International Journal of Data Science and Analytics manuscript No.
(will be inserted by the editor)

Accurate and Efficient Privacy-Preserving String Matching

Sirintra Vaiwsri, Thilina Ranbaduge, and Peter Christen

Received: date / Accepted: date

Abstract The task of calculating similarities between strings held by different organisations without revealing these strings is an increasingly important problem in areas such as health informatics, national censuses, genomics, and fraud detection. Most existing privacy-preserving string matching approaches are either based on comparing sets of encoded characters allowing only exact matching of encoded strings, or they are aimed at long genomics sequences that have a small alphabet. The set-based privacy-preserving similarity functions that are commonly used to compare name and address strings in the context of privacy-preserving record linkage do not take the positions of sub-strings into account. As a result, two very different strings can potentially be considered as a match leading to wrongly linked records. Furthermore, existing set-based techniques cannot identify the length of the longest common sub-string across two strings. In this paper we propose two new approaches for accurate and efficient privacy-preserving string matching that provide privacy against various attacks. In the first approach we apply hashing based encoding on sub-strings (q-grams) to compare sensitive strings, while in the second approach we generate one bit array from the sub-strings of a string to identify the longest common bit sequences. We evaluate our approaches on several data sets with different types of strings, and validate their privacy, accuracy, and complexity compared to three baseline techniques, showing that they outperform all baselines.

Sirintra Vaiwsri · Peter Christen
School of Computing, The Australian National University, Canberra ACT 2600, Australia; E-mail: sirintra.vaiwsri@anu.edu.au, peter.christen@anu.edu.au

Thilina Ranbaduge
Data61, Black Mountain, Canberra ACT 2600, Australia;
E-mail: thilina.ranbaduge@data61.csiro.au

Keywords Secure hash encoding · Bit array encoding · String comparison · Privacy-preserving record linkage · Bloom filter encoding

1 Introduction

In application domains such as banking, health, bioinformatics, and national security, it has become an increasingly important aspect in decision making activities to integrate information from multiple databases [11, 19, 56]. Integrating databases can help to identify and link similar records that correspond to the same entity, a task known as *record linkage* [9]. This in turn can facilitate efficient and effective data analysis that is not possible on an individual database.

Increasingly, record linkage needs to be conducted across databases held by different organisations [59], where the complementary information held by these organisations can, for example, help to identify patient groups that are susceptible to certain adverse drug reactions (linking doctor, hospital, and pharmacy databases), or detect welfare cheats (linking taxation with employment and social security databases). However, in many of these applications the databases to be linked contain sensitive information about people which cannot be shared between the organisations that are involved in a linkage protocol [11, 59]. Similarly, in the bioinformatics domain the comparison of genomics data often raises confidentiality concern as genomics sequences might contain proprietary information and such data are often highly sensitive in nature [51].

Research in the area of *privacy-preserving record linkage* (PPRL) [58] aims to develop techniques for linking databases without the need of sharing the original (unencoded) sensitive values between the organisations

Table 1 Example string pairs from a real US voter database [10] that have the same set of q-grams with $q = 2$ (bigrams), and therefore Jaccard or Dice co-efficient similarities of 1.0. On the other hand, their edit distance similarities [9] are (correctly) much lower.

Attribute	String 1	String 2	Bigram set	Edit dist. similarity
Zip code	27828	28278	{27, 28, 78, 82}	0.600
First name	amira	ramir	{am, ir, mi, ra}	0.600
First name	geroge	roger	{er, ge, og, ro}	0.500
First name	jeane	jeaneane	{an, ea, je, ne}	0.625
Last name	avera	raver	{av, er, ra, ve}	0.600
Last name	einstein	steins	{ei, in, ns, st, te}	0.500
Last name	gering	ringer	{er, ge, in, ng, ri}	0.333

that participate in the linkage protocol. In PPRL, the attribute values of records are usually encoded or encrypted in some form before they are being compared. Any encoding or encryption technique used must ensure that approximate similarities can still be calculated between encoded values without the need for sharing the corresponding sensitive plaintext values [58]. PPRL is conducted in such a way that only limited information about the record pairs classified as matches is revealed to the participating organisations in the linkage process. The techniques used in PPRL must guarantee that no participating party, nor any external party, can compromise the privacy of the entities that are represented by records in the databases being linked [11].

One popular technique to allow privacy-preserving string comparison is based on converting strings into sets of q-grams (sub-strings of length q characters) and encoding these sets into Bloom filters (BFs) [45]. BFs are bit arrays where multiple independent hash functions are used to encode the elements of a set by setting those bit positions to 1 that are hit by a hash function. BFs can be compared using set-based similarity functions such as the Dice co-efficient [9]. It has been shown that BF based PPRL is both efficient and can achieve accurate linkage results comparable to non PPRL approaches [42,43]. A related similar approach based on tabulation hash (TMH) encoding was recently proposed by Smith [52]. The proposed approach applies Min-hash locality sensitive hashing [5] and uses the Jaccard similarity function for comparing bit arrays.

One drawback of set-based comparisons as used with BFs or TMH is that the sequence of characters in a string is lost when the string value is converted into a q-gram set. As shown in Table 1, in certain cases [16] two different strings can result in the same q-gram set which would be encoded into the same bit pattern. This can lead to falsely matched record pairs because of too high similarities between rather different string values [9]. The likelihood of two different strings sharing the same or a highly similar q-gram set increases if the size of the alphabet Σ (the set of unique characters used to generate the strings to be encoded) becomes smaller, because less unique q-grams can be generated. Therefore, strings generated using only digits (alphabet of size $|\Sigma| = 10$), such as zip codes or telephone numbers, will more likely result in increased q-gram set similarities compared to strings that contain letters ($|\Sigma| = 26$), such as first and last names.

Another drawback of set-based string comparison functions is that they only allow the calculation of an overall similarity between two strings. However, identifying the longest common sub-string between two strings can be crucial in certain applications. For example, Financial Intelligence Units around the world, including FinCEN (US), the National Crime Agency (UK), and AUSTRAC (Australia), collect financial information to help identify tax evasion, money laundering, and terrorism financing. This involves linking records from different reporting entities such as banks, casinos, and money remitters such as Western Union, and requires finding matches in a privacy-preserving way where bank identifiers such as SWIFT or BIC codes need to be paired with bank account numbers. Sub-string matching is crucial because leading zeros are often omitted, such that the identifier ‘DK54000074491162’ would be the same account as ‘DK5474491162’.

Contributions: In this paper we propose two novel approaches to privacy-preserving string matching, where we encode each string based on its generated q-gram list. In the first approach we encode the q-grams in each list into hash values, while in the second approach we encode each q-gram into a bit array of fixed length to improve privacy of q-grams. However, it requires more runtime for encoding and comparison than the first approach, resulting in a trade-off between privacy and scalability of our two approaches. In both approaches we randomly shift the encoded q-grams in order to hide position information that could be exploited by an adversary. The encoded strings are then sent to a third party for identifying the longest common encoded sub-string for each pair of encoded string pairs. We analyse our proposed approaches in terms of complexity, accuracy, and privacy, and evaluate them using several real and synthetic data sets that contain different types of strings (only letters, only digits, and mixed).

2 Related Work

The privacy-preserving comparison of values (such as strings or numbers) is a common problem for many application domains. Therefore, various techniques and algorithms have been proposed, as shown in Table 2.

Table 2 Overview of related privacy-preserving string matching techniques, where we show the complexity for encoding and matching one string. In this table, l is the string length, $|\Sigma|$ the size of the alphabet Σ, h is the number of hash functions used, b is the length of a Bloom filter or bit array, t is the number of hash tables, q is the length of a sub-string (q-gram), and $|\mathbf{D}|$ is the size of a string database $\mathbf{D}$.

Methods / Authors	Data type	Match type	Encoding compl.	Matching compl.	Application
Bloom filter (Schnell et al. [45])	String	Approximate	$O(l \times h)$	$O(b)$	PPRL
Tabulation hashing (Smith [52])	String	Approximate	$O(l \times t \times h)$	$O(b)$	PPRL
DGK approximate string matching (Essex [24])	String	Approximate	$O(\Sigma^q)$	$O(\Sigma^q)$	PPRL
Burrows-Wheeler transformation (Shimizu et al. [51])	Genomes	Exact	$O(l \times \sqrt{l \times \lvert\Sigma\rvert})$	$O(l^2 \times \lvert\Sigma\rvert)$	Genomics
Longest prefix and exact match (Nakagawa et al. [41])	Genomes	Exact	$O(l\lvert\mathbf{D}\rvert)$	$O(l)$	Genomics
Symmetric encrypted suffix tree (Chase and Shen [6])	String	Exact	$O(l \times b)$	$O(l \times b)$	Cloud computing
Bloom filter tree (Bezawada et al. [3])	String	Exact	$O(l^2 \times h)$	$O(l \times \log\ l)$	Cloud computing
Secure verifiable pattern matching (Chen et al. [7])	Genomes	Exact	$O(l)$	$O(l)$	Cloud computing
Secure query sub-string (Hahn et al. [30])	String	Exact	$O(2l)$	$O(l)$	PPRL
Frequent q-grams matching (Bonomi et al. [4])	String	Approximate	$O(3l)$	$O(b)$	PPRL
Secure pattern matching (Zarazadeh et al. [62])	Genomes	Exact&Approx.	$O(l \times \lvert\Sigma\rvert)$	$O(l^2)$	Cloud computing

String matching is often used in the PPRL context where encoded values of quasi-identifying attributes of individuals (such as their names and addresses) need to be compared across two or more databases to link records [59]. Bloom filter (BF) encoding is widely used in PPRL because it allows efficient encoding of values and supports approximate matching of strings [45, 59], numerical values [34,57], hierarchical codes (such as of occupation and diseases) [46,47], geographical locations [48], and Chinese characters [54].

Although BF encoding is considered as a standard for PPRL, BFs cannot be used to identify the longest common sub-strings, because they require values to be converted into q-gram sets whereby positional information of q-grams in their corresponding string values are lost. Furthermore, the hash functions used in BF encoding likely lead to collisions (where several q-grams are hashed to the same bit position) and therefore the similarities between BFs, can be higher than the actual Dice co-efficient similarity between their corresponding q-gram sets. Other set-based techniques, such as tabulation based hashing (TMH) [52] have similar drawback because any set-based encoding of q-grams into bit arrays does not preserve their positional order.

Privacy-preserving matching of genome sequences is increasingly required in bioinformatics applications where the aim is to find the longest matching subsequences for a query sequence in large genome databases [51,60]. The algorithms used in such applications often have high computational complexities [51,60].

Shimizu et al. [51] proposed an approach for searching similar string patterns in a genome database using a recursive oblivious transfer protocol based on an additive homomorphic encryption [26] to query sequences in the genome database while ensuring each query does not lead to the identification of other similar strings in the database. However, the approach does not scale to queries of longer sequences because these incur high computational and communication costs due to the complex cryptographic functions used [51].

Later, Nakagawa et al. [41] proposed an approach to improve the time complexity and communication costs of genome sequence matching. They used a recursive oblivious transfer technique and a compressed indexing data structure [25] to find the longest prefix and longest exact match of a query sequence in a genome database. In this approach the time complexity depends upon the length of the genome sequence that is being queried rather than the size of the genome database, thus, it consumes less time to query a sequence from a large genome database compared to the approach proposed by Shimizu et al. [51] and the other secure genome sequence matching technique [53].

Suffix trees [38] are often used in bioinformatics applications to search for patterns in genome or protein sequences [61]. A suffix tree allows searching for a given pattern with a linear complexity in terms of the length of the query string being searched [38]. Ukkonen [55] showed how suffix trees can be used for string matching efficiently, however his approach requires more space to hold a suffix tree than the original string collection.

The use of suffix trees in privacy-preserving substring matching has been investigated by Chase and Shen [6]. Their approach constructs a queryable encryption scheme for finding all occurrences of a query string in the encrypted suffix tree stored on an untrusted server. However, this approach reveals information of client queries to the server which compromises the privacy of a client's data and can potentially lead to the identification of the encrypted string values.

Bezawada et al. [3] proposed a protocol based on a pattern aware secure search tree where each tree node contains a BF that encodes a set of encrypted strings. The approach is aimed for two parties to compare strings securely over a cloud infrastructure, where the parties only learn if their strings are matched but not the actual matching sub-strings. This approach therefore does not allow the privacy-preserving identification of the longest common sub-strings.

Chen et al. [7] proposed a secure pattern matching approach based on suffix arrays and order hashing, where each hashed character in a string is concatenated with the hashed value of the next character in the string. In this approach a database owner (DO) sends the encrypted data to an untrusted server and transmits the key to the clients. This key is then used by the clients for verifying if the encoded query substring results that the clients received from the server are correct. However, in this approach the server can learn the actual string length from the encrypted suffix array received from the DO.

