of 1-encodings, $\mathbf{e}^1$, where each $e^1 \in \mathbf{e}^1$ is generated following Eq. (1). If the DO has agreed to do the 0-encoding, the DO uses the function *genZeroEnc*() to encode b into a set of 0-encodings, $\mathbf{e}^0$, where each $e^0 \in \mathbf{e}^0$ is generated following Eq. (2). Figure 3(b) illustrates the sets $\mathbf{e}^1$ and $\mathbf{e}^0$ generated for $\mathbf{D}_A$ and $\mathbf{D}_B$ from Fig. 3(a), respectively. The DO then uses the function *genHashList*() to generate a list of hash values, $\mathbf{H}$, for its set of encodings $\mathbf{e}$ ($\mathbf{e}^1$ or $\mathbf{e}^0$).

The function *genHashList*(), in lines 17 to 21, first initialises the list of hash values, $\mathbf{H}$, in line 17, and then loops over $e \in \mathbf{e}$ in line 18. In line 19, the function HMAC() is used to encode e to a hash value, h. In the function HMAC(), e is first concatenated with the reference value r into a single string e'. This r is used as a salting value [47] because it is possible that different binary strings corresponding to different references can be encoded into the same (1- or 0-) encodings, and therefore using r as a salting value can reduce the frequency distribution of each encoding e. Thus, it is more difficult for an adversary to analyse the frequencies of n_v^s. The string e' is then hash encoded by a one-way hash function (we use SHA256 [48]), resulting in a hash value h. This hash value h is then added to the list $\mathbf{H}$ in line 20. The steps in lines 18 to 20 are repeated until every $e \in \mathbf{e}$ has been hashed. Finally, the list of hash values $\mathbf{H}$ is returned in line 21.

Back to the main program, where in lines 13 and 14 the DO encrypts *sid* into a ciphertext *eid* and encrypts *rid* into a ciphertext *hid* by using the function *encrypt*(). The DO then inserts *eid*, the ciphertext E, and the list $\mathbf{H}$ into the inverted index $\mathbf{E}$ under the key *hid* in line 15. The DO repeats steps in lines 2 to 15 until every $r \in \mathbf{T}$ has been encrypted. Finally, in line 16, the DO sends its $\mathbf{E}$ to the LU.

5.2. One-to-One Comparison

The LU receives the inverted indexes $\mathbf{E}_1$ and $\mathbf{E}_2$ from the two DOs. As we outline in Algorithm 4, the LU first initialises the inverted index of compared results, $\mathbf{M}$, in line 1. The LU finds common identifiers $\mathbf{E}_C$ between $\mathbf{E}_1$ and $\mathbf{E}_2$ in line 2 and loops over them in line 3. For each $hid \in \mathbf{E}_C$, the LU extracts the corresponding identifiers (eid_1 and eid_2), ciphertexts (E_1 and E_2), and the lists of hash values ($\mathbf{H}_1$ and $\mathbf{H}_2$) from $\mathbf{E}_1$ and $\mathbf{E}_2$ in lines 4 and 5.

In lines 6 to 9, the LU finds common hash values between $\mathbf{H}_1$ and $\mathbf{H}_2$ based on Eq. (3a). If there are any common hashes, the LU selects the ciphertext E_2 as the minimum value *min*. If there is no common hash, the LU selects the ciphertext E_1 as the minimum value *min*. We show the example of the comparison results of unencrypted 1- and 0-encodings in Fig. 3(b) in the white box. The LU generates a pair of identifiers (eid_1, eid_2), and inserts *min* into the inverted index $\mathbf{M}$ using (eid_1, eid_2) as a key in line 10. The LU repeats the steps in lines 3 to 10 until all ciphertexts corresponding to all $hid \in \mathbf{E}_C$ have been compared. Finally, the LU returns $\mathbf{M}$ to the DOs in line 11.

6. Many-to-One Approach

Our PPRL approach based on multiple l_v^s values encrypted into one ciphertext uses the packing method proposed by Cheon et al. [13] and compares the ciphertexts using Eq. (4) to Eq. (6) [14], as we described in Section 3.2.

6.1. Many-to-One Encryption

The packing method allows multiple decimal numbers to be encrypted into a single ciphertext [13]. Therefore, we can encrypt the l_v^s values without converting them to integer numbers as needed in our one-to-one approach. Algorithm 5 outlines the many-to-one encryption by a DO. In lines 1 and 2, the DO initialises the list of identifiers, **eid**, and list of l_v^s values, $\mathbf{l}_v^s$, respectively. The DO then sorts $\mathbf{T}$ based on reference values (keys of $\mathbf{T}$) in alphabetical order, resulting in the inverted index $\mathbf{T}'$ in line 3. This sorting step ensures that the DOs will insert their l_v^s values that correspond to the same sub-strings in the same order, and therefore the LU will compare ciphertexts and return correct results to the DOs.

In line 4, the DO loops over reference value r and (sid, l_v^s) in $\mathbf{T}'$. The DO then encrypts the *sid* into a ciphertext *eid* by using the function *encrypt*() in line 5. The DO inserts the *eid*

Algorithm 6: ***Many-to-One comparison by the LU***

Input:
- $\mathbf{eid}_1, E_1$: List of encrypted identifiers and Ciphertext from the first DO
- $\mathbf{eid}_2, E_2$: List of encrypted identifiers and Ciphertext from the second DO

Output:
- **pid**: List of pairs of identifiers *min*: Compared result

1: **pid** ← [ ] // Initialise a list of identifier pairs
2: $n = |\mathbf{eid_1}| = |\mathbf{eid_2}|$ // Initialise a length of identifiers where $|\mathbf{eid1}| = |\mathbf{T}|$
3: **for** $i = 1$ to n **do**: // Loop over index in the list
4: $eid_1 \leftarrow \mathbf{eid}_1[i]$ // Get an encrypted identifier from $\mathbf{eid}_1$
5: $eid_2 \leftarrow \mathbf{eid}_2[i]$ // Get an encrypted identifier from $\mathbf{eid}_2$
6: $pid \leftarrow (eid_1, eid_2)$ // Generate pair of identifiers
7: **pid**.$append(pid)$ // Add the pair of identifiers to the list
8: $min \leftarrow compareCip(E_1, E_2)$ // Compare ciphertexts
9: return **pid**, *min*

into the list **eid** and inserts the l_v^s into the list $\mathbf{l}_v^s$ in lines 6 and 7, respectively. The DO repeats the steps in lines 4 to 7 until $|\mathbf{eid}| = |\mathbf{T}'|$. An example of generating $\mathbf{l}_v^s$ is shown in Fig. 3(c), where these $\mathbf{l}_v^s$ lists ($\mathbf{l}_x^s$ and $\mathbf{l}_y^s$) are generated from $\mathbf{D}_A$ and $\mathbf{D}_B$ in Fig. 3(a). In line 8, the DO permutes the lists **eid** and $\mathbf{l}_v^s$ by using the agreed secret salt value s_v as a seed to hide the original positions of values in these lists, while an *eid* and its corresponding l_v^s in these lists still refer to the same sub-string. The DO encrypts the permuted list $\mathbf{l}_v^s$ into a ciphertext E by using the function *packEncrypt*() [13] in line 9. Finally, in line 10, the DO sends the list **eid** and the ciphertext E to the LU for comparison.