Hahn et al. [30] proposed a privacy-preserving secure sub-string or q-gram query approach where the frequency distribution of sensitive data are hidden by applying a frequency-hiding order preserving encryption [36]. This approach involves three parties for processing a secure q-gram query, which are (1) the DO which owns the sensitive information, encodes the q-grams, and generates the encoded data tables (indexes and q-grams); (2) the untrusted party which holds the index tables that are used for searching encoded q-gram; and (3) the clients who want to query their encoded q-grams. In this approach the complexity of querying depends upon the q-gram length because it determines the number of iterations required for querying encoded q-grams from the untrusted party.

Bonomi et al. [4] proposed a PPRL approach to compare string values using bit arrays based on the embedding of the frequent q-grams. The DOs that participate in a PPRL protocol individually apply differential privacy [21] to generate a table of frequent q-grams that occur in their databases. The DOs then send their frequent q-gram tables to one of the DOs to find the common frequent q-grams (shared frequent q-grams) and send them to all DOs that participate in the protocol. Each DO then uses the shared frequent q-grams to embed their strings into bit arrays, and sends these bit arrays to a third party to compare pairs of bit arrays. However, in this approach the DO that identifies the shared frequent q-grams is able to learn the frequent q-grams of the other databases which can compromise the privacy of the entities in those databases.

Zarazadeh et al. [62] proposed a protocol for secure pattern matching on a client and server architecture using ElGamal encryption [23] and bit arrays. This approach is able to match either exact or approximate patterns with or without wildcard characters, or patterns with random bit vectors added (for hiding the length of strings). The server cannot learn anything about the pattern matching results. However, the clients can learn the positions of a matched pattern in a string in the database stored on the server.

Essex [24] proposed a secure two-party approximate string matching protocol using DGK homomorphic encryption [17] and private set intersection cardinality. Each DO first generates a list of all possible q-grams (based on letters a to z), where the DO replaces a q-gram in the list of all possible q-grams with the encryption of 1 if a q-gram is found in its string. The first DO sends its list to the other DO to conduct a set intersection cardinality and the Dice co-efficient calculation based on the lengths of q-gram lists of the two strings in a pair, then returns the results to the first DO for decrypting the results, where the decryption of 1 means a pair of strings is classified as a match. However, using the lengths of the lists of q-grams to calculate the Dice co-efficient can lead to false matches because some q-grams in the two lists are not common. Furthermore, this approach consumes a lot of memory as the list of all possible q-grams for each string needs to be kept in memory for the comparison process.

Recently, Mullaymeri and Karakasidis [40] proposed a two-party private approximate string matching protocol based on polynomial coefficients generated using a reference database and a Fuzzy Vault scheme [32]. The idea behind this approach is that if the set of keys (generated from reference strings) of the two strings in a pair are similar, the polynomial coefficients generated from the keys of these two strings must be the same, and therefore the two strings in a pair are classified as a match. However, the main drawback of this approach is that the reference strings that are used to generate keys must be very similar to the strings in a pair to ensure that the polynomial coefficients of the two strings are the same. Therefore, the number of reference strings must be large enough to allow the two parties to generate the same polynomial coefficients.

The approaches discussed above mostly allow a user to query a database of strings or sequences for similar patterns, while the problem we aim to address involves the identification of similar sub-strings in two databases owned by different parties without each party having to reveal their strings. In contrast to most existing techniques, our approaches allow the efficient and accurate privacy-preserving comparison of pairs of strings that share a sub-string with a certain minimum length.

3 Privacy-Preserving String Matching

We assume our approaches follow a *honest-but-curious* (HBC) adversary model [28,31]. As illustrated in Fig. 1, we assume two database owners (DOs) want to find the length of the longest common sub-string (LCS) between

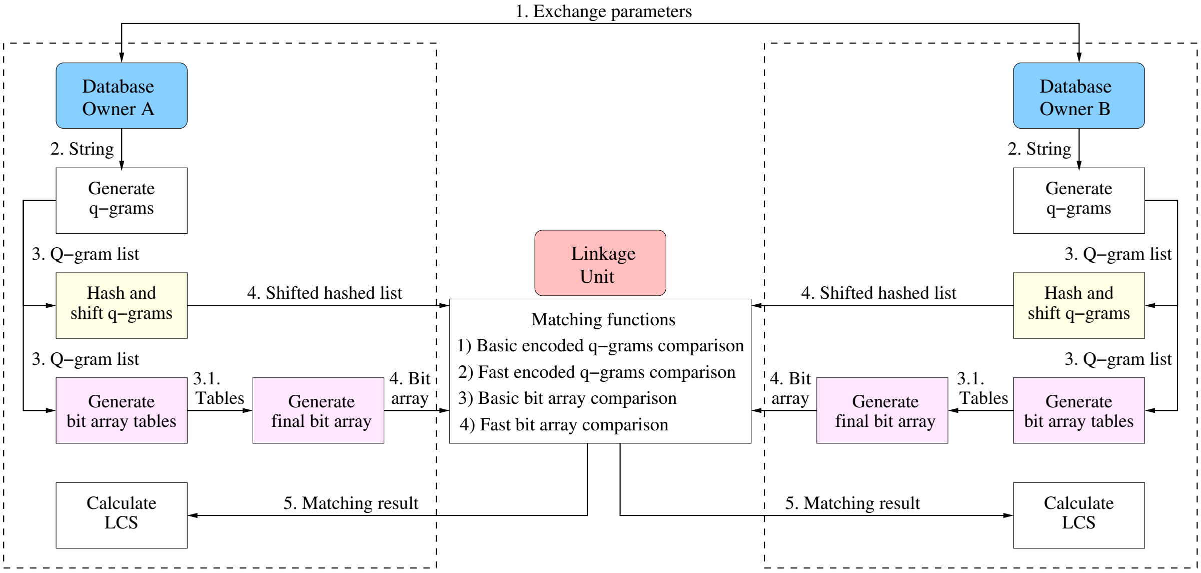


Fig. 1 Overview of our privacy-preserving string matching approaches. The blue boxes are the Database Owners, where the steps they execute are shown in the dashed rectangles. The white boxes are the steps common to both our two approaches. The yellow box shows the step of the shifted hash encoded q-gram based encoding approach (described in Section 4), while the pink boxes show the steps of the bit array based approach (described in Section 5). The Linkage Unit is shown in the red box and its functions are shown in the box below it. The arrows show inputs and outputs of the steps, where the numbers given show the sequence of the steps being conducted.

pairs of sensitive strings in their databases. The DOs do not communicate with each other, except to agree on the parameters to be used. We assume a linkage unit (LU), which is a semi-trusted party [27], is involved in the protocol to compare the strings sent to it by the DOs. Because the DOs do not want to reveal the sensitive string values in their databases to any other party that participates in the protocol, these strings need to be encoded before being sent to the LU such that the LU cannot learn anything about them.

In some cases, the first characters in a string value can reveal some information. For example, the distribution of the first digits in numerical values can follow Benford's law [2], while the first letters in first and last names can follow Zipf's law [63]. This potentially allows an adversary to learn some of the encoded q-grams at the beginning of encodings by identifying the q-grams that occur frequently in a public database [11]. Hence, we propose two novel encoding approaches to prevent any q-grams from being re-identified.

In our encoding approaches (as illustrated in Fig. 1), the DOs first generate sub-strings of length q, called q-grams, from their unique sensitive string values. The DOs then individually encode all q-grams of each string and send these encoded q-grams to the LU. The LU then compares the encoding of a pair of strings and returns the length of the longest common sequence of hash encoded q-grams (elements), called the *Longest Common Elements* (LCE), or the length of the longest common bit array, called the *Longest Common Bit Array* (LCB), back to the DOs. The DOs can then calculate the actual length of LCS based on the information received from the LU, as we describe in Section 6.

As we discuss in Section 4, the first encoding approach improves privacy of encoded q-grams by randomly shifting the position of the encoded q-grams in the generated encoded q-gram lists. The shifting of encoded q-grams hides their actual positions in a string, which makes a position based frequency analysis of q-grams more difficult and thereby prevents an adversary from identifying the string values that were encoded. This approach is useful for linking databases that require fast and accurate linkage results, such as linking the phone number of a criminal between databases to facilitate a fast response for police to take action.

In the second approach, described in Section 5, we improve the privacy of q-grams further by encoding q-grams into bit arrays. We hide the actual sub-string positions and the length of the encoded strings by adding random bit arrays at the beginning and end of the bit array that encodes a list of q-grams. This ensures that the bit arrays of all string values in a database have the same length, further increasing the difficulty for an adversary to identify the original string values that have been encoded into bit arrays. However, this approach uses more runtime than our first approach. Therefore, it can be useful for linking databases in application domains that require high privacy and accurate linkage results, but are less concerned about runtime, for example, linking credit card numbers between databases for financial crime investigations.

Both our approaches provide accurate calculations of the length of LCS while hiding the actual sensitive string values from any parties. The DOs and the LU cannot learn the original string values nor the positions of the LCS from the compared encodings. A DO cannot learn anything about the sensitive values of the other DO because the DOs do not communicate with each other, and they only receive the length of the LCE or LCB, respectively, from the LU.

As notation, we use italics type letters for numbers and strings, bold lowercase letters for lists and sets, uppercase bold letters for lists and sets of lists or sets. We use $||$ to denote the concatenation of strings and bit arrays, and $+$ when concatenating lists. Lists are shown with square and sets with curly brackets, where lists have an order while sets do not. We show the elements of a list $\mathbf{l}$ as $\mathbf{l}[i]$, with $0 \leq i < |\mathbf{l}|$, where i is the position (index) of a list element. We denote sub-lists as $\mathbf{l}[i{:}] = [\mathbf{l}[i], \mathbf{l}[i+1], \ldots \mathbf{l}[|\mathbf{l}|-1]]$, $\mathbf{l}[{:}j] = [\mathbf{l}[0], \mathbf{l}[1], \ldots, \mathbf{l}[j-1]]$, and $\mathbf{l}[i{:}j] = [\mathbf{l}[i], \mathbf{l}[i+1], \ldots, \mathbf{l}[j-1]]$, with $i < j$.

4 String Matching based on Shifted Hash Encoded Q-grams

As shown in Fig. 1, our first approach consists of five steps: (1) parameter agreement, (2) generating q-grams, (3) hashing of q-grams and shifting encoded q-gram lists, (4) comparing lists of encodings by the LU, and (5) calculation of the length of the LCS by the DOs.

4.1 Parameter Agreement

Before the protocol starts, the DOs agree on the parameters to be used, which are:

- The length of q-grams, q, to be used for generating the q-grams, as we describe in Section 4.2.
- The padding characters, α and β, to be added to string values to avoid any incorrect length of LCS calculations, as we describe in Sections 4.2 and 8.2.
- The secret salting value, s, to be concatenated with the generated q-grams. This is to avoid a dictionary attack by the LU [11], as we describe in Section 4.3.
- The one-way hash function [11], $\mathcal{H}$ (such as SHA [44]), to be used for hashing q-grams before sending them from the DOs to the LU, as we describe in Section 4.3.
- The minimum length of the LCS, m, where $m \geq q$. This is used for selecting those string pairs that have a LCS of at least m, as we describe in Section 6.

4.2 Generating Q-grams

Before the generation of q-grams from a string value, the DOs first add the agreed padding characters α and β to the beginning and end of their unique strings. Let us assume the first DO has a database $\mathbf{D}_A$ and uses the padding character α, while the second DO has a database $\mathbf{D}_B$ and uses the padding character β, where $\alpha \neq \beta$. The padding characters are used to ensure the beginning and end of the compared strings are different. Due to the shifting process of encoded q-grams (as we describe in Section 4.3), without the padding characters the length of LCS could be calculated incorrectly, as we discuss in Section 8.2.

Assuming Σ is the alphabet of all characters in the databases $\mathbf{D}_A$ and $\mathbf{D}_B$, $\Sigma = \{a \in v : v \in \mathbf{D}_A \cup \mathbf{D}_B\}$, where a is a character in a string value v in the two databases. It needs to hold that $\alpha \notin \Sigma$ and $\beta \notin \Sigma$. Let us assume two padded strings $x' = \alpha||x||\alpha$ and $y' = \beta||y||\beta$, where x and y are strings with $x \in \mathbf{D}_A$ and $y \in \mathbf{D}_B$, respectively.

Once the DOs have added the padding characters to their strings, they independently generate the q-gram lists of each padded string in their databases. Each padded string in a database (let us use $\mathbf{D}_A$), consists of characters and can be written as $x' = [a_0 \ldots a_i \ldots a_{n-1}]$, where $a_0 = a_{n-1} = \alpha$, and $a_i \in \Sigma$ for $0 < i < (n-1)$, with $n = |x'|$. We define a q-gram as $q_i = a_i \ldots a_{i+q-1}$, and a q-gram list as $\mathbf{q} = [q_0, \ldots, q_{n-q}]$, where q is the q-gram length as agreed by the DOs.

For example, assume the agreed q-gram length is $q = 2$ and the string is $x =$ "$mary$". The DO adds the padding character $\alpha =$ "\$" to x, resulting in $x' =$ "\$$mary$\$". The DO then generates a q-gram list of the padded string x', resulting in $\mathbf{q} = [\$m, ma, ar, ry, y\$]$, as also shown in the third column in Table 3.

4.3 Hashing of Q-grams and Shifting Q-gram Lists

Before the DOs send their databases to the LU, they individually hash encode the q-grams in each of the q-gram lists using the agreed hash function $\mathcal{H}$. To prevent a dictionary attack on the encoded q-grams, we use a salted hash encoding approach [11]. Given s is the secret salt value and $\mathcal{H}$ is the hash function agreed by the DOs, and assuming $q_i \in \mathbf{q}$, where $\mathbf{q}$ is the q-gram list and $0 \leq i < n$, with $n = |\mathbf{q}|$, we hash encode each q-gram q_i as $h_i = \mathcal{H}(q_i||s)$, and define the hash encoded q-gram list as $\mathbf{h} = [h_0, h_1, ..., h_{n-1}]$.

Once the q-grams in each list $\mathbf{q}$ are hashed into a list $\mathbf{h}$, each DO generates a random number, r, for each of its hash encoded q-gram lists, $\mathbf{h}$, where $0 \leq r < |\mathbf{h}|$. The DO then shifts (rotates) the list $\mathbf{h}$ by r, resulting

Table 3 Example strings, and their corresponding q-grams and randomly shifted q-gram lists, where the patterns of where common characters occur are shown in the first column (where b, m, and e represent the LCS occurring at the beginning, middle, or end of a string, respectively). The string pairs, with minimum length of the LCS $m = 3$, are shown in the second column. Q-grams are generated using $q = 2$. LCE refers to the longest common list of elements. The random numbers used to shift each q-gram list are shown in the fourth column. The common sub-strings and the common elements are shown in bold.

Common patterns	String pairs	Q-gram lists (including padding characters)	Random shift	Shifted q-gram lists	LCE length, *lce*	Calculated LCS, *lcs*
b-b	x = “**mar**y” y = “**mar**ie”	$[\$m, \mathbf{ma}, \mathbf{ar}, ry, y\$]$ $[\#m, \mathbf{ma}, \mathbf{ar}, ri, ie, e\#]$	$r_x = 0$ $r_y = 2$	$[\$m, \mathbf{ma}, \mathbf{ar}, ry, y\$]$ $[ie, e\#, \#m, \mathbf{ma}, \mathbf{ar}, ri]$	2	$2 + 2 - 1 = 3$
b-m	x = “**marr**ie” y = “e**marr**y”	$[\$m, \mathbf{ma}, \mathbf{ar}, \mathbf{rr}, ri, ie, e\$]$ $[\#e, em, \mathbf{ma}, \mathbf{ar}, \mathbf{rr}, ry, y\#]$	$r_x = 1$ $r_y = 3$	$[e\$, \$m, \mathbf{ma}, \mathbf{ar}, \mathbf{rr}, ri, ie]$ $[\mathbf{rr}, ry, y\#, \#e, em, \mathbf{ma}, \mathbf{ar}]$	3	$3 + 2 - 1 = 4$
b-e	x = “**lar**isa” y = “ca**lar**”	$[\$l, \mathbf{la}, \mathbf{ar}, ri, is, sa, a\$]$ $[\#c, ca, al, \mathbf{la}, \mathbf{ar}, r\#]$	$r_x = 4$ $r_y = 2$	$[ri, is, sa, a\$, \$l, \mathbf{la}, \mathbf{ar}]$ $[\mathbf{la}, \mathbf{ar}, r\#, \#c, ca, al]$	2	$2 + 2 - 1 = 3$
m-m	x = “m**ari**sa” y = “c**ari**e”	$[\$m, ma, \mathbf{ar}, \mathbf{ri}, is, sa, s\$]$ $[\#c, ca, \mathbf{ar}, \mathbf{ri}, ie, e\#]$	$r_x = 3$ $r_y = 2$	$[is, sa, a\$, \$m, ma, \mathbf{ar}, \mathbf{ri}]$ $[ie, e\#, \#c, ca, \mathbf{ar}, \mathbf{ri}]$	2	$2 + 2 - 1 = 3$
m-e	x = “m**alar**y” y = “c**alar**”	$[\$m, ma, \mathbf{al}, \mathbf{la}, \mathbf{ar}, ry, y\$]$ $[\#c, ca, \mathbf{al}, \mathbf{la}, \mathbf{ar}, r\#]$	$r_x = 1$ $r_y = 3$	$[y\$, \$m, ma, \mathbf{al}, \mathbf{la}, \mathbf{ar}, ry]$ $[\mathbf{la}, \mathbf{ar}, r\#, \#c, ca, \mathbf{al}]$	3	$3 + 2 - 1 = 4$
e-e	x = “m**ary**” y = “c**ary**”	$[\$m, ma, \mathbf{ar}, \mathbf{ry}, y\$]$ $[\#c, ca, \mathbf{ar}, \mathbf{ry}, y\#]$	$r_x = 2$ $r_y = 3$	$[\mathbf{ry}, y\$, \$m, ma, \mathbf{ar}]$ $[\mathbf{ar}, \mathbf{ry}, y\#, \#c, ca]$	2	$2 + 2 - 1 = 3$
be-be	x = “**mary**” y = “**marary**”	$[\$m, \mathbf{ma}, \mathbf{ar}, \mathbf{ry}, y\$]$ $[\#m, \mathbf{ma}, \mathbf{ar}, ra, \mathbf{ar}, \mathbf{ry}, y\#]$	$r_x = 2$ $r_y = 5$	$[\mathbf{ar}, \mathbf{ry}, y\$, \$m, \mathbf{ma}]$ $[\mathbf{ar}, ra, \mathbf{ar}, \mathbf{ry}, y\#, \#m, \mathbf{ma}]$	2	$2 + 2 - 1 = 3$

in the shifted hash encoded q-gram list, $\mathbf{h}'$. A given hash encoded value at a position i, with $0 \leq i < n$, is shifted to a new position $i' = ((i+r) \bmod n)$, also with $0 \leq i' < n$. This shifting process aims to hide the original positions of the hash encoded q-grams and therefore the corresponding positions of characters in each string. Hence, the frequency distribution of shifted hash encoded q-grams will not follow Benford's [2] law anymore because the first encoded q-grams are distributed to different positions in the shifted hash encoded q-gram lists, thereby preventing a position based frequency attack on these encoded q-gram lists.

For example, assume the string "*mary*" has been padded and hashed into the hash encoded q-gram list $\mathbf{h}_x = [\mathcal{H}(\$m||s), \mathcal{H}(ma||s), \mathcal{H}(ar||s), \mathcal{H}(ry||s), \mathcal{H}(y\$||s)]$, and $r_x = 2$. Therefore, the hash encoded q-grams are shifted by two positions, resulting in $\mathbf{h}'_x = [\mathcal{H}(ry||s), \mathcal{H}(y\$||s), \mathcal{H}(\$m||s), \mathcal{H}(ma||s), \mathcal{H}(ar||s)]$. We show other examples of shifted q-gram lists in Table 3.

4.4 Comparison of Hash Encoded Q-gram Lists

To simplify notation, we now use $\mathbf{h}_x$ and $\mathbf{h}_y$ to represent the shifted hash encoded q-gram lists $\mathbf{h}'_x$ and $\mathbf{h}'_y$.

For a pair of strings, the LU needs to find the length of the longest common (same) sub-list elements (LCE), *lce*, between the two lists $\mathbf{h}_x$ and $\mathbf{h}_y$. We propose two algorithms for this comparison process, a basic and a fast algorithm. The first is a naive method to conduct the comparison by the LU which however requires longer runtimes for comparing pairs of hashed q-gram lists. We then propose an alternative, more efficient, algorithm which allows for a faster comparison process, as we describe in Section 4.4.2.

4.4.1 Basic Encoded Q-grams Comparison Algorithm

Because of the random shifting process performed by the DOs, the LU does not know the start and end positions of the hash encoded q-grams in the lists $\mathbf{h}_x$ and $\mathbf{h}_y$ to be compared. To compare two shifted hash encoded q-gram lists, the LU needs to rotate the two lists to find the length of their LCE, *lce*, and then returns the *lce* to the DOs.

Algorithm 1 outlines the naive way for comparing two encoded q-gram lists. In line 1, the LU first initialises the length of LCE, *lce*. It then checks if the two lists, $\mathbf{h}_x$ and $\mathbf{h}_y$, have any common elements (in line 2). If there are common elements between the two lists, then the LU loops over each position, p_x, of the elements in the list $\mathbf{h}_x$ (in line 3), where we denote the element in $\mathbf{h}_x$ at position p_x as $\mathbf{h}_x[p_x]$.

In line 4, the LU finds all positions where the element $\mathbf{h}_x[p_x]$ occurs in the list $\mathbf{h}_y$ using the function *getPosMatch*(). This function returns a list of positions, $\mathbf{p}_y$, where this list is empty if $\mathbf{h}_x[p_x]$ does not occur in $\mathbf{h}_y$. In line 5, the LU then generates the shifted (rotated) list, $\mathbb{h}_x$, with a starting position of p_x by concatenating the two sub-lists $\mathbf{h}_x[p_x{:}]$ and $\mathbf{h}_x[{:}p_x]$ into one shifted list $\mathbb{h}_x$.

The LU then loops over each position, p_y, in the list of positions $\mathbf{p}_y$ in line 6, and in line 7 generates the shifted list, $\mathbb{h}_y$, with starting position p_y, similar to the shifted list $\mathbb{h}_x$ generated in line 5. The common element at positions p_x and p_y now becomes the first element of the two rotated lists $\mathbb{h}_x$ and $\mathbb{h}_y$, respectively.

In line 8, the LU initialises an index k as a count of the number of common elements between $\mathbb{h}_x$ and $\mathbb{h}_y$. The LU then loops over the rotated list that has the

Algorithm 1: ***Basic encoded q-grams comparison process by the LU***

```
Input:
- $\mathbf{h}_x$:   Hashed and shifted q-gram list of string $x$
- $\mathbf{h}_y$:   Hashed and shifted q-gram list of string $y$
Output:
- $lce$:   LCE of the hashed q-gram list pair
1:  $lce \leftarrow 0$                                                   // Initialise the length of LCE
2:  if $set(\mathbf{h}_x) \cap set(\mathbf{h}_y) \neq \emptyset$ do:                          // Check if common hashed elements exist
3:     for $0 \leq p_x < |\mathbf{h}_x|$ do:                                 // Loop over each position in the list $\mathbf{h}_x$
4:        $\mathbf{p}_y \leftarrow getPosMatch(\mathbf{h}_x[p_x], \mathbf{h}_y)$        // Get list of positions in $\mathbf{h}_y$ where the element $\mathbf{h}_x[p_x]$ occurs
5:        $\mathbb{h}_x \leftarrow \mathbf{h}_x[p_x{:}] + \mathbf{h}_x[{:}p_x]$                   // Get the current shifted (rotated) list of the list $\mathbf{h}_x$
6:        for $p_y \in \mathbf{p}_y$ do:                      // Loop over each position in the list of positions $\mathbf{p}_y$
7:           $\mathbb{h}_y \leftarrow \mathbf{h}_y[p_y{:}] + \mathbf{h}_x[{:}p_y]$                      // Get the current shifted list of the list $\mathbf{h}_y$
8:           $k \leftarrow 0$                                          // Initialise index and common count $k$
9:           while $(k < min(|\mathbb{h}_x|, |\mathbb{h}_y|))$ and $(\mathbb{h}_x[k] = \mathbb{h}_y[k])$ do:  // Loop over $\mathbb{h}_x$ and $\mathbb{h}_y$ if a common element occurs
10:             $k \leftarrow k + 1$                                                   // Increment $k$
11:          $lce \leftarrow max(lce, k)$                     // Keep the length of the so far maximum length of LCE
12: return $lce$                                           // Send found the length of LCE back to the DOs
```