6.2. Many-to-One Comparison

The LU receives the lists of identifiers ($\mathbf{eid}_1$ and $\mathbf{eid}_2$) and ciphertexts (E_1 and E_2) from the two DOs. As we outline in Algorithm 6, the LU first initialises the list of pairs of identifiers **pid** in line 1 and the length of a list of identifiers n in line 2. The LU then loops over the indexes in line 3. In lines 4 and 5, for each index i in the lists $\mathbf{eid}_1$ and $\mathbf{eid}_2$, the LU extracts the encrypted identifiers eid_1 and eid_2, respectively. The LU then generates a pair of identifiers, *pid*, in line 6, and inserts this pair into the list **pid** in line 7. The LU repeats the steps in lines 3 to 7 until $|\mathbf{pid}| = n$. In line 8, the LU uses the function *compareCip*() to find *min* [14], where this function conducts a comparison following Eq. (4) to Eq. (6). Finally, the LU returns **pid** and *min* to the DOs in line 8. We show an example of the comparison results of unencrypted values in the white box in Fig. 3(c).

7. Longest Common Sub-string Selection

In the last step of our approaches, each DO selects the (normalised) length of the LCS, lcs, based on the results they received from the LU. If the DOs follow the one-to-one approach, they decrypt each ciphertext in $\mathbf{M}$ to an integer n_v^s, and then convert it to l_v^s by calculating $l_v^s = n_v^s / 10^d$.

If the DOs follow the many-to-one approach, they decrypt a ciphertext E to multiple l_v^s values. Each DO then extracts each pair of identifiers and selects the string identifier from its database that corresponds to its *eid* in the pair. Once the DO has all l_v^s values for each of its strings, the DO finds the highest l_v^s value for the string which is the length of the LCS, lcs, between the pair of strings. Finally, the DO only keeps the string pairs that have an $lcs \geq s_t$ and their corresponding length of (common) sub-string of at least m as matches.

8. Analysis

In this section, we analyse our proposed approaches in terms of linkage quality, time complexity, and privacy.

8.1. Linkage Quality Analysis

The global database, $\mathbf{G}$, is the major factor to determine the linkage quality of our approaches. This is because the sub-strings in $\mathbf{G}$ must be in the same domain as the databases $\mathbf{D}_A$ and $\mathbf{D}_B$ to ensure common sub-strings of the two DOs will be compared. If there are sub-strings in $\mathbf{D}_A$ and $\mathbf{D}_B$ that are not in $\mathbf{G}$, then some common sub-strings of the DOs will not be compared because they have no reference values. Therefore, this can result in more false non-matches (missed matches) which leads to lower recall.

The sorting of reference values and the same permutation of the lists **eid** and $\mathbf{l}_v^s$ are important in our many-to-one approach because they ensure that the DOs will insert their l_v^s values in the same order. Without using the sorted list of references and the same permutations, the two l_v^s values that correspond to different sub-strings will be compared. As a result, the linkage process can result in false matches and false non-matches.

Our approaches use homomorphic encryption [13, 49] for encrypting the l_v^s and n_v^s values. However, by using homomorphic encryption [13, 49], there can be errors (noise) which lead to lower linkage quality. These errors can be caused by noise addition to improve security, the encryption/decryption process, more addition or multiplication operations needed to be conducted over ciphertexts, or even rounding of numerical values before conducting the encryption process [13]. When using the comparison method suggested by Cheon et al. [14], errors can occur when conducting multiple arithmetic operations over ciphertexts, where these errors often occur when two ciphertexts are the encryption of very close (or the same) numerical values [14]. Therefore, in our approaches, if errors are not between 0.0 and 1.0 (errors are large or negative numbers [50]), the DOs can select their l_v^s values as the minimum values because such errors imply that the values of the two DOs are very close or the same. For errors that are between 0.0 and 1.0, which can also occur in homomorphic encryption, such errors can lead to false matches and false non-matches.

8.2. Complexity Analysis

To generate blocks $\mathbf{B}$, each DO requires a complexity of $O(|\mathbf{D}|)$ for extracting string values and generating the list $\mathbf{q}$ for each string, where $|\mathbf{D}|$ is the number of strings in $\mathbf{D}$. For each $\mathbf{q}$, the DO requires $O(|\mathbf{q}|)$ time complexity for generating the list **bkv**, and for each **bkv** the DO then requires $O(|\mathbf{bkv}|)$ time

complexity to calculate the l_v^s values that correspond to the sub-strings of all *bkv* in the list **bkv**. Therefore, overall the block generation process requires a complexity of $O(|\mathbf{D}| \times |\mathbf{q}| \times |\mathbf{bkv}|)$, where we assume $|\mathbf{q}|$ is the average length of the q-gram lists of strings in **D**, and $|\mathbf{bkv}|$ is the average number of *bkv* of **D**.

To generate the list of references **R**, each DO requires $O(|\mathbf{G}|)$ complexity for extracting string values from **G** and generating a list **q**. Similar to the block generation process, each DO requires $O(|\mathbf{q}|)$ time complexity to generate reference values and insert them into **R**. Overall, the DO requires a maximum $O(|\mathbf{G}| \times |\mathbf{q}|)$ complexity for generating **R**. To generate the inverted index, **T**, the DO requires $O(|\mathbf{R}|)$ complexity to loop over **R** and $O(|\mathbf{B}|)$ for checking if $r \in \mathbf{R}$ exists in **B**. If $r \in \mathbf{B}$, the DO requires $O(|\mathbf{B}[r]|)$ to find the maximum l_v^s in the block $\mathbf{B}[r]$. The DO inserts values into **T** for which it requires a time complexity of $O(|\mathbf{T}|)$ to find the number n_r of actual l_v^s that needs to be replaced with random l_v^s, and requires $O(n_r)$ complexity for the replacing value step. Overall, the DO requires $O(|\mathbf{R}| \times |\mathbf{B}| \times n + |\mathbf{T}| + n_r)$, where we assume n is the average number of elements of all $\mathbf{B}[r]$ in **B** to generate **T**, and ensure actual and random l_v^s are equal.

In the one-to-one encryption process, each DO first loops over **T**, and therefore it requires $O(|\mathbf{T}|)$ complexity. The DO then generates 1- or 0-encodings and hash encodes the generated encodings into hash values which requires $O(|\mathbf{e}|)$, where **e** is the set of 1- or 0-encodings. Therefore, for the one-to-one encryption process, each DO requires a time complexity of $O(|\mathbf{T}| \times |\mathbf{e}|)$, where we assume $|\mathbf{e}|$ is the average number of elements of all **e**. In the one-to-one comparison step, the LU requires $O(|\mathbf{E}_C|)$ complexity for finding the common identifiers between $\mathbf{E}_1$ and $\mathbf{E}_2$, where $|\mathbf{E}_C| = |\mathbf{T}|$ because the two DOs generated their inverted indexes based on **T**. The LU then loops over $hid \in \mathbf{E}_C$ which again requires $O(|\mathbf{E}_C|)$ complexity. For a pair of lists of hash values under each key *hid*, the LU requires a time complexity of $O(|\mathbf{H}|)$ to find common hash values, where we assume $|\mathbf{H}| = |\mathbf{H}_1| = |\mathbf{H}_2|$. Overall, the LU requires a time complexity of $O(|\mathbf{E}_C| + |\mathbf{E}_C| \times |\mathbf{H}|)$.

In the many-to-one encryption process, each DO first sorts **T** based on keys, resulting in the sorted inverted index $\mathbf{T}'$. In this step, the DO requires the worst time complexity of $O(|\mathbf{T}|^2)$, while the best time complexity is $O(|\mathbf{T}|\ log\ |\mathbf{T}|)$. The DO requires $O(|\mathbf{T}'|)$ complexity to loop over $\mathbf{T}'$ to generate the lists **eid** and $\mathbf{l}_v^s$. The DO then requires $O(|\mathbf{eid}|)$ and $O(|\mathbf{l}_v^s|)$ for permuting these lists, where $|\mathbf{eid}| = |\mathbf{l}_v^s|$. Overall the DO has the worst-case complexity of $O(|\mathbf{T}|^2 + |\mathbf{T}'| + |\mathbf{eid}| + |\mathbf{l}_v^s|)$. In the many-to-one comparison step, the LU receives $\mathbf{eid}_1$ and $\mathbf{eid}_2$, and ciphertexts E_1 and E_2 from the two DOs. Overall, the LU requires a complexity of $O(|\mathbf{eid}|)$ which is needed for generating pairs of identifiers, where $|\mathbf{eid}| = |\mathbf{eid}_1| = |\mathbf{eid}_2| = |\mathbf{T}|$. The LU only requires $O(1)$ for conducting a comparison between ciphertexts E_1 and E_2.