Algorithm 2: ***Fast encoded q-grams comparison process by the LU***

```
Input:
- $\mathbf{h}_x$:   Shifted hashed q-gram list of string $x$
- $\mathbf{h}_y$:   Shifted hashed q-gram list of string $y$
Output:
- $lce$:   The LCE length of the pair of hashed q-gram lists
1:  $lce \leftarrow 0$                                                                 // Initialise the length of LCE
2:  $\mathbf{c} \leftarrow set(\mathbf{h}_x) \cap set(\mathbf{h}_y)$                                                   // Get common hashed elements
3:  if $|\mathbf{c}| \geq 1$ do:                                            // Check if at least one common hashed element exists
4:     $\mathbf{h}_s, \mathbf{h}_l \leftarrow \mathbf{h}_x, \mathbf{h}_y$ if $|\mathbf{h}_x| < |\mathbf{h}_y|$ else $\mathbf{h}_y, \mathbf{h}_x$                                  // Order the lists by their lengths
5:     $\mathbb{h}_s \leftarrow \mathbf{h}_s + \mathbf{h}_s$                                              // Concatenate shorter list with a copy of itself
6:     $\mathbb{h}_l \leftarrow \mathbf{h}_l + \mathbf{h}_l$                                               // Concatenate longer list with a copy of itself
7:     $\mathbf{p}_s \leftarrow getConsecCommon(\mathbb{h}_s, \mathbf{c})$           // Get a list of positions of consecutive common hashed elements in the list $\mathbb{h}_s$
8:     if $|\mathbf{p}_s| = 0$ do:                                            // Check if no consecutive hashed elements are found
9:        $lce \leftarrow 1$                                                // There is no consecutive common hashed element
10:    else:                                                                   // If consecutive elements are found
11:       for $p_s \in \mathbf{p}_s$ do:                     // Loop over each position in the list of consecutive common element positions $\mathbf{p}_s$
12:          $\mathbf{p}_l \leftarrow getPosMatch(\mathbb{h}_s[p_s], \mathbb{h}_l)$                   // Get the list of positions in $\mathbb{h}_l$ where the element $\mathbb{h}_s[p_s]$ occurs
13:          for $p_l \in \mathbf{p}_l$ do:                                     // Loop over each position in the list of positions $\mathbf{p}_l$
14:             $k \leftarrow 0$                                                       // Initialise index and common count $k$
15:             while $(k < min(|\mathbb{h}_s[p_s{:}p_s + |\mathbf{h}_s|]|, |\mathbb{h}_l[p_l{:}p_l + |\mathbf{h}_l|]|))$ and $(\mathbb{h}_s[p_s + k] = \mathbb{h}_l[p_l + k])$ do:  // Loop over elements in $\mathbb{h}_s$ and $\mathbb{h}_l$
16:                $k \leftarrow k + 1$                                                                // Increment $k$ by 1
17:             $lce \leftarrow max(lce, k)$                                                            // Keep the length of LCE
18: return $lce$                                                       // Send found the length of LCE back to the DOs
```

shortest length, with the condition that the elements in the two rotated lists at position k are common (are the same). The LU keeps the maximum k identified over all iterations of rotated lists in line 11.

The LU repeats the steps in lines 3 to 11 until the element $\mathbf{h}_x[|\mathbf{h}_x|-1]$ becomes the first element in the rotated shifted list $\mathbb{h}_x$. Finally, the LU returns the length of LCE, lce, to the DOs in line 12.

4.4.2 Fast Encoded Q-grams Comparison Algorithm

As outlined in Algorithm 2, the LU uses concatenated hashed q-gram lists, $\mathbb{h}_s$ and $\mathbb{h}_l$, for the comparison. We use such a concatenation technique because (1) the concatenated list contains the actual sequence of consecutive elements in the hash encoded q-gram list before it has been shifted, and (2) even after concatenation the actual positions of the hash encoded q-grams are not being revealed to the LU, as we discuss in Section 8.3.

Let us use the q-gram list $\mathbf{q} = [\$m, ma, ar, ry, y\$]$ as an example. With the random number $r = 2$, the shifted q-gram list becomes $\mathbf{q}' = [ry, y\$, \$m, ma, ar]$. We now concatenate $\mathbf{q}'$ with itself to generate a concatenated list $\mathbb{q}' = [ry, y\$, \$\mathbf{m}, \mathbf{ma}, \mathbf{ar}, \mathbf{ry}, \mathbf{y}\$, \$m, ma, ar]$. As can be seen from this example, the concatenated list $\mathbb{q}'$ does contain the actual sequence of consecutive elements of the q-gram list $\mathbf{q}$ (shown in bold).

In line 1 of Algorithm 2, the LU first initialises the length of the LCE, lce. It then finds the common elements between the two lists, $\mathbf{h}_x$ and $\mathbf{h}_y$, it received from the DOs, and adds these common elements into the set $\mathbf{c}$ in line 2. If common elements occur between $\mathbf{h}_x$ and $\mathbf{h}_y$, then the LU orders the two lists by their lengths, and assigns the shorter list to $\mathbf{h}_s$ and the longer list to $\mathbf{h}_l$ in lines 3 and 4. The LU then concatenates in lines 5 and 6 each of $\mathbf{h}_s$ and $\mathbf{h}_l$ with a copy of itself, resulting in the two concatenated lists $\mathbb{h}_s$ and $\mathbb{h}_l$, respectively.

In line 7 the LU then finds the list $\mathbf{p}_s$ of consecutive positions in $\mathbb{h}_s$ of the common elements in $\mathbf{c}$ by using the function *getConsecCommon*(). For example, the first string pair in Table 3, the string $x =$ "*mary*" has the list $\mathbf{p}_s = [1, 2]$ which are the positions of q-grams *ma* and *ar*, respectively. The list $\mathbf{p}_s$ is empty if there is no sequence of consecutive common elements occurring in both $\mathbf{h}_x$ and $\mathbf{h}_y$. The length of LCE is returned as $lce = 1$ in line 9 (the lce is 1 because the set $\mathbf{c}$ is not empty, as tested in line 3).

If $\mathbf{p}_s$ does contain consecutive common elements, then the LU loops over each position p_s in the list $\mathbf{p}_s$ in lines 10 and 11. In line 12, the LU finds the list of positions, $\mathbf{p}_l$, in the longer concatenated list, $\mathbb{h}_l$, where the element at position p_s of the concatenated list, $\mathbb{h}_s$, occurs by using the function *getPosMatch*(). The LU then loops over each position p_l in $\mathbf{p}_l$ in line 13.

In line 14 the LU then initialises the index k as the count of the number of common elements between the two concatenated lists, $\mathbb{h}_s$ and $\mathbb{h}_l$. In line 15, it loops over the shorter concatenated sub-list $\mathbb{h}_s$, where this sub-list is not longer than $|\mathbf{h}_s|$, starting from position p_s, and the sub-list of the longer concatenated list $\mathbb{h}_l$ is not longer than the length $|\mathbf{h}_l|$, starting from position p_l. The loop terminates once the elements at positions $p_s + k$ and $p_l + k$ are different.

If there are common elements in the two compared sub-list $\mathbb{h}_s$ and $\mathbb{h}_l$, then the LU increases the index k in line 16. Once the loop is terminated, in line 17 the LU finds the maximum length of the LCE identified so far. The LU repeats the steps in lines 11 to 17 until there are no further positions to be compared. Finally, the LU returns the length of the LCE, lce, of the pair of hash encoded q-gram lists to the DOs in line 18.

5 String Matching based on Shifted Random Bit Arrays

While our approach based on shifted hash encoded q-grams prevents frequency attacks that are exploiting Benford's law [2], an adversary might still be able to identify the most frequent q-grams because these are encoded into hash values that will become the most frequent in the lists of hash encoded q-grams, $\mathbf{h}$. To prevent such attacks, we improve the privacy of the shifted hash encoded q-gram based approach by encoding each unique q-gram list $\mathbf{q}$ into a bit array. Each such bit array is padded at the beginning and end with random bits to ensure the bit arrays of all encoded strings have the same length even if the length of their q-gram lists differ. This approach prevents the LU from identifying which sub-sequence of bits in a bit array correspond to which q-gram, as we discuss further in Section 8.3.

5.1 Generating Bit Arrays for Strings

To generate bit arrays for the strings in a database $\mathbf{D}$, each DO builds two tables of unique bit arrays. The first is the table $\mathbf{T}_\Sigma$ which contains one unique bit array for each possible q-gram that can be generated from the alphabet Σ, where Σ contains all characters that occur in the string values of the databases $\mathbf{D}_A$ and $\mathbf{D}_B$ being compared, plus the padding characters, α and β. The second table, $\mathbf{T}_R$, contains random bit arrays which will be used as padding to make the bit arrays of all q-gram lists the same length, where each DO needs to generate a unique table of such random bit arrays in order to prevent false matches, as we discuss further below.

Before building the tables $\mathbf{T}_\Sigma$ and $\mathbf{T}_R$, each DO first generates all unique q-grams that can be obtained from the alphabet Σ based on the q-gram length, q, where $\Sigma = \{a \in v : v \in \mathbf{D}_A \cup \mathbf{D}_B\} \cup \{\alpha, \beta\}$. The total number of possible q-grams we obtain is $|\Sigma|^q$.

Based on the sizes of Σ, the DOs now need to calculate the q-gram bit array length, l_q, to be used for generating the unique bit array for each possible q-gram. Because each DO needs to generate two tables of bit arrays ($\mathbf{T}_\Sigma$ and $\mathbf{T}_R$), where the random bit arrays $\mathbf{T}_R$ must be different between the two databases $\mathbf{D}_A$ and $\mathbf{D}_B$, the bit array length l_q must be large enough to allow at least $3|\Sigma|^q$ unique bit arrays to be generated. We can calculate a minimum length for l_q as $l_q^{min} = log_2(3|\Sigma|^q)$. This would however require every possible combination of bits to be generated, including $[0] \times l_q^{min}$ and $[1] \times l_q^{min}$. Such patterns could however reveal information as their frequencies of occurrence could be analysed by an adversary.

Therefore, to provide maximum entropy, which will make it more difficult for a frequency attack to be performed, each DO randomly generates bit arrays where bits are set to 0 or 1 with equal probability [11,49]. For a given bit array length l_q, the number of unique bit arrays that can be generated with half their bits set to 1 is $\binom{l_q}{l_q/2}$, where this number needs to be at least $3|\Sigma|^q$ in our case. We can calculate an estimation of this number based on Stirling's formula [15,29] as:

$$l_q^{est} = \left\lceil 2^{l_q} \Big/ \sqrt{\pi l_q/2} \right\rceil. \quad (1)$$

It holds that $l_q^{min} \leq l_q^{est}$, and Table 4 shows values for both l_q^{min} and l_q^{est} for alphabets of different sizes and different q-gram lengths. In the following, and in our implementation, we assume that the value of l_q has been calculated based on Equation (1).

Once the DOs have calculated the q-gram bit array length, l_q, to be used, they engage in a secure protocol [50] to find the maximum length l_s that corresponds

Algorithm 3: *Bit array generation by a DO*

```
Input:
- $\mathbf{D}$:   Database as one list of q-grams per unique string value      - $l_t$:   Final bit array length
- $\Sigma$:   Alphabet                                                   - $s$:    Common secret salt value
- $q$:   Q-gram length                                               - $s_d$:   Individual secret salt value
- $l_q$:   Q-gram bit array length
Output:
- $\mathbf{B}$:   Bit array inverted index
1:  $\mathbf{B} \leftarrow \{\}$                                        // Initialise inverted index of bit arrays
2:  $\mathbf{T}_\Sigma \leftarrow \{\}$                                       // Initialise table of q-gram bit arrays
3:  $\mathbf{T}_R \leftarrow \{\}$                                       // Initialise table of random bit arrays
4:  $\mathbf{Q}_\Sigma \leftarrow genQgramSet(\Sigma, q)$                  // Generate set of all possible q-grams from $\Sigma$
5:  for $q_\Sigma \in \mathbf{Q}_\Sigma$ do:                                        // Loop over q-gram in $\mathbf{Q}_\Sigma$
6:      $b_q \leftarrow genBitArr(\mathbf{T}_\Sigma, l_q, s, q_\Sigma)$  // Generate a unique bit array for each q-gram (see function below for details)
7:      $\mathbf{T}_\Sigma[q_\Sigma] \leftarrow b_q$                                   // Add q-gram bit array to table $\mathbf{T}_\Sigma$
8:  while $|\mathbf{T}_R| \leq |\mathbf{T}_\Sigma|$ do:          // Loop to generate table $\mathbf{T}_R$, to contain $|\mathbf{T}_\Sigma|$ random bit arrays
9:      $\mathbf{T}_T \leftarrow genRandBitArr(\mathbf{T}_\Sigma, l_q, s_d)$             // Generate temporal set of random bit arrays
10:     $\mathbf{T}_C \leftarrow setIntersectDOs(\mathbf{T}_T)$         // Find common random bit array generated by more than one DO
11:     $\mathbf{T}_T \leftarrow \mathbf{T}_T \setminus (\mathbf{T}_C \cup \mathbf{T}_R)$        // Remove random bit arrays also generated by other DOs from $\mathbf{T}_T$
12:     $\mathbf{T}_R.add(\mathbf{T}_T)$                                         Add to local random bit array of this DO
13: for $(sid, \mathbf{q}) \in \mathbf{D}$ do:                             // Loop over each q-gram list in the database
14:     $b'_q \leftarrow []$                                          // Initialise bit array for this q-gram list
15:     for $q_\Sigma \in \mathbf{q}$ do:                                      // Loop over q-grams in the q-gram list
16:         $b'_q \leftarrow b'_q + \mathbf{T}_\Sigma[q_\Sigma]$              // Concatenate the bit arrays of all q-grams in the q-gram list
17:     $l_r \leftarrow l_t - |b'_q|$                        // Calculate the number of random bits required for padding
18:     $b_f \leftarrow padRandBitArr(\mathbf{T}_R, b'_q, l_r)$   // Pad bit array at both beginning and end with a total of $l_r$ random bits
19:     $\mathbf{B}[sid] \leftarrow b_f$                      // Add the final bit array for q-gram list $\mathbf{q}$ to the inverted index $\mathbf{B}$
20: return $\mathbf{B}$
Function $genBitArr(\mathbf{T}_\Sigma, l_q, s, q_\Sigma)$:
21:     $b_q \leftarrow [0] \times l_q$                                // Initialise a bit array of 0-bits of length $l_q$
22:     $random.seed(q_\Sigma || str(s))$                  // Set PRNG seed value as q-gram concatenated with salt
23:     for $0 < i \leq l_q$ do:                                  // Loop over all positions of the bit array
24:         $b_q[i] \leftarrow random.select(0, 1)$            // Choose a 0 or 1 bit value randomly with equal probability
25:     while $b_q \in \mathbf{T}_\Sigma$ do:                               // Regenerate bit array until it is unique
26:         $s = s + 1$                                                         // Increase salt value
27:         $b_q = genBitArr(\mathbf{T}_\Sigma, l_q, s, q_\Sigma)$                 // Re-generate bit array using increased salt value
28:     return $b_q$
```