8.3. Privacy Analysis

We assume the DOs do not collude with the LU and assume the LU to be a semi-honest adversary who is interested in learning the string values of DOs. The DOs first agree on parameter settings which allows them to learn the parameters that are used in a protocol, but they cannot learn any sensitive information about the strings in each other's databases. The DOs individually conduct blocking of their databases, and therefore they cannot learn any information from each other's databases in this step. The DOs then generate their lists **R** based on the agreed global database **G**. This allows the DOs to know the reference values in **R**.

As we use the list of references **R** and the l_v^s (both actual and random) values, no sensitive information will be revealed to the LU. We assume the worst-case where the LU uses any key attacks, such as the attack proposed by Li and Micciancio [51] which successfully identifies the private key used in CKKS [13] (where we use their packing method in our many-to-one approach). Even though the security of the homomorphic encryption scheme we used in our many-to-one approach can be compromised with such attacks, it is still uncertain for the LU to correctly analyse the frequencies of sub-strings and re-identify the string values of the DOs. This is because (1) each l_v^s can exist (the DO sends its actual l_v^s) or not (the DO sends random l_v^s) to the LU, (2) the LU does not know the lengths of the DO's strings, and therefore it cannot calculate and learn the lengths of sub-strings of the DO, and (3) the LU does not know the sub-string values and lengths of sub-strings or lengths of strings of the DO, and therefore it is difficult to analyse the original string values.

In the one-to-one approach, the LU receives inverted indexes from the DOs. The LU then finds common identifiers between these inverted indexes which allow the LU to learn the length of **R** ($|\mathbf{R}| = |\mathbf{T}|$), but it does not allow the LU to learn any sensitive information of the DOs' databases because all identifiers are in common as the DOs generated them based on the same **R** (keys in **T**). The LU learns which lists of hashes have common hash values, but it cannot learn the original values encoded in them. Furthermore, the DOs use the function HMAC(), where the reference values are used as secret values, and therefore it prevents the LU from conducting dictionary attacks successfully. The LU can count the number of lists of hashes, and extract references from **G** and then conduct a comparison with the ciphertext it received from the DOs. However, these do not allow it to conduct any frequency analysis correctly because each inverted index of a DO contains equal numbers of actual and random l_v^s.

For the many-to-one approach, the LU receives the lists $\mathbf{eid}_1$ and $\mathbf{eid}_2$, and ciphertexts E_1 and E_2 from the two DOs. Similar to the one-to-one approach, although the LU can count the number of identifiers, which equals $|\mathbf{R}|$, the LU cannot learn the number of sub-strings (that corresponded to the actual l_v^s values) of the DOs because the LU does not know which *eid* is the identifier of an actual or a random l_v^s.

The LU returns the compared results to the DOs. Each DO decrypts the ciphertexts and selects the normalised lengths of the LCS, *lcs*, of string pairs. The DO can learn the length of the LCS between a string in its database and the other database, but it cannot learn the position where the LCS occurs. The DO can count the number of its common sub-strings by excluding a number of its random l_v^s values. However, the DO cannot learn the frequencies of sub-strings in the other database be-

cause the DO does not know which sub-strings of the other DO correspond to the actual or random l_v^s. The DO can learn some string lengths of the other DO from the compared results returned from the LU. As a DO knows the reference value that the l_v^s corresponds to, it can learn the length of the sub-string, l_s, of the other DO. If the l_v^s returned from the LU is the number of the other DO, then the DO can calculate the length of the string of the other DO as l_s/l_v^s.

For example, let us assume the DO_A knows that the common sub-string is "*ary mille*" with $l_s = 8$ and the l_v^s returned from the LU is 0.8 where this $l_v^s = 0.8$ is the l_v^s from DO_B while the DO_A has $l_v^s = 1.0$. DO_A calculates $8/0.8 = 10$. Therefore, DO_A learns that the length of the string of the other DO (DO_B) is 10 and this string contains the sub-string "*ary mille*". DO_A can use any publicly available database and the calculated number as described in the example above to analyse the possible string of DO_B. However, it would be time consuming to do this analysis because there are many words or sequences of numbers that contain the same sub-strings.

The list **R** and random l_v^s with equal frequencies to the actual l_v^s in the range $[t \ldots s_t]$ also make it more difficult for each DO to correctly re-identify the original sub-strings and strings in the database of the other DO. For example, assume DO_A and DO_B agreed on the similarity threshold $s_t = 0.8$. Let us assume there are two common reference values $r_1 = maarryymmiil$ and $r_2 = ryymmiillllleer$. We assume DO_A has the sub-strings corresponding to both r_1 and r_2, where its sub-string corresponding to r_1 is with $l_v^s = 0.7$ but DO_A replaces this l_v^s with the random $l_v^s = 0.75$, and DO_A has the sub-string corresponding to r_2 with $l_v^s = 0.8$. We assume DO_B does not have any of the sub-string that correspond to these references then it adds random $l_v^s = 0.7$ for r_1 and $l_v^s = 0.75$ for r_2. The LU conducts the comparison and returns $l_v^s = 0.7$ for r_1 and $l_v^s = 0.75$ for r_2 to the DOs. DO_B cannot learn if the DO_A does or does not have a sub-string that corresponds to r_1 and r_2 in the database because it can be:

1. DO_A does not have both r_1 and r_2, therefore, it adds random l_v^s values.
2. DO_A has both r_1 and r_2 but their corresponding l_v^s values are higher than the values of DO_B, therefore, the LU returns the l_v^s values of DO_B.
3. DO_A has either r_1 or r_2, while one of their corresponding l_v^s values is random and one is higher than the value of DO_B.
4. DO_A has both r_1 and r_2 where their corresponding l_v^s values equal the values of DO_B.
5. DO_A has both r_1 and r_2 but it replaces one of the l_v^s values with a random l_v^s value while the other l_v^s value equals the value of DO_B.
6. DO_A has both r_1 and r_2 but it replaces one of the l_v^s values with a random l_v^s value while the other l_v^s value is higher than the value of DO_B.

9. Experimental Evaluation

We evaluated the linkage quality, time complexity, and privacy of our approaches compared to Bloom filter (BF) encoding [5], the shifted hash encoded q-gram [11] (named ShiftedHash), bit array based approach [11] (named BitArray), and Damgård-Geisler-Krøigaard homomorphic encryption as proposed by Essex [30] (named DGK). We compare our approaches with these baselines because BF encoding is considered as a standard PPRL technique, while ShiftedHash and BitArray provide accurate linkage results, and DGK is a recent homomorphic encryption based string matching technique.

9.1. Datasets and Parameter Settings

We employed datasets that are commonly used in PPRL from two data sources [6, 11, 52]. First, we used 100K, 500K, and 1M real-world string records from the North Carolina Voter Registration[1] (NCVR) [6, 11, 52], where K is 1,000 and M is a million. We extracted these datasets from the snapshot of 2011, 2016, and 2019, where we used the snapshot of 2011 and 2019 as the first and second datasets in a pair, and used the snapshot of 2016 as the global dataset. Second, we used datasets from the European census database[2] (Euro) [6, 52]. We used 25,343 records of Census data which is a fictional dataset that represents some observations from a decennial Census as the first dataset. We used 24,614 records of Customer Information System (CIS) data which is a fictional observation from a CIS that combines administrative data from the tax and benefit systems as the second dataset. For the global dataset, we used 26,625 records of Personal data which is the data underlying the Census and CIS data. For each of the NCVR and Euro datasets, we extracted attribute first names (FN), first and last names (FN and LN), and first, last, and street address names (FN, LN, and SA) to evaluate our approaches and the baselines.