Table 4 Minimum and estimated lengths of bit arrays, l_q^{min} and l_q^{est}, for sizes of Σ (includes the two padding characters, α and β) and q-gram lengths, q.

		l_q^{min}			l_q^{est}		
Alphabet Σ	$\lvert\Sigma\rvert$	q=2	q=3	q=4	q=2	q=3	q=4
Genomics	4+2	7	10	12	10	12	16
Digits	10+2	9	13	16	12	16	20
Letters	26+2	12	17	21	14	20	24
Digits and letters	36+2	13	18	23	16	20	26

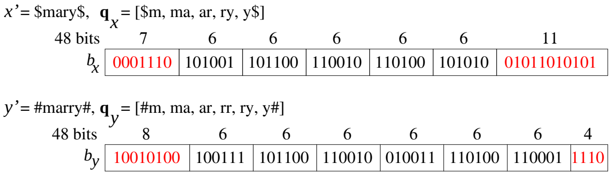


Fig. 2 Two example bit arrays, where each is of length $l_t = 48$ bits, with different random bit arrays padded at the beginning and end (shown in red).

to the longest q-gram list in their respective two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$. To ensure all q-gram lists can be padded with random bits both at the beginning and end, the DOs add 2 to the length l_s, and then calculate the final bit array length as $l_t = (l_s + 2) \times l_q$.

For example, as illustrated in Fig. 2, assume two padded strings are x' = "*\$mary\$*" and y' = "*#marry#*", and their corresponding q-gram lists are $\mathbf{q}_x = [\$m, ma, ar, ry, y\$]$ and $\mathbf{q}_y = [\#m, ma, ar, rr, ry, y\#]$, respectively. As shown in this figure, the longest q-gram list is $\mathbf{q}_y$ with length $l_s = 6$, where each q-gram bit array is of length $l_q = 6$ (to simplify visualisation). Thus, the final bit array length is $l_t = (6 + 2) \times 6 = 48$ bits.

Algorithm 3 outlines the bit array generation by the DOs. In line 1, each DO initialises the bit array inverted index, $\mathbf{B}$, to be used for the generated bit arrays that correspond to its unique strings. Each DO then initialises the tables $\mathbf{T}_\Sigma$ and $\mathbf{T}_R$, respectively, in lines 2 and 3. Each DO in line 4 generates the set of all possible q-grams, $\mathbf{Q}_\Sigma$, based on the agreed alphabet, Σ, and q-gram length, q. Then, in lines 5 to 7, the DO loops over each q-gram $q_\Sigma \in \mathbf{Q}_\Sigma$ to generate a bit array for this q-gram using the function $genBitArr()$. The details of this function are provided in lines 21 to 28.

In line 21, the function $genBitArr()$ first initialises a bit array of length l_q with only 0 bits. Then the q-gram q_Σ is concatenated with the secret salt value, s, that was agreed by the two DOs. This concatenated value is used as the seed to a pseudo random number generator (PRNG) [11]. With the same seed a PRNG will generate the same sequence of random outputs, and therefore all DOs generate the same random bit

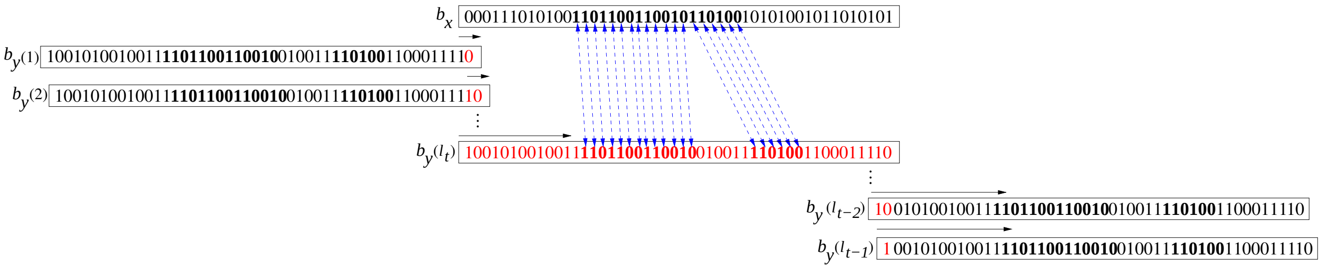

b_x 000111010100110110011001011010010101001011010101
$b_{y^{(1)}}$ 10010100100111101100110010010011110100110001111 0
$b_{y^{(2)}}$ 1001010010011110110011001001001111010011000111 10
⋮
$b_{y^{(l_t)}}$ 100101001001111011001100100100111101001100011110
⋮
$b_{y^{(l_{t-2})}}$ 10 0101001001111011001100100100111101001100011110
$b_{y^{(l_{t-1})}}$ 1 00101001001111011001100100100111101001100011110

Fig. 3 Comparison of bit arrays b_x and b_y using the basic comparison method. The numbers in brackets show the iteration number. The common bits are shown in bold and those bits that are compared between b_x and b_y in red. The blue lines show the bits that are common between b_x and b_y (we have two sequences of common bits, one 13 bits and the other 6 bits long).

arrays for the q-gram q_Σ. The loop in lines 23 and 24 will generate l_q such random bits, where the function $random.select(0, 1)$ returns a 0- or a 1-bit with equal probability. As a result, the bit array b_q should be filled with roughly 50% 1-bits. To ensure that each q-gram has a unique bit array, in line 25 we check this condition, and if required we generate a new bit array based on a changed secret salt value s. Because all DOs employ the same PRNG, they will generate the same bit arrays for the q-gram set $\mathbf{Q}_\Sigma$ (which is the same for all DOs). The function returns b_q in line 28, where $b_q \notin \mathbf{T}_\Sigma$.

Back to the main program, where in line 7, each DO inserts the generated bit array b_q into the inverted index $\mathbf{T}_\Sigma$, where the corresponding q-gram, q_Σ, is used as a key. Each DO repeats this process until one bit array, b_q, has been generated for every q-gram $q_\Sigma \in \mathbf{Q}_\Sigma$.

Each DO then generates in lines 8 to 12 its random bit array table, $\mathbf{T}_R$, where $|\mathbf{T}_R| = |\mathbf{T}_\Sigma|$ using the function $genRandBitArr()$. A temporary table of random bit arrays, $\mathbf{T}_T$, is generated first (in line 9), where the function $genRandBitArr()$ calls the function $genBitArr()$ (as described above) to generate each random bit array. Each DO uses its individual secret salt value, s_d, as its random seed. These individual secret salt values should result in different random bit arrays being generated by the DOs. However, to ensure no random bit array is generated by more than one DO, a secure set intersection protocol [18] is employed in line 10 between the DOs. Any found random bit array that has been generated by more than one DO will be returned by the $setIntersectDOs()$ function in the set $\mathbf{T}_C$ in line 10, and only those random bit arrays unique to a DO are then added to its list $\mathbf{T}_R$. The DO repeats the steps in line 9 to 12 until $|\mathbf{T}_R| = |\mathbf{T}_\Sigma|$.

In the last phase of the bit array generation process, each DO generates the final bit array, b_f, of length, l_t, for each string (assumed to be available as a q-gram list) in its database. Each DO first loops over the q-gram lists $\mathbf{q}$ in its database, $\mathbf{D}$, in line 13. For each $\mathbf{q}$, the DO initialises a q-gram bit array b'_q in line 14, and in line 15 loops over each q-gram, $q_\Sigma \in \mathbf{q}$. Each DO then selects the q-gram bit array, b_q, that corresponds to q_Σ from $\mathbf{T}_\Sigma$ and concatenates the selected b_q to the bit array, b'_q, in line 16.

Finally, to ensure the generated bit arrays for all q-gram lists $\mathbf{q} \in \mathbf{D}$ are of the same length of l_t, in line 17 we calculate the number of random bits, l_r, that are required for padding based on the length of the generated bit array b'_q. Using the function $padRandBitArr()$, in line 18 the DO then generates the final bit array, b_f, where a random number of bits are padded both at the beginning and end of b'_q such that a total of l_r bits are padded, and where these random bits are sourced from the list of random bits arrays, $\mathbf{T}_R$. We illustrated this random padding process in Fig. 2. Finally, in line 19 the DO inserts the final bit array b_f into the bit array inverted index $\mathbf{B}$ which will be sent to the LU for comparison, as we describe next.

5.2 Comparison of Bit Arrays

For a pair of bit arrays, b_x and b_y, as received from the DOs, the LU needs to find the longest consecutive sequence of bits that are the same across the two bit arrays. In the following we denote such a sequence as *common bits*, and the length of the longest such sequence as the length of *longest common bits* (LCB), *lcb*. We propose two algorithms for this comparison process. Similar to the comparison of hash encoded q-grams described in Section 4.4, the first algorithm is a basic algorithm that follows a naive comparison method, while the second algorithm is a fast algorithm which substantially improves the runtime of the comparison process.

5.2.1 Basic Bit Arrays Comparison Algorithm

Fig. 3 shows an example of the basic comparison algorithm between two bit arrays, b_x and b_y, where b_y is moved over b_x by one bit position per iteration. In each iteration, the LU compares the bit segment in the overlapping positions between b_x and b_y to find the longest sequence of common bits between the two segments.

Algorithm 4: ***Basic bit array comparison process by the LU***

```
Input:
- $b_x$:   First bit array
- $b_y$:   Second bit array
Output:
- $lcb$:   The number of bits in the longest common bit sequence

1:  $lcb \leftarrow 0$                                              // Initialise the length of the LCB
2:  $l_t \leftarrow |b_x|$                                          // Get length of bit array, $l_t$
3:  for $-(l_t - 1) \leq i \leq l_t - 1$ do:                        // Loop over index position $i$
4:      $x_s \leftarrow max(0, i)$                       // Set start position of $b_x$ to compare with $b_y$
5:      $x_e \leftarrow min(l_t - 1, i + (l_t - 1))$      // Set end position of $b_x$ to compare with $b_y$
6:      $y_s \leftarrow max(-i, 0)$                      // Set start position of $b_y$ to compare with $b_x$
7:      $y_e \leftarrow min(l_t - 1, l_t - (i + 1))$      // Set end position of $b_y$ to compare with $b_x$
8:      $b'_x \leftarrow b_x[x_s{:}x_e + 1]$       // Generate the bit segment, $b'_x$, of $b_x$ for comparison
9:      $b'_y \leftarrow b_y[y_s{:}y_e + 1]$       // Generate the bit segment, $b'_y$, of $b_y$ for comparison
10:     $c \leftarrow findCommon(b'_x, b'_y)$  // Find the length of the common bit array in the segment
11:     $lcb \leftarrow max(lcb, c)$  // Keep the maximum length of so far identified the length of LCB
12: return $lcb$
```

Algorithm 5: ***Fast bit array comparison process by the LU***

```
Input:
- $b_x$:   First bit array
- $b_y$:   Second bit array
- $l_\gamma$:   Segment length
Output:
- $lcb$:   The number of bits in the longest common bit sequence

1:  $lcb \leftarrow 0$                                                          // Initialise the length of the LCB
2:  $\mathbf{s}_y \leftarrow genSegment(b_y, l_\gamma)$                         // Generate list of segments of bit array $b_y$
3:  for $i \in (0, |\mathbf{s}_y| - 1)$ do:                        // Loop over each segment in the list of segments of $\mathbf{b}_y$
4:      $\mathbf{p}_x \leftarrow getPosMatch(\mathbf{s}_y[i], b_x)$         // Get the list of positions in $b_x$ where the segment $\mathbf{s}_y[i]$ occurs
5:      for $p_x \in \mathbf{p}_x$ do:                                                  Loop over all positions in the list $\mathbf{p}_x$
6:          $c_l \leftarrow getCommonLeft(\mathbf{s}_y, b_x[:p_x])$    // Find the length of the common bit array in the left side of the current position in $b_x$
7:          $c_r \leftarrow getCommonRight(\mathbf{s}_y, b_x[p_x:])$  // Find the length of the common bit array in the right side of the current position in $b_x$
8:          $c = c_l + |\mathbf{s}_y[i]| + c_r$                    // Calculate length of common bit sequence, where $|\mathbf{s}_y[i]| \leq l_\gamma$
9:          $lcb = max(lcb, c)$                                                 // Keep the maximum length of the LCB
10: return $lcb$
```

Algorithm 4 outlines the steps in the basic bit array comparison process. The LU first initialises the LCB to $lcb = 0$, and then obtains the length of the bit arrays b_x and b_y as $l_t = |b_x|$, where we assume $|b_x| = |b_y|$.