To be comparable, we follow the parameter settings of the baselines [11] where we generated q-grams using $q = 3$ and used $m = q$ for all datasets. We set the threshold $t = 0.7$ for generating blocking key and reference values and used similarity thresholds $s_t = [0.8, 0.9, 1.0]$ for classifying a string pair as a match. In the one-to-one approach, we set $d = 2$ for converting l_v^s to n_v^s. We used the one-way hash function $\mathcal{H}() = SHA256$ [48] for the HMAC() function to generate hashes of 1- and 0-encodings [12], where we let the first DO to generate 1-encodings and the second DO to generate 0-encodings. We then used the Paillier [49] cryptosystem to generate a ciphertext of each n_v^s. In the many-to-one approach, we used the secret salt value $s_v = 45$ for permuting the lists of identifiers and l_v^s values.

For the baselines, we employed the same parameter settings as our approaches. For BF encoding [5], we used a BF length of $l = 1{,}000$ bits, the optimal number of hash functions [53] (calculated as $(l/n) \times ln(2)$, where l is a BF length and n is the number of elements being encoded [54]) and random hashing to improve the degree of privacy as suggested by Schnell and Borgs [55], and set the individual secret salt value $s_A = 65$ and $s_B = 56$ for generating random bit arrays for the two DOs in the BitArray approach [11]. For the DGK approach [30], we used keys of size 1,024 bits as used in the original approach, and we adapted their approach for a three-party protocol rather

[1] http://dl.ncsbe.gov/

[2] https://cros-legacy.ec.europa.eu/content/job-training_en

Table 2: Precision and recall for different datasets and approaches. DGK provides the worst precision or recall results, where the actual results of 0 are around 0.005. ShiftedHash and BitArray approaches provide the best results, where all precision and recall are 1.0.

Dataset	Approach	NCVR 100K						NCVR 500K						NCVR 1M						Euro					
		$s_t = 0.8$		$s_t = 0.9$		$s_t = 1.0$		$s_t = 0.8$		$s_t = 0.9$		$s_t = 1.0$		$s_t = 0.8$		$s_t = 0.9$		$s_t = 1.0$		$s_t = 0.8$		$s_t = 0.9$		$s_t = 1.0$	
		prec	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*	*prec*	*reca*
FN	One-to-One	1.0	1.0	1.0	0.95	1.0	0.95	1.0	1.0	1.0	0.95	1.0	0.95	1.0	1.0	1.0	0.95	1.0	0.95	1.0	1.0	1.0	0.98	1.0	0.98
	Many-to-One	1.0	1.0	1.0	0.96	1.0	0.96	1.0	1.0	1.0	0.95	1.0	0.95	1.0	1.0	1.0	0.95	1.0	0.95	1.0	1.0	1.0	0.98	1.0	0.98
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BF	0.56	1.0	0.81	1.0	1.0	1.0	0.46	1.0	0.73	1.0	1.0	1.0	0.42	1.0	0.69	1.0	1.0	1.0	0.68	1.0	0.89	1.0	1.0	1.0
	DGK	0.39	1.0	0.29	0.58	0.31	0.56	0.3	1.0	0.2	0.56	0.21	0.52	0.27	1.0	0.17	0.55	0.18	0.5	0.47	1.0	0.40	0.73	0.41	1.0
FN and LN	One-to-One	1.0	1.0	1.0	1.0	1.0	0.99	1.0	1.0	1.0	0.99	1.0	0.99	1.0	1.0	1.0	0.99	1.0	0.99	1.0	1.0	1.0	0.99	1.0	0.99
	Many-to-One	1.0	1.0	1.0	1.0	1.0	0.99	1.0	1.0	1.0	0.99	1.0	0.99	1.0	1.0	1.0	0.99	1.0	0.98	1.0	1.0	1.0	0.99	1.0	0.99
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BF	0.53	1.0	0.95	1.0	1.0	1.0	0.44	1.0	0.88	1.0	1.0	1.0	0.62	1.0	0.89	1.0	1.0	1.0	0.47	1.0	0.86	1.0	1.0	1.0
	DGK	0.43	1.0	0.17	0	0.83	0	0.36	1.0	0.21	0	0.7	0	0.53	1.0	0.28	0	0.75	0	0.34	1.0	0.41	0.02	0.72	0.01
FN, LN, and SA	One-to-One	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Many-to-One	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BF	0.62	1.0	0.91	1.0	1.0	1.0	0.64	1.0	0.92	1.0	1.0	1.0	0.64	1.0	0.92	1.0	1.0	1.0	0.53	1.0	0.75	1.0	1.0	1.0
	DGK	0.61	1.0	0	0	0	0	0.63	1.0	0	0	0	0	0.62	1.0	0	0	0	0	0.51	1.0	0	0	0	0

Table 3: F-measure (F^*) for different datasets and approaches. DGK provides the worst results, where the lowest F^* is 0.01. ShiftedHash and BitArray approaches provide the best results which all are 1.0.

Dataset	Approach	NCVR 100K			NCVR 500K			NCVR 1M			Euro		
		$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$	$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$	$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$	$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$
FN	One-to-One	1.0	0.97	0.97	1.0	0.97	0.97	1.0	0.97	0.97	1.0	0.99	0.99
	Many-to-One	1.0	0.98	0.98	1.0	0.97	0.97	1.0	0.97	0.97	1.0	0.99	0.99
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BF	0.72	0.90	1.0	0.63	0.84	1.0	0.60	0.82	1.0	0.81	0.94	1.0
	DGK	0.56	0.39	0.40	0.46	0.30	0.30	0.43	0.26	0.26	0.64	0.52	0.58
FN and LN	One-to-One	1.0	1.0	0.99	1.0	0.99	0.99	1.0	0.99	0.99	1.0	0.99	0.99
	Many-to-One	1.0	1.0	0.99	1.0	0.99	0.99	1.0	0.99	0.99	1.0	0.99	0.99
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BF	0.69	0.97	1.0	0.61	0.94	1.0	0.77	0.94	1.0	0.64	0.92	1.0
	DGK	0.60	0.01	0.01	0.53	0.01	0.01	0.69	0.01	0.01	0.51	0.04	0.02
FN, LN, and SA	One-to-One	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	Many-to-One	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	ShiftedHash	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BitArray	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
	BF	0.77	0.95	1.0	0.78	0.96	1.0	0.78	0.96	1.0	0.69	0.86	1.0
	DGK	0.76	0.01	0.01	0.77	0.01	0.01	0.77	0.01	0.01	0.68	0	0

than a two-party protocol as proposed by Essex [30]. This is to make it comparable with our approaches and the other three baselines.

We implemented our approaches using Python and ran experiments on a server with 2.4 GHz CPUs running Ubuntu 18.04.

9.2. Linkage Quality Results

As we described in Section 8.1, ciphertexts can contain errors (noise). Our many-to-one approach results in errors ranging from 0 to 0.92% for NCVR datasets and 0 to 0.15% for Euro datasets. These errors are calculated by checking the decrypted lcs if it is a large or negative number. For the linkage quality evaluation, we used precision, recall, and F-measure [1, 56]. We also evaluate the percentage of linkage quality improvement of our approaches compared to the BF. To facilitate fair comparison, we compared the normalised length of the LCS of plaintext string pairs, $\mathcal{L}$, calculated using Eq. (7), with the selected normalised length of the LCS, lcs, of the corresponding encrypted l_v^s based on Eq. (9) for our approaches.

For the ShiftedHash and BitArray approaches, we compared the $\mathcal{L}$ of plaintext string pairs with their corresponding lcs calculated based on the calculations in [11]. For BF encoding [5] and DGK [30], we compared the Dice-coefficient of q-gram sets and of their corresponding BFs and DGK encryptions. We calculated $\mathcal{L}$ and Dice-coefficient sim_D^q between plaintext values of a pair, and use them as the true similarity. For a given s_t, we classified the corresponding encrypted pair with its lcs (sim_D^b for BF encoding and DGK encryptions) as:

- A true positive if $\mathcal{L} \geq s_t$ and $lcs \geq s_t$.
- A false positive if $\mathcal{L} < s_t$ and $lcs \geq s_t$.
- A true negative if $\mathcal{L} < s_t$ and $lcs < s_t$.
- A false negative if $\mathcal{L} \geq s_t$ and $lcs < s_t$.

We then used these classified values to calculate precision, recall, and F-measure (F^*), where the F^* is then used to calculate

Table 4: Percentage of improvement of our approaches over BF approach, where the positive values represent our approaches have higher linkage quality than the BF while the negative values represent the BF has higher linkage quality than our approaches.

Dataset	Approach	NCVR 100K			NCVR 500K			NCVR 1M			Euro		
		$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$	$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$	$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$	$s_t = 0.8$	$s_t = 0.9$	$s_t = 1.0$
FN	One-to-One VS BF	38.89%	7.78%	-3.0%	58.73%	15.48%	-3.0%	66.67%	18.29%	-3.0%	23.46%	5.32%	-1.0%
	Many-to-One VS BF	38.89%	8.89%	-2.0%	58.73%	15.48	-3.0%	66.67%	18.29%	-3.0%	23.46%	5.32%	-1.0%
FN and LN	One-to-One VS BF	44.92%	3.09%	-1.0%	63.93%	5.32%	-1.0%	29.87%	5.32%	-1.0%	56.25%	7.61%	-1.0%
	Many-to-One VS BF	44.92%	3.09%	-1.0%	63.93%	5.32%	-1.0%	29.87%	5.32%	-1.0%	56.25%	7.61%	-1.0%
FN, LN, and SA	One-to-One VS BF	29.87%	5.26%	0.0%	28.21%	4.17%	0.0%	28.21%	4.17%	0.0%	44.93%	16.28	0.0%
	Many-to-One VS BF	29.87%	5.26%	0.0%	28.21%	4.17%	0.0%	28.21%	4.17%	0.0%	44.93%	16.28	0.0%

Table 5: Numbers of references / blocks and comparisons, where the four baselines have the same numbers of blocks and comparisons because they are generated from the same datasets and use the same calculation (following Eq. (10)) for generating *bkv*. Our one-to-one and many-to-one approaches have the same number of reference values, while the numbers of comparisons for our many-to-one approach are all 1. Ref. and Comp. stand for reference and comparison, respectively.

Dataset	NCVR 100K				NCVR 500K				NCVR 1M				Euro			
	One-to-One		Baselines		One-to-One		Baselines		One-to-One		Baselines		One-to-One		Baselines	
	Ref.	Comp.	Block	Comp.	Ref.	Comp.	Block	Comp.	Ref.	Comp.	Block	Comp.	Ref.	Comp.	Block	Comp.
FN	32K	32K	32K	61K	91K	91K	91K	258K	140K	140K	140K	473K	6.9K	6.9K	6.8K	4.8K
FN and LN	927K	927K	927K	701K	4M	4M	4M	4M	7M	7M	7M	10M	175K	175K	171K	53K
FN, LN, and SA	4M	4M	4M	2M	19M	19M	19M	8M	39M	39M	39M	16M	175K	175K	781K	48K

the percentage of improvement comparing our approaches and the BF as the BF is considered as a standard PPRL.

We provide the precision and recall values for different datasets in Table 2, while providing the F^* in Table 3. The percentage of improvement of our approaches over the BF is shown in Table 4. We limited the number of comparisons for baselines to 1 million sub-string pairs because the total number of comparisons is very large, as we show in Table 5, and because the baselines use high runtimes for conducting comparisons, especially the BitArray and DGK approaches, as we illustrate in Fig. 4 and discuss in the next section.

In our approaches, errors in ciphertexts can cause lower recall, and the common sub-strings of the two DOs with no reference values can also cause lower recall because those sub-strings were not compared. However, as can be seen in Tables 2 and 3, our approaches provide high linkage quality because there are a small number of errors with ranging from 0 to 0.92% for NCVR datasets and 0 to 0.15% for Euro datasets. Our approaches provide precisions of 1.0 for the evaluations on both NCVR and Euro datasets while providing recall ranging from 0.95 to 1.0 for NCVR datasets and ranging from 0.98 to 1.0 for Euro datasets. Our approaches provide the F^* ranging from 0.97 to 1.0 for the NCVR datasets and provide the F^* ranging from 0.99 to 1.0 for the Euro datasets.

Overall, the ShiftedHash and BitArray approaches provide the highest precision and recall values. This is because these approaches encode the strings that are to be compared based on the lists of q-grams (sub-strings) generated from these strings. This allows these approaches to use positional information of the q-grams in their comparison processes. As a result, these approaches can provide high linkage quality.

The DGK approach [30] provides the lowest precision, recall, and F^* values. This is because the Dice-coefficient of the encryptions is calculated based on the cardinality, while some of them are the encryptions of not common q-grams between datasets, but are common in the two lists of all possible q-grams. The BF encoding [5] provides low precision values and thus low F^* because it results in many false positives which are likely caused by hash collisions in BFs [2, 5].

Given BF encoding is considered as a standard PPRL technique, we compared the percentages of improvement of our approaches over BF encoding by calculating $((F^* - F^*_{BF})/F^*_{BF}) \times 100$, where F^* refers to the F-measure results obtained with our method and F^* is the F-measure of BF encoding. As can be seen in Table 4, overall our approaches provide higher linkage quality compared to BF encoding, where on a few occasions our approaches provide lower linkage quality than BFs (shown as negative percentage values). In other words, in most cases our PPRL approaches improve the linkage quality and provide higher linkage quality over a commonly used standard BF encoding PPRL technique.

9.3. Time Complexity Results

Table 5 shows the number of references, the number of blocks, and the number of comparisons of our approaches and the baselines. The baselines have the same numbers of blocks and comparisons because they are generated from the same datasets and use the same calculation (following Eq. (10)) for generating *bkv*.

As expected, our two approaches have the same number of references. Our one-to-one approach has the number of comparisons equal to the number of references. This is because the DOs select the maximum l^s_v value that corresponds to each reference value and then encrypt all selected l^s_v values before sending them to the LU. For our many-to-one approach, the number of comparisons is 1 because all l^s_v values of each DO are inserted into one list before being encrypted into a single ciphertext. The number of comparisons of the baselines depends upon the common blocks (*bkv*) between the databases to be linked and the number of strings in these common blocks. Therefore, the number of comparisons can be higher or lower than the number of blocks.