To compare the bit arrays b_x and b_y, from line 3 onwards the LU then loops over index position i, where $-(l_t - 1) \leq i \leq l_t - 1$. It calculates the start (x_s and y_s) and end (x_e and y_e) positions for the two bit segments in b_x and b_y to be compared, based on the value of i, where $0 \leq x_s \leq x_e < l_t$ and $0 \leq y_s \leq y_e < l_t$. In lines 8 and 9 the LU then generates the corresponding two bit segments, b'_x of b_x and b'_y of b_y.

In line 10, the LU uses the function $findCommon()$ to find the length of the LCB by applying the XOR operation on b'_x and b'_y and identifying the length of the longest consecutive sequence of 0-bits, which represents the LCB between b'_x and b'_y. In line 11 the LU then keeps the longest length so far identified the length of the LCB, and it repeats the steps in lines 3 to 11 for all positions i. Finally, the LU returns the found lcb to the DOs in line 12.

5.2.2 Fast Bit Arrays Comparison Algorithm

In the fast bit array comparison process, the DOs first agree on a segment length, l_γ, where $0 < l_\gamma \leq l_{lcb}$, and l_{lcb} is the minimum required length of LCB. The DOs can calculate $l_{lcb} = l_q \times (m - q + 1)$, where l_q is the q-gram bit array length, m is the minimum length of LCS required, and q is the agreed q-gram length. If two encoded strings share m consecutive characters, then they need to have a common bit sequence of at least length of l_{lcb}.

We use the segment length l_γ because it allows the LU to compare bit arrays one segment after another, which reduces the runtime required by the LU. Furthermore, the LU will not be able to learn any information about the original bit arrays that represent individual q-grams, even if the segment length $l_\gamma = l_q$, because it does not know l_q.

Algorithm 5 outlines the fast comparison process by the LU and Fig. 4 shows an example of this process on two bit arrays, b_x and b_y. As input, the LU receives b_x and b_y, and the segment length l_γ, from the DOs. In line 1, it initialises the length of LCB, lcb, and in line 2 it generates a list of segment, $\mathbf{s}_y$, from b_y using the function $genSegment()$ (as illustrated in Fig. 4 (a)).

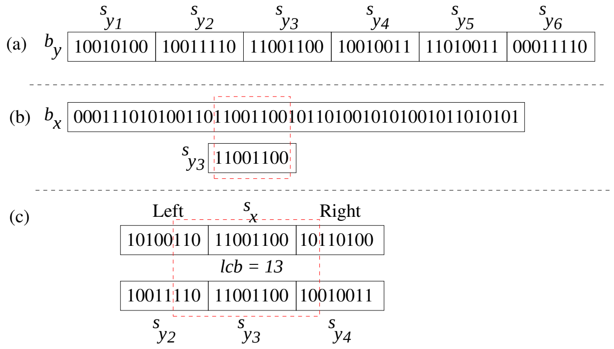


Fig. 4 The fast bit arrays comparison method between bit arrays b_x and b_y, where the bits in the red dashed box are the common bits, with the LCB set to $lcb = 13$. Figure (a) illustrates the segmentation of b_y into segments of length $l_\gamma = 8$ as done by the LU, (b) shows how the LU then finds the positions of the common bits, and (c) illustrates how the LU compares the segments to the right and left.

Each segment in $\mathbf{s}_y$ has a length of l_γ or less bits (last segment in the $\mathbf{s}_y$). The LU then loops over the segments in the list $\mathbf{s}_y$ in line 3, and for each segment in line 4 the LU finds the list of common positions, $\mathbf{p}_x$, in the bit array b_x where the segment $\mathbf{s}_y[i]$ occurs by using the function $getPosMatch()$, as shown in Fig. 4(b).

Because each bit array contains random and q-gram bit arrays, a given segment can contain bits from both. For each position, p_x, in the position list $\mathbf{p}_x$, the LU therefore needs to check if there are sequences of common bits between b_x and b_y both to the left and right of the common segment, because in either direction there can be further common bits, as is illustrated in Fig. 4(c). The LU uses the functions $getCommonLeft()$ and $getCommonRight()$ in lines 6 and 7 to find the number of common bits on the left and right, respectively, between the current segment in b_y, $\mathbf{s}_y[i]$, and bits in b_x. The LU calculates the current length of the common bit sequence in line 8, and checks if it is a new LCB, lcb, in line 9. The LU repeats the steps in lines 3 to 9 for all segments in $\mathbf{s}_y$. Finally, the LU returns the found lcb to the DOs in line 10.

6 LCS Length Calculation

As shown in Fig. 1, in the last step of our string matching approaches, using Equation (2) the DOs calculate the length of the LCS, lcs, based on the matching results they received from the LU. For the approach based on shifted hash encoded q-grams we discussed in Section 4, the DOs calculate the lcs based on the length of the longest sequence of common elements, lce, while if they use the approach based on bit arrays described in Section 5, they calculate the lcs based on the length of the longest sequence of common bits, lcb.

$$lcs = lce + q - 1 \quad \text{// For shifted hash encoded q-grams}$$
$$lcs = \lfloor lcb/l_q \rfloor + q - 1 \quad \text{// For bit array encoding} \qquad (2)$$

The DOs then only keep the string pairs that have a $lcs \geq m$. The last column in Table 3 shows examples of the calculated LCS length based on the lce for different pairs of strings.

7 Scalability Aspects

In this section we describe how we can scale our proposed string matching approaches to large databases. The number of string pairs increases quadratic with the numbers of strings in the two databases being compared. We can improve the complexity of the comparison process by applying a privacy-preserving blocking technique [11,20] to reduce the number of encoded string pairs that need to be compared by the LU.

We apply a q-gram based blocking approach [11] to generate blocks for each database. In this approach, each block is generated based on a permutation of q-grams in the q-gram list of each string. The DOs first agree on a secret salt value, s, a hash function, $\mathcal{H}$, and the length of q-gram set permutations, t_q. In our approach, we calculate t_q based on the agreed minimum length of the LCS, m, and the q-gram length, q (as described in Section 4.1) as $t_q = m - q + 1$.

The DOs then independently generate the q-gram permutation lists of length t_q for each of their q-gram lists. Each DO concatenates the q-grams in each such list into one string, q_s, which is used to generate a blocking key value, bkv, by concatenating it with the agreed secret salt value, s. This is followed by a hash encoding of this concatenated string, resulting in $bkv = \mathcal{H}(q_s \| s)$. Finally, all q-gram lists in a database that have the same bkv are inserted into the same block. Once the DOs have generated their blocks, they then send these blocks to the LU for conducting comparisons. The LU finds the common bkv between the received databases and only compares the encoded string pairs in blocks that have the same bkv.

For example, let us consider the DOs have agreed on $m = 3$ and $q = 2$, and therefore they calculate $t_q = 3-2+1 = 2$. We assume the two strings in the two databases, $x \in \mathbf{D}_A$ and $y \in \mathbf{D}_B$, are $x =$ "*mary*" and $y =$ "*marie*", with the q-gram lists $\mathbf{q}_x = [ma, ar, ry]$ and $\mathbf{q}_y = [ma, ar, ri, ie]$, respectively. They then individually generate the bkv of their strings as $\mathbf{bkv}_x = \{\mathcal{H}(maar\|s), \mathcal{H}(mary\|s), \mathcal{H}(arry\|s)\}$ and $\mathbf{bkv}_y = \{\mathcal{H}(maar\|s), \mathcal{H}(mari\|s), \mathcal{H}(maie\|s), \mathcal{H}(arri\|s), \mathcal{H}(arie\|s),$

$\mathcal{H}(riie\|s)\}$. The encoding of strings x and y are inserted into every block with the $bkv_x \in \mathbf{bkv}_x$ and $bkv_y \in \mathbf{bkv}_y$, respectively. Once the DOs have sent their blocks to the LU, the LU can find the common $bkv = \mathcal{H}(maar\|s)$. Therefore two encoded strings x and y are being compared.

In the random bit arrays based approach, we generate blocks by applying Hamming Locality Sensitivity Hashing (HLSH) [20,35]. In this approach, the LU receives two sets of bit arrays from the two DOs. The LU uses a set of hash functions to select certain bits, and it concatenates these bits into a bit array of fixed length, l_b, to be used as a bkv. The bit arrays that have the same bkv are then inserted into the same block.

In our approach, the DOs individually generate blocks of bit arrays before sending them to the LU. The DOs first agree on the secret salt value, s, a hash function, $\mathcal{H}$, and a bit percentage, p_b. They use p_b to calculate the length of a bit segment to be used for HLSH blocking as $l_b = (p_b \times l_{lcb})/100$, where we described l_{lcb} in Section 5.2.2. They then generate segments of the selected q-gram bit arrays, b'_q, each of length of l_b. Each of these segments is then used to generate a bkv by concatenating them with the agreed secret salt value, s, followed by a hash encoding using the function $\mathcal{H}$. The bit segments that have the same hash encoded bkv are inserted into the same block. However, the length of the b'_q is possibly not divisible by l_b, and therefore the last segment might be shorter than l_b. To ensure every generated segment has the same length, we therefore extend any segment that is too short by adding bits from the left segment. For example, if we assume $b'_q = 11001100$ (with $|b'_q| = 8$) and $l_b = 3$. The generated segments of this b'_q are 110, 011, 00. Therefore, the last segment, 00, is extended with the last bit from the second segment, resulting in the last segment becoming 100. The bkv of this b'_q are $\mathbf{bkv} = \{\mathcal{H}(110\|s), \mathcal{H}(011\|s), \mathcal{H}(100\|s)\}$.

8 Analysis of Our Protocol

We now analyse our approaches in terms of complexity, accuracy, and privacy.

8.1 Complexity Analysis

In the shifted hash encoded q-gram based approach, each DO requires $O(l_h)$ for each step of the encoding process, where l_h is the length of hash encoded q-grams list corresponding to each string in its database.

In the comparison process, let us assume the two shifted hash encoded q-gram lists, $\mathbf{h}_x$ and $\mathbf{h}_y$, are sent from the DOs to the LU, where these lists have the same length, l_h. In the basic comparison algorithm, the LU requires $O(l_h)$ for checking if $set(\mathbf{h}_x) \cap set(\mathbf{h}_y) \neq \emptyset$. For each common hash encoded q-gram $h_x \in \mathbf{h}_x$, the LU requires $O(l_h)$ to find the positions of h_x that occur in $\mathbf{h}_y$. It then requires $O(l_h^2)$ to find the length of the LCE, lce, between $\mathbf{h}_x$ starting from h_x (shifted $\mathbf{h}_x$) and every shifted list of $\mathbf{h}_y$. Therefore, overall the basic comparison algorithm requires $O(2l_h^2 + 2l_h)$.

In the fast comparison algorithm, the LU requires $O(l_h)$ for checking if $\mathbf{h}_x$ and $\mathbf{h}_y$ share elements. The LU then concatenates each list with itself and orders them, resulting in the shorter list $\mathbb{h}_s$ and longer list $\mathbb{h}_l$. The LU then requires $O(l_h)$ for finding the positions of consecutive common hash encoded q-grams in $\mathbb{h}_s$. For each $h_s \in \mathbb{h}_s$, it requires $O(l_h)$ to find the positions in h_s that occur in $\mathbb{h}_l$, and it requires $O(l_h)$ to find the lce between $\mathbb{h}_s$ starting from h_s and $\mathbb{h}_l$. Overall, the LU therefore requires $O(2l_h^2 + 2l_h)$. However, this is the worst case which only occurs when $\mathbf{h}_x$ and $\mathbf{h}_y$ contain exactly the same encodings. Otherwise, the LU requires less than $O(2l_h^2)$ to find the lce between $\mathbb{h}_s$ and $\mathbb{h}_l$. Therefore, the fast comparison algorithm is faster experimentally than the basic comparison algorithm, as we will show in Section 9.5.