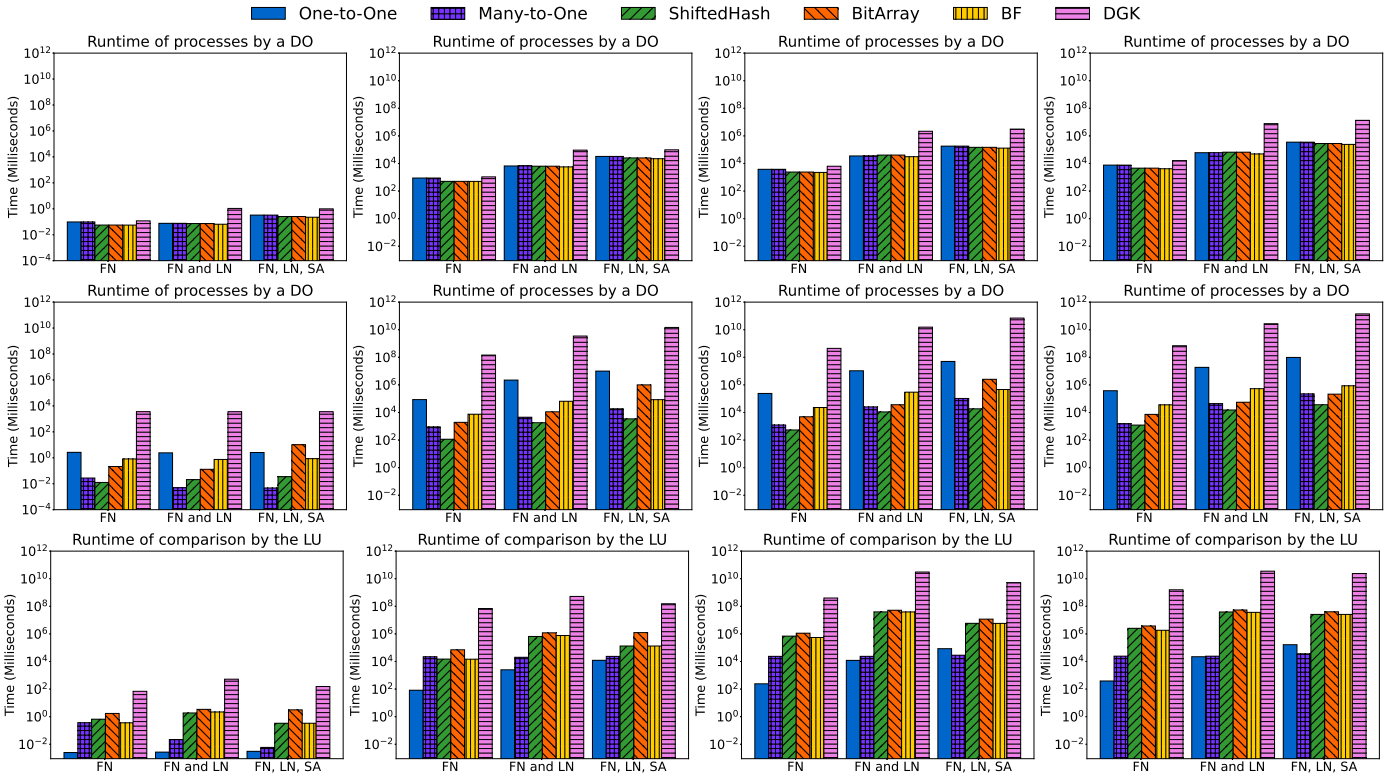

Figure 4: Runtimes for the preparation (top), the encryption (middle), and comparison (bottom) processes by a DO and the LU of our approaches and the baselines. Runtimes per sub-string, 100K, 500K, and 1M are shown from left to right.

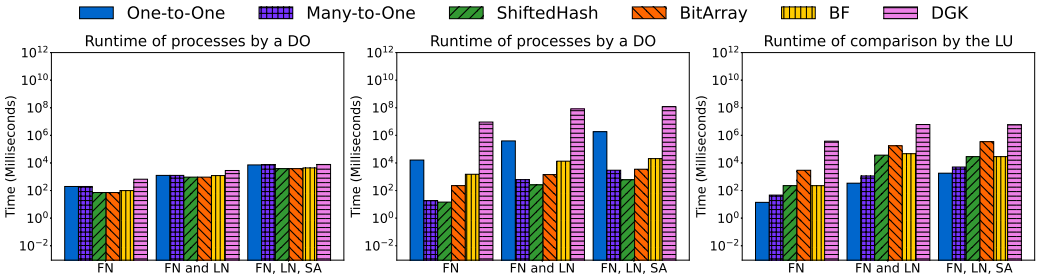

Figure 5: Runtimes for the preparation (left), encryption (middle), and comparison (right) processes by a DO and the LU of our approaches and the baselines on the Euro dataset.

We show the runtimes of the processes by a DO and a LU for the NCVR and Euro datasets in Fig. 4 and Fig. 5, respectively. As can be seen, the DGK approach uses the longest runtimes in data preparation processed by a DO for both NCVR and Euro datasets. This is because the DGK approach needs to generate a list of all possible q-grams and then map these q-grams with the q-grams (sub-strings) to be linked. The ShiftedHash approach is the fastest while the DGK is the slowest encryption approach. This is because the ShiftedHash requires only $O(|\mathbf{q}|)$, where $|\mathbf{q}|$ is the length of a list of q-grams, while the DGK approach has longer runtimes because it encrypt each q-gram that occurs in each string to be linked.

Overall, the runtimes for the baselines are depended upon the number of blocks and strings to be encoded in these blocks. The runtimes for our one-to-one approach are depended upon the number of reference values. The runtime required for comparison by our many-to-one approach does not affected by the number of reference value selection because it encrypts multiple values in a single step.

As can be seen from the comparison by the LU in Fig. 4 and Fig. 5, the DGK approach provides the slowest comparison. In contrast to BF encoding, ShiftedHash, and BitArray approaches, our approaches achieved the fastest runtime when comparing string pairs for both NCVR and Euro datasets, where our one-to-one approach is the fastest.

9.4. Privacy Results

As used with previous PPRL approaches, we evaluated privacy by using relative information gain (*RIG*) [57] and disclosure risks (*DR*) [56], where the *RIG* measures the difficulty of inferring the original plaintexts in a database based on the information about encrypted values, and the *DR* refers to the likelihood of correct re-identification of the original plaintexts.

Table 6: Privacy measures for different datasets and approaches. The best and worst results are shown in bold and italic, respectively. DR_{mx}, DR_{mn}, DR_{md}, and DR_{mk} are maximum, mean, median, and marketer disclosure risks, respectively. ShiftedHash [11], BitArray [11], and BF encoding [5] provide the same results.

Dataset	Approach	NCVR 100K					NCVR 500K					NCVR 1M					Euro				
		RIG	DR_{mx}	DR_{mn}	DR_{md}	DR_{mk}	*RIG*	DR_{mx}	DR_{mn}	DR_{md}	DR_{mk}	*RIG*	DR_{mx}	DR_{mn}	DR_{md}	DR_{mk}	*RIG*	DR_{mx}	DR_{mn}	DR_{md}	DR_{mk}
FN	One-to-One	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	0.86	*1.0*	0.06	0.33	**0.0**	0.87	*1.0*	0.06	0.33	**0.0**	0.87	*1.0*	0.05	0.17	**0.0**	0.84	*1.0*	0.08	0.33	0.01
	BitArray	0.86	*1.0*	0.06	0.33	**0.0**	0.87	*1.0*	0.06	0.33	**0.0**	0.87	*1.0*	0.05	0.17	**0.0**	0.84	*1.0*	0.08	0.33	0.01
	BF	0.86	*1.0*	0.06	0.33	**0.0**	0.87	*1.0*	0.06	0.33	**0.0**	0.87	*1.0*	0.05	0.17	**0.0**	0.85	*1.0*	0.08	0.33	0.01
	DGK	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
FN and LN	One-to-One	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	0.82	0.33	**0.0**	0.10	**0.0**	0.84	0.33	**0.0**	0.10	**0.0**	0.85	0.33	**0.0**	0.10	**0.0**	0.81	*1.0*	0.01	0.1	**0.0**
	BitArray	0.82	0.33	**0.0**	0.10	**0.0**	0.84	0.33	**0.0**	0.10	**0.0**	0.85	0.33	**0.0**	0.10	**0.0**	0.81	*1.0*	0.01	0.1	**0.0**
	BF	0.82	0.33	**0.0**	0.10	**0.0**	0.84	0.33	**0.0**	0.10	**0.0**	0.85	0.33	**0.0**	0.10	**0.0**	0.81	*1.0*	0.01	0.1	**0.0**
	DGK	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
FN, LN, and SA	One-to-One	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*	*1.0*
	Many-to-One	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**
	ShiftedHash	0.75	0.07	**0.0**	0.03	**0.0**	0.78	0.10	**0.0**	0.03	**0.0**	0.79	0.10	**0.0**	0.03	**0.0**	0.74	0.1	**0.0**	**0.0**	**0.0**
	BitArray	0.75	0.07	**0.0**	0.03	**0.0**	0.78	0.10	**0.0**	0.03	**0.0**	0.79	0.10	**0.0**	0.03	**0.0**	0.74	0.1	**0.0**	**0.0**	**0.0**
	BF	0.75	0.07	**0.0**	0.03	**0.0**	0.78	0.10	**0.0**	0.03	**0.0**	0.79	0.10	**0.0**	0.03	**0.0**	0.74	0.1	**0.0**	0.03	**0.0**
	DGK	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**	**1.0**	**0.0**	**0.0**	**0.0**	**0.0**

Lower *RIG* and *DR* values indicate an approach could provide stronger privacy protection.

We assume the worst-case scenario where an adversary uses the same parameter settings, global database, and the same database to be encrypted as a DO. We illustrate the results of our approaches compared to the baselines for different datasets in Table 6, where we show the evaluation results of the lists of hashes for our one-to-one approach and the results of ciphertexts for our many-to-one approach.

As can be seen, for both NCVR and Euro datasets our approaches provide the same results where our one-to-one approach provides the weakest privacy protection because of the relationship between a reference value and the list of hash values is a one-to-one relationship. Therefore, due to the direct relationship between the reference values and the list of hash values, it is highly likely that an adversary will be able to infer plaintext values due to less uncertainty. This one-to-one approach provides high values of *RIG* and *DR* because we assume the adversary uses the same databases (global database and the database to be encrypted) and parameter settings as a DO. If the similarity between the attacking database and the plaintext database is low, then the *RIG* and *DR* values will be lower depending upon how many strings in the adversary's database and DO's database are in common. However, it is possible that the adversary might still be able to infer some plaintext values based on common strings.

On the other hand, our many-to-one approach and the DGK [30] approach provide the strongest privacy protection. This is because this approach encrypts multiple values into a single ciphertext, while in the DGK approach the same string value is encrypted into different ciphertexts using homomorphic encryption. The *RIG* value is 1.0 for these approaches because for database **D**, given encryptions **E**, $H(\mathbf{D}|\mathbf{E})$ [56] results in 0.0, and therefore the *RIG* becomes 1.0. However, as can be seen from different *DR* measures of these approaches, they provide 0.0 in all occasions which implies that there is no risk of disclosure for these approaches. In a case where an adversary can correctly guess the private key and is able to decrypt the ciphertext(s) [51], there is still a high uncertainty for the adversary to re-identify the plaintext values as we described in Section 8.3.

As we assume an adversary uses the same parameters and database as a DO, for BF encoding [5], the adversary can generate the same BFs as an encoded database of the DO. For the ShiftedHash [11] approach, although the list of hash values has been shifted, an adversary can rotate the list until it finds the same list of hash values as the list of the DO. Similarly for the BitArray [11] approach, as an adversary uses the same parameters and database as the DO,it knows the q-gram bit array. Therefore, the adversary can find the bit array of a string before it has been padded with random bits.

10. Conclusion

We proposed two encryption based privacy-preserving string matching approaches that allow fast comparison and provide high linkage quality. Our one-to-one approach encrypts each length of a sub-string in a string into a ciphertext, while our many-to-one approach encrypts multiple lengths of sub-strings in strings into a single ciphertext.

Our experimental evaluation has shown that our approaches result in high linkage quality. Our one-to-one approach provides the fastest comparison results, while our many-to-one and the DGK [30] approaches provide the strongest privacy protection. Our evaluation results show that the one-to-one is more suitable where high accuracy is required such as financial service applications. This is because the CKKS as we used in our many-to-one can introduce some errors. However, we recommend using our many-to-one approach for linking large databases that require fast, high linkage quality, and high degrees of privacy such as bioinformatics, healthcare, and crime detection applications. As future work, we aim to improve our approaches to provide more accurate linkage results by reducing errors that occur in ciphertexts. We also aim to improve the time complexity of data preparation and improve the degree of

privacy in the agreement of parameter settings and extend our approaches to scenarios where DOs collude with the LU.

References

[1] P. Christen, Data Matching – Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection, Springer, Heidelberg, 2012.

[2] P. Christen, T. Ranbaduge, R. Schnell, Linking Sensitive Data: Methods and Techniques for Practical Privacy-Preserving Information Sharing, Springer, 2020.

[3] A. Gkoulalas-Divanis, D. Vatsalan, D. Karapiperis, M. Kantarcioglu, Modern privacy-preserving record linkage techniques: An overview, IEEE Transactions on Information Forensics and Security 16 (2021) 4966–4987.

[4] D. Vatsalan, P. Christen, V. Verykios, A taxonomy of privacy-preserving record linkage techniques, Information Systems 38 (6) (2013) 946–969.

[5] R. Schnell, T. Bachteler, R. J., Privacy-preserving record linkage using Bloom filters, BMC Med Inform Decis Mak 9 (1) (2009) 1–11.

[6] A. Vidanage, P. Christen, T. Ranbaduge, R. Schnell, A vulnerability assessment framework for privacy-preserving record linkage, ACM Transactions on Privacy and Security (2023).

[7] A. Vidanage, T. Ranbaduge, P. Christen, R. Schnell, A taxonomy of attacks on privacy-preserving record linkage, Journal of Privacy and Confidentiality 12 (1) (2022).

[8] O. Goldreich, Secure multi-party computation, Tech. rep., Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Israel (2002).

[9] D. Vatsalan, Z. Sehili, P. Christen, E. Rahm, Privacy-preserving record linkage for Big data: Current approaches and research challenges, in: Handbook of Big Data Technologies, Springer, 2017, pp. 851–895.

[10] D. Karapiperis, A. Gkoulalas-Divanis, V. S. Verykios, Efficient record linkage in data streams, in: IEEE International Conference on Big Data, IEEE, 2020, pp. 523–532.

[11] S. Vaiwsri, T. Ranbaduge, P. Christen, Accurate and efficient privacy-preserving string matching, International Journal of Data Science and Analytics (2022) 1–25.

[12] H.-Y. Lin, W.-G. Tzeng, An efficient solution to the millionaires' problem based on homomorphic encryption, in: International Conference on Applied Cryptography and Network Security, Springer, 2005.

[13] J. H. Cheon, A. Kim, M. Kim, Y. Song, Homomorphic encryption for arithmetic of approximate numbers, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2017, pp. 409–437.

[14] J. H. Cheon, D. Kim, D. Kim, Efficient homomorphic comparison methods with optimal complexity, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2020, pp. 221–256.

[15] E. Ukkonen, Approximate string-matching over suffix trees, in: Annual Symposium on Combinatorial Pattern Matching, Springer, 1993, pp. 228–242.

[16] J. Wang, R. Li, A new cluster merging algorithm of suffix tree clustering, Springer US, Boston, MA, 2007, pp. 197–203.

[17] J. Wang, X. Yang, B. Wang, C. Liu, An adaptive approach of approximate substring matching, in: International Conference on Database Systems for Advanced Applications, Springer, 2016, pp. 501–516. doi:10.1007/978-3-319-32025-0_31.