In the random bit arrays based approach, each DO requires $O(|\Sigma|^q)$ to generate the bit array table of all possible q-grams, $\mathbf{T}_\Sigma$. To generate each random bit array, b_r, a DO checks if $b_r \notin \mathbf{T}_\Sigma \cup \mathbf{T}_{R_A} \cup \mathbf{T}_{R_B}$, where $\mathbf{T}_{R_A}$ and $\mathbf{T}_{R_B}$ are the random bit array tables of the two DOs. Each DO therefore requires a maximum $O(3|\Sigma|^q)$ to generate the random bit array table of size $|\mathbf{T}_\Sigma|$. To generate the final bit array, b_f, of each string, a DO requires $O(l_h)$ to concatenate the q-gram bit arrays, b_q, into a bit array, b'_q, and $O(n_r)$ to pad each b'_q, with random bit arrays, where n_r is the number of random bit arrays to be selected from $\mathbf{T}_R$.

We assume the LU receives two bit arrays, b_x and b_y, from the DOs. In the basic bit array comparison algorithm, the LU requires $O(l_t^2)$ to find the length of the LCB between b_x and b_y, where $l_t = |b_f|$. In the fast bit array comparison, the LU requires $O(l_t)$ to generate a list of segments, $\mathbf{s}_y$, from b_y. For each $s_y \in \mathbf{s}_y$, it then requires $O(l_t)$ to find the positions in b_x where each s_y occurs. For each position p_x in b_x, the LU requires $O(l_t)$ to find the sequence of common bits that occur to the left and right of bit at the position p_x in b_x. Therefore, the LU requires a total of $O(l_t + |\mathbf{s}_y|(l_t + l_t^2))$.

When the DOs apply blocking to their databases, each DO requires $O(l_h \times |\mathbf{D}|)$ to generate blocks based on q-gram based blocking, while with HLSH based blocking the DOs require $O(b_B \times |\mathbf{D}|)$, where $b_B = \lceil l_t/l_b \rceil$ and l_b is the length of bit segments used to generate a blocking key. In the comparison process, the LU requires $O(n^2/n_b)$ block comparisons, where n is the number of hash encoded q-gram lists or bit arrays in each database (we assume $|\mathbf{D}_A| = |\mathbf{D}_B|$) and n_b is the number of blocks.

8.2 Accuracy Analysis

As mentioned in Section 4, we use different padding characters between databases to ensure that the calculated length of LCS, lcs, is correct. Let us describe why this approach is required by using a q-gram list without padding characters as an example. We assume the strings to be compared are x = "*mary*" and y = "*marary*". The correct LCS between these two strings is "*mar*" with $lcs = 3$. We assume the DOs have agreed on $q = 2$ and they use random numbers for shifting their q-gram lists $r_x = 2$ and $r_y = 4$, respectively. Therefore, their shifted q-gram lists are $\mathbf{q}'_x = [\mathbf{ar}, \mathbf{ry}, \mathbf{ma}]$ and $\mathbf{q}'_y = [ar, ra, \mathbf{ar}, \mathbf{ry}, \mathbf{ma}]$, where the common q-grams are shown in bold. When the LU compares these lists, it returns the $lce = 3$ to the DOs. The DOs then use Equation (2) to calculate $lcs = 3 + 2 - 1 = 4$. Therefore, the DOs obtain an incorrect result. As this example shows, our approach does not work when strings are not padded using different characters. Examples of correct LCS calculations are shown in Table 3.

Apart from the padding characters, to calculate an accurate lcs, the minimum length of the LCS, m, must be at least of length q, $m \geq q$. This is because when $m < q$, in the shifted hash encoded q-gram based approach, the LU cannot find the length of LCE, lce, between the two hash encoded q-grams lists. Let us use the two q-gram lists, $\mathbf{q}_x$ and $\mathbf{q}_y$, as an example. We assume $m = 3, q = 4$, and two padded strings are x' = "\$*mary*\$" and y' = "#*marary*#". The corresponding q-gram lists of x' and y' are $\mathbf{q}_x = [\$mar, mary, ary\$]$ and $\mathbf{q}_y = [\#mar, arar, rary, ary\#]$, respectively. There is no common q-gram between these lists and therefore the DOs obtain the length of LCS, $lcs = 0$, for this pair of strings, where the actual LCS between x' and y' is "*mar*" with the $lcs = 3$. The same issue also occurs in the random bit arrays based approach because each bit array is generated based on a list of q-grams.

In the random bit arrays based approach, hash collisions [8], where two or more q-grams are encoded into the same q-gram bit array, b_q, can affect the accuracy of string matching. The probability of a hash collision, P_b, that the bit can be set to 1 in this approach can be calculated by applying the dependent probability calculation [1] as shown in Equation (3):

$$P_b = \frac{l_q/2}{l_q} \times \frac{(l_q/2)-1}{l_q-1} \times \ ... \ \times \frac{1}{(l_q/2)+1}, \qquad (3)$$

where l_q is the length of the bit array of each q-gram, and $l_q/2$ means 50% of l_q is set to 1. When selecting the first bit position, there is a $l_q/2$ out of l_q chance that the position is being selected to be set to 1 by two or more q-grams. The number of chances decreases by 1 once each position is selected. Finally, when selecting the last position, there remain 1 out of $l_q/2+1$ chances that a position can be selected. For example, assume we use $l_q = 6$, the probability that a hash collision can occur is $P_b = 3/6 \times 2/5 \times 1/4 = 0.05$ or 5% of l_q.

8.3 Privacy Analysis

We assume the DOs and the LU follow the *honest-but-curious* (HBC) adversary model where no DO colludes with the LU [37]. The HBC model is commonly used in PPRL and private string comparison protocols [59] because of its applicability to real scenarios. In this model, each party tries to learn as much as possible about the other parties' data based on what it receives from the other parties, while following the protocol steps.

In our approaches, the DOs first communicate with each other to agree on parameter settings. This allows them to learn the parameters that are being used in the encoding processes but they cannot learn any sensitive information of the strings in each other's databases.

In both approaches, the DOs then individually encode the unique strings in their databases using the agreed parameters without learning any information from the other database. However, to generate the random bit arrays in the random bit arrays based approach, the DOs employ a secure set intersection protocol [18] to find and exclude the common random bit arrays from their random bit arrays tables. These random bit arrays however do not represent any actual q-grams, and therefore the DOs do not learn any sensitive information from each other.

When blocking is used, the DOs apply a privacy-preserving blocking algorithm [11,20] on their encoded strings before these are being sent to the LU. We assume such a blocking algorithm to be secure such that it does not allow the DOs to learn any sensitive information about each other's databases. The LU then receives the two encoded databases from the DOs. It first finds common blocks between them and then compares only encoded strings that are in the same blocks. In this step, the LU does learn which encodings occur in both databases, but not their actual content.

In our shifted hash encoded q-gram based approach, the LU can learn the string length by guessing the length of q-grams, q (commonly used values are 2 and 3), and checking the length of the hash encoded q-gram lists. The LU can then generate q-grams from a public database using the guessed q and compare the frequency of the generated q-grams and the hash encodings in a received database. However, in order to identify encoded q-grams, this public database must contain a very similar set of values with the same frequency distribution to the encoded database as otherwise the LU cannot employ a frequency analysis. Furthermore, an injection of faked values can be used to prevent such a frequency based attack [33].

In the basic encoded q-gram comparison algorithm (described in Section 4.4.1), for each pair of shifted hash encoded q-gram lists, the LU finds the length of LCE by iteratively comparing and rotating the two lists. While the LU can keep the positions where the common hash encoded q-grams occur, it cannot learn the actual positions of these common hash encoded q-grams nor the positions of the original q-grams because (1) the common q-grams between two lists can occur at any position in the lists, and (2) the hash encoded q-grams in the two lists have been shifted by our random shifting technique. This results in the common patterns of the original string pairs to be distributed to different patterns of the encoded string pairs. In other words, the original positions where common q-grams occur in the q-gram lists have been shifted to other positions in the encoded and shifted lists.

Similarly, in the fast encoded q-gram comparison algorithm, although the actual sequence of hash encoded q-grams is contained in the concatenation of the shifted lists (as described in Section 4.4.2), the LU still cannot learn the actual positions of neither where the original q-grams nor the LCS occur in the two strings.

In the random bit arrays based approach, the LU receives bit arrays which are randomly padded by random bits. The LU cannot learn the length of the original strings because every bit array has the same length. It also cannot learn the frequency distribution of the bits that encode each q-gram because the q-gram bit arrays are shifted by a random number of bits, and therefore it cannot re-identify the original q-grams. The LU can only learn that the common bits occur in the middle of two bit arrays (common pattern *m-m*) but it cannot learn the actual positions where the LCS occurs in the strings that correspond to a bit array pair.

Once the LU has compared all encoded string pairs, it returns the LCE or LCB to the DOs. Each DO then calculates the length of the LCS, lcs. This allows each DO to learn the LCS between a string in its database and a string in the other DO's database, but the DO cannot learn the positions where the LCS occurs in their string. Therefore, the DOs only learn that there is a sub-string match.

9 Experimental Evaluation

We evaluated the accuracy, privacy, and scalability of our privacy-preserving string matching approaches compared to Bloom filter (BF) encoding [45], tabulation based hashing (TMH) [52], and DGK approximate string matching (DGK) [24]. We compared our approaches with these three baselines because BF encoding [45] is considered as a standard technique for PPRL, TMH [52] is a more secure technique compared to BF encoding, and DGK [24] is a recently proposed approach for secure string matching that encrypts strings based on their q-grams. We implemented all approaches using Python 2.7 and ran experiments on a server with 2.4 GHz CPUs running Ubuntu 18.04.

9.1 Data Sets

In our evaluation, we require pairs of data sets where each pair contains different common patterns as illustrated in Table 3. We used both synthetic and real data of types numbers, letters, and mixed.

To generate the synthetic data, we used the Python package *Faker*[1] to create data sets of credit card, barcode, and IBAN (International Bank Account Number) numbers, where each such data set contains 1,000 unique strings. We used each of these data sets as the first data set in a pair. We then created the second data set of a pair by replacing characters at different positions in each string in the first data set by random characters of the same alphabet. We ensured each data set pair does contain different common patterns, however the common pattern *b-e* (see Table 3) cannot occur for IBAN numbers because these numbers begin with letters and end with digits. Therefore, the beginning of the first IBAN number cannot be in common with the end of a second IBAN number in a string pair.

For real data, we extracted 1,000 strings of first names, cities, zip codes, and telephone numbers from the North Carolina Voter Registration (NCVR)[2] database, with snapshots from 2011 (first data set) and 2020 (second data set). We also extracted 1,000 strings of US security numbers from the Social Security Death Master File[3]. Similar to generating the synthetic data sets, we used the extracted US security numbers to generate the second data set by randomly replacing characters at different positions in each string.

In total we evaluated our approaches and baselines on eight data set pairs including three sets of synthetic

[1] See: `https://pypi.org/project/Faker/`

[2] See: `https://dl.ncsbe.gov`

[3] See: `http://ssdmf.info/download.html`

Table 5 Lengths of the longest q-gram list, l_s, q-gram bit array, l_q, calculated using l_q^{est} (Equation (1)), and final bit array, l_t, of different data set pairs and alphabet sizes, $|\Sigma|$.

Data set pair	Data set	$\|\Sigma\|$	l_s	l_q	l_t
NCVR 2011-2020	First names	28	15	20	300
”	Cities	31	22	20	440
”	Zip codes	10	11	16	176
”	Telephone numbers	11	11	20	220
Extracted-Corrupted	US security numbers	26	14	20	280
Synthetic-Corrupted	Credit card numbers	10	17	20	340
Synthetic-Corrupted	Barcode numbers	10	14	20	280
Synthetic-Corrupted	IBAN numbers	36	23	26	598

Table 6 Number of bits used for generating blocks and *bkv* for different data sets and approaches.

Data set	Our approach			BF	TMH
	p_b=30	p_b=50	p_b=80		
NCVR First names	6	10	16	16	16
NCVR Cities	6	10	16	16	16
NCVR Zip codes	4	8	12	12	12
NCVR Telephone numbers	6	10	16	16	16
US security numbers	6	10	16	16	16
Credit card numbers	6	10	16	16	16
Barcode numbers	6	10	16	16	16
IBAN numbers	7	13	20	20	20

data, four sets of real NCVR data, and one set of real social security death index data.

9.2 Parameter Settings

We padded strings in the two databases, $\mathbf{D}_A$ and $\mathbf{D}_B$, using the padding characters α = “\$” and β = “#”, respectively. We generated q-grams using $q = 3$ for first names, cities, zip codes, and the US security numbers data sets, while we used $q = 4$ for telephone, credit card, barcode, and IBAN numbers. For each data set, we used the minimum length of LCS, $m = q$.