[18] M. Yu, C. Chai, G. Yu, A tree-based indexing approach for diverse textual similarity search, IEEE Access 9 (2020) 8866–8876.

[19] M.-S. Kim, K.-Y. Whang, J.-G. Lee, n-Gram/2L-approximation: a two-level n-gram inverted index structure for approximate string matching, Computer Systems Science and Engineering 22 (6) (2007) 365.

[20] M. S. R. Mahdi, M. M. Al Aziz, N. Mohammed, X. Jiang, Privacy-preserving string search on encrypted genomic data using a generalized suffix tree, Informatics in Medicine Unlocked 23 (2021) 100525.

[21] C. Paar, J. Pelzl, Understanding cryptography: a textbook for students and practitioners, Springer Science & Business Media, 2009.

[22] D. Vatsalan, P. Christen, Privacy-preserving matching of similar patients, Journal of Biomedical Informatics 59 (2016) 285–298.

[23] D. Karapiperis, A. Gkoulalas-Divanis, V. S. Verykios, Distance-aware encoding of numerical values for privacy-preserving record linkage, in: IEEE International Conference on Data Engineering, 2017, pp. 135–138.

[24] N. Wu, D. Vatsalan, S. Verma, M. A. Kaafar, Fairness and cost constrained privacy-aware record linkage, IEEE Transactions on Information Forensics and Security 17 (2022) 2644–2656.

[25] C. Dwork, Differential privacy, International Colloquium on Automata, Languages and Programming (2006) 1–12.

[26] M. Kuzu, M. Kantarcioglu, A. Inan, E. Bertino, E. Durham, B. Malin, Efficient privacy-aware record integration, in: Proceedings of the 16th International Conference on Extending Database Technology, 2013, pp. 167–178.

[27] W. Xue, D. Vatsalan, W. Hu, A. Seneviratne, Sequence data matching and beyond: New privacy-preserving primitives based on bloom filters, IEEE Transactions on Information Forensics and Security 15 (2020) 2973–2987.

[28] Ú. Erlingsson, V. Pihur, A. Korolova, Rappor: Randomized aggregatable privacy-preserving ordinal response, in: Proceedings of the 2014 ACM SIGSAC conference on computer and communications security, 2014, pp. 1054–1067.

[29] S. Yao, Y. Ren, D. Wang, Y. Wang, W. Yin, L. Yuan, Snn-pprl: A secure record matching scheme based on siamese neural network, Journal of Information Security and Applications 76 (2023) 103529.

[30] A. Essex, Secure approximate string matching for privacy-preserving record linkage, IEEE Transactions on Information Forensics and Security 14 (10) (2019) 2623–2632.

[31] I. Damgård, M. Geisler, M. Krøigaard, Efficient and secure comparison for on-line auctions, in: Australasian conference on information security and privacy, Springer, 2007, pp. 416–430.

[32] T. K. Saha, D. Rathee, T. Koshiba, Efficient protocols for private wildcards pattern matching, Journal of Information Security and Applications 55 (2020) 102609.

[33] X. Mullaymeri, A. Karakasidis, Using fuzzy vaults for privacy preserving record linkage., in: DOLAP, 2021, pp. 101–110.

[34] A. Juels, M. Sudan, A fuzzy vault scheme, Designs, Codes and Cryptography 38 (2006) 237–257.

[35] S. Stammler, T. Kussel, P. Schoppmann, F. Stampe, G. Tremper, S. Katzenbeisser, K. Hamacher, M. Lablans, Mainzelliste secureepilinker (mainsel): privacy-preserving record linkage using secure multi-party computation, Bioinformatics 38 (6) (2022) 1657–1668.

[36] P. Contiero, A. Tittarelli, G. Tagliabue, A. Maghini, S. Fabiano, P. Crosignani, R. Tessandori, The epilink record linkage software, Methods of Information in Medicine 44 (01) (2005) 66–71.

[37] D. Demmler, T. Schneider, M. Zohner, Aby-a framework for efficient mixed-protocol secure two-party computation., in: NDSS, 2015.

[38] Y. Nakagawa, S. Ohata, K. Shimizu, Efficient privacy-preserving variable-length substring match for genome sequence, Algorithms for Molecular Biology 17 (1) (2022) 1–22.

[39] P. Ferragina, G. Manzini, Opportunistic data structures with applications, in: Proceedings 41st annual symposium on foundations of computer science, IEEE, 2000, pp. 390–398.

[40] R. Durbin, Efficient haplotype matching and storage using the positional burrows–wheeler transform (pbwt), Bioinformatics 30 (9) (2014) 1266–1272.

[41] R. Hall, S. E. Fienberg, Privacy-preserving record linkage, in: International Conference on Privacy in Statistical Databases, Springer, 2010, pp. 269–283. doi:10.1007/978-3-642-15838-4_24.

[42] Y. Lindell, B. Pinkas, Secure multiparty computation for privacy-preserving data mining (2008).

[43] P. Christen, R. Schnell, D. Vatsalan, T. Ranbaduge, Efficient cryptanalysis of Bloom filters for privacy-preserving record linkage, in: Pacific-Asia Conference on Knowledge Discovery and Data Mining, 2017, pp. 628–640.

[44] P. Christen, T. Ranbaduge, D. Vatsalan, R. Schnell, Precise and fast cryptanalysis for bloom filter based privacy-preserving record linkage, IEEE Transactions on Knowledge and Data Engineering 31 (11) (2018) 2164–2177.

[45] A. Vidanage, T. Ranbaduge, P. Christen, R. Schnell, Efficient pattern mining based cryptanalysis for privacy-preserving record linkage, in: International Conference on Data Engineering, IEEE, 2019, pp. 1698–1701.

[46] A. Karakasidis, V. S. Verykios, Reference table based k-anonymous private blocking, in: Proceedings of the 27th Annual ACM Symposium on Applied Computing, 2012, pp. 859–864.

[47] F. Niedermeyer, S. Steinmetzer, M. Kroll, R. Schnell, Cryptanalysis of basic Bloom filters used for privacy-preserving record linkage, German Record Linkage Center, Working Paper Series, No. WP-GRLC-2014-04 (2014).
[48] B. Schneier, et al., Applied cryptography-protocols, algorithms, and source code in c (1996).
[49] P. Paillier, Public-key cryptosystems based on composite degree residuosity classes, in: International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 1999.
[50] J. H. Cheon, D. Kim, D. Kim, H. H. Lee, K. Lee, Numerical method for comparison on homomorphically encrypted numbers, in: International Conference on the Theory and Application of Cryptology and Information Security, Springer, 2019, pp. 415–445.
[51] B. Li, D. Micciancio, On the security of homomorphic encryption on approximate numbers, in: Annual International Conference on the Theory and Applications of Cryptographic Techniques, Springer, 2021, pp. 648–677.
[52] T. Ranbaduge, R. Schnell, Securing bloom filters for privacy-preserving record linkage, in: Proceedings of the 29th ACM International Conference on Information & Knowledge Management, 2020, pp. 2185–2188.
[53] M. Mitzenmacher, E. Upfal, Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis, CUP, 2005.
[54] M. Mitzenmacher, E. Upfal, Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis, Cambridge university press, 2017.
[55] R. Schnell, C. Borgs, Randomized response and balanced Bloom filters for privacy-preserving record linkage, in: IEEE 16th International Conference on Data Mining Workshops, IEEE, 2016, pp. 218–224.
[56] D. Vatsalan, P. Christen, C. M. O'Keefe, V. S. Verykios, An evaluation framework for privacy-preserving record linkage, Journal of Privacy and Confidentiality 6 (1) (2014).
[57] A. Karakasidis, V. S. Verykios, P. Christen, Fake injection strategies for private phonetic matching, in: Data Privacy Management and Autonomous Spontaneus Security, Springer, 2011, pp. 9–24.