In the shifted hash encoded q-gram based approach, to generate hashed q-grams, we used the hash function $\mathcal{H} = \mathit{SHA256}$ [44] and the agreed secret salt value $s = 45$. This salt value was also concatenated with q-grams for generating each q-gram bit array, b_q, in the bit arrays based approach. To generate the random bit arrays for databases $\mathbf{D}_A$ and $\mathbf{D}_B$, we used the individual secret salt values, $s_A = 65$ and $s_B = 56$, respectively. We calculated the length of q-gram bit arrays, l_q, using Equation (1). Table 5 shows the alphabet sizes and bit array length for each data set.

We compared our approaches with three baselines, which are BF [45], TMH [52], and DGK approximate string matching (DGK) [24]. We used the same parameter settings as we used in our approaches, such as padding character α, q-gram length q, minimum length m, secret salt value s, and the hash function $\mathcal{H}$.

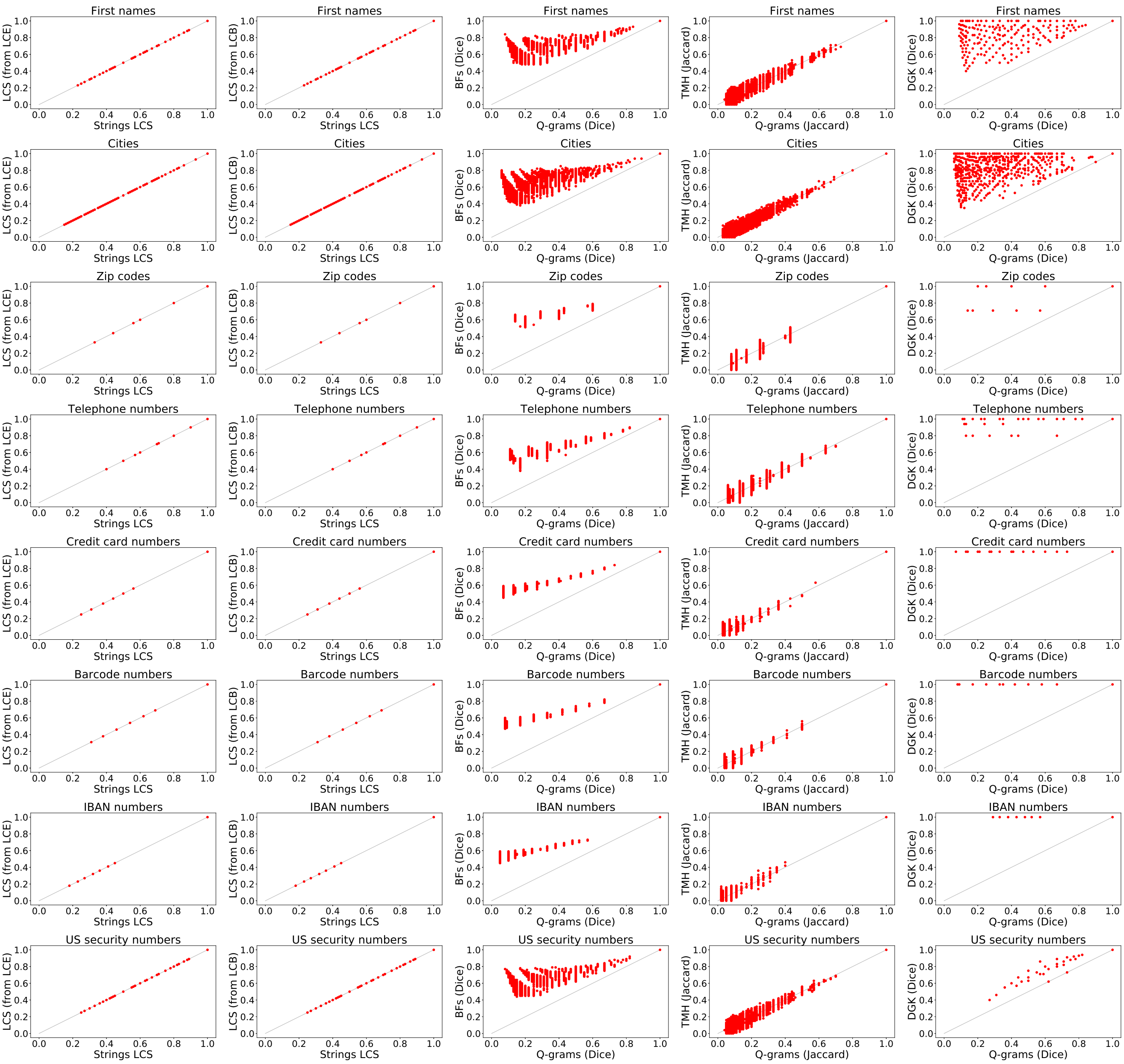
To generate the BF for a string, we encoded each q-gram set into a BF of 1,000 bits as this is a commonly used BF length for PPRL [45]. We used the optimal number of hash functions [39] for each data set, which is 139 for first names, 87 for cities, 139 for zip codes, 87 for telephone numbers, 116 for US security numbers, 46 for credit card numbers, 58 for barcode numbers, and 33 for IBAN numbers. For the TMH approach, we followed the original publication [52] and used 8 tabulation hash keys each of 64 bits length to generate a bit array of length 1,000 bits to encode a string. For the DGK approach, we used keys of size 1,024 bits, and rather than using a two-party protocol as proposed in the original publication [24], we implemented a three-party protocol to be comparable with our approaches and the other two baselines by using a LU for conducting the comparison process.

To improve scalability, we applied a q-gram based blocking technique for all approaches and applied HLSH based blocking on the random bit array, BF, and TMH approaches, as we described in Section 7. However, we only show results based on q-gram based blocking for BF [45] and TMH [52] as both blocking approaches (q-gram and HLSH based blockings) provide highly similar results. For q-gram based blocking, we calculated the length of q-gram set permutations for generating blocks based on m and q, resulting in $t_q = 1$.

For HLSH based blocking, in our random bit arrays based approach, we generated blocking key values, *bkv*, using the length of bit segments l_b calculated based on the bit percentage, p_b = 30, 50, and 80 (the l_b calculation is described in Section 7). For the BF [45] and TMH [52] baselines, we used the same length of bit segments l_b calculated based on $p_b = 80$ in our approach because when using $p_b < 80$ the resulting bit segments are too short and generate too many blocks for BFs or bit arrays of length 1,000 bits. This would lead to a large number of comparisons. For example, as shown in the first names data set in the Table 6, the length of bit segments $l_b = 6$ and number of blocks is 166 blocks when it is calculated based on $p_b = 30$, while $l_b = 16$ and number of blocks is 62 when calculated based on $p_b = 80$. To generate each blocking key, *bkv*, we used the agreed secret salt value, $s = 45$, and the hash function $\mathcal{H} = \mathit{SHA256}$.

9.3 Accuracy Results

We evaluated the accuracy of all approaches based on the correctness of similarity calculations. We compared the length of the LCS of unencoded string pairs with the calculated length of LCS, *lcs*, of the corresponding encoded string pairs based on Equation (2). To be compa-



First names

Cities

Zip codes

Telephone numbers

Credit card numbers

Barcode numbers

IBAN numbers

US security numbers

Fig. 5 Similarity plots of shifted hash encoded q-gram based approach (first column), shifted random bit arrays based approach (second column), Bloom filter (BF) encoding [45] (third column), tabulation based hashing (TMH) [52] (fourth column), and the DGK approximate threshold [24] based approach (last column). As can be seen, both of our approaches provide accurate similarity calculations our LCS equals the actual LCS that is calculated on unencoded string pairs), while the BF and TMH approaches both can lead to substantially changed similarities even between very similar strings. The DGK approach results in the similarity of a pair of encoded strings to be higher than the similarity of its corresponding unencoded strings.

rable with the BF [45], TMH [52], and DGK [24] baselines, we normalised the lcs into the range [0...1] of similarity values, calculated as $sim_{lcs} = lcs/max(|x|, |y|)$, where x and y are the strings in a pair.

For the BF approach, we calculated the similarity of q-gram sets and of BFs using the Dice co-efficient similarity [45], while we used the Jaccard similarity calculation for q-gram sets and of bit arrays generated by the TMH encoding technique [52]. For the DGK approach, we calculated the similarity of q-gram and encryption (ciphertext) lists using the Dice co-efficient [24].

Fig. 5 shows scatter plots where the horizontal axis shows unencoded similarities and the vertical axis shows the corresponding encoded similarities. Points on the diagonal show pairs of strings where both the unencoded and the encoded similarities are the same, while any point off the diagonal shows differences in the calculated similarities between unencoded and encoded

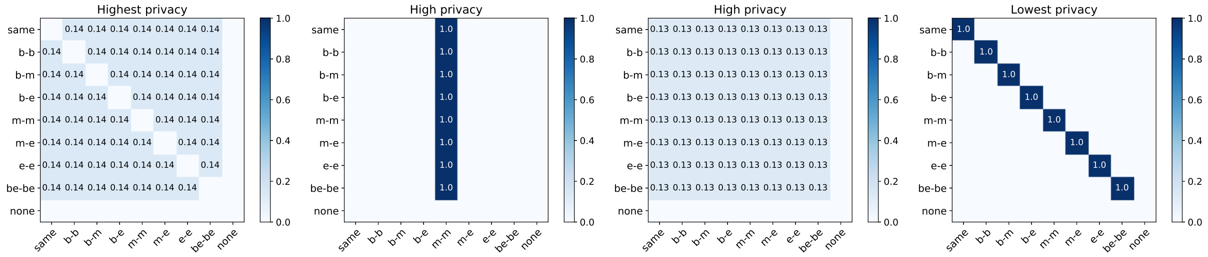


Fig. 6 Heatmap [22] plots of different privacy levels of a PPRL approach. The highest to lowest privacy are shown from left to right. In each plot, the vertical axis shows the common pattern of unencoded (or unencrypted) string pairs and the horizontal axis shows the common pattern of encoded (or encrypted) string pairs.

string pairs. As can be seen, both our approaches provide accurate string similarity results, while BF [45] and TMH [52] encodings can result in inaccurate similarities. This is because of high number of hash collisions that occur with both encoding approaches, where different q-grams are hashed into the same bit positions.

The DGK [24] approach also results in inaccurate similarities. This is because the Dice co-efficient of the ciphertexts is calculated based on the cardinality, where some ciphertexts that represent encrypted q-grams of strings in a pair are not common, although these ciphertexts are common between the two lists of all possible q-grams that were used to generate the intersection set of cardinality.

9.4 Privacy Results

We first evaluated the privacy of our approaches by identifying the common patterns between unencoded (or unencrypted) and encoded (or encrypted) string pairs. Fig. 6 shows example heatmap [22] plots of different levels of privacy provided by a PPRL approach, where in each plot the vertical axis shows the common patterns of unencoded (or unencrypted) string pairs and the horizontal axis shows the common patterns of encoded (or encrypted) string pairs. The dark blue colour indicates a higher percentage of unencoded and encoded string pairs while lower percentage of pairs are shown in light blue.

An approach provides the highest privacy when there are no common patterns of unencoded and encoded string pairs (as we show in the leftmost plot in Fig. 6). In contrast to the highest privacy, an approach provides the lowest privacy when all patterns of unencoded and encoded string pairs are in common (as we show in the rightmost plot in Fig. 6). An approach provides high privacy when (1) the patterns of unencoded string pairs become the *m-m* pattern when the strings in these pairs are encoded (as illustrated in the second left plot), or (2) the patterns of unencoded string pairs become different patterns when the string in these pairs are encoded (as illustrated in the second right plot). Therefore, when common patterns of unencoded string pairs become different patterns in encoded string pairs, it is more difficult for an adversary to identify the original q-grams and the positions where the LCS occurs.

As can be seen in Fig. 7 and Fig. 8, the common patterns of encoded string pairs using our approaches are different from the common patterns of unencoded string pairs, where no string pair has the *same* common pattern. With the shifted hash encoded q-grams based approach, each common pattern of unencoded string pairs is distributed to different common patterns when the strings in the pairs are encoded. The highest number of encoded string pairs in many data sets is the common pattern *m-m*, which means the encoded strings in a pair are common in the middle of the hash encoded q-gram list, **h**. Similarly, in the random bit array based approach, most common patterns of unencoded string pairs become the common pattern *m-m* in the encoded string pairs, as we described in Section 8.

For the DGK approach [24], the common patterns of unencrypted and encrypted string pairs are all the same because the common patterns of unencrypted string pairs are not distributed to other common patterns of encrypted string pairs. Each unencrypted q-gram in a pair and its corresponding ciphertext (encryption of 1) is located at the same position in the list of all possible q-grams. However, this approach still provides high privacy because of the use of the DGK homomorphic encryption [17] which results in the same value being encrypted into different ciphertexts. Therefore, although the common patterns of unencrypted and encrypted string pairs are the same, it would be difficult to re-identify the original string values because an adversary cannot learn if a ciphertext represents a 0 or 1.

For the BF [45] and TMH [52] encoding approaches, the common patterns of the same string value in different data sets are not being distributed to other common patterns when they are encoded, and therefore the common pattern is the *same* common pattern. The en-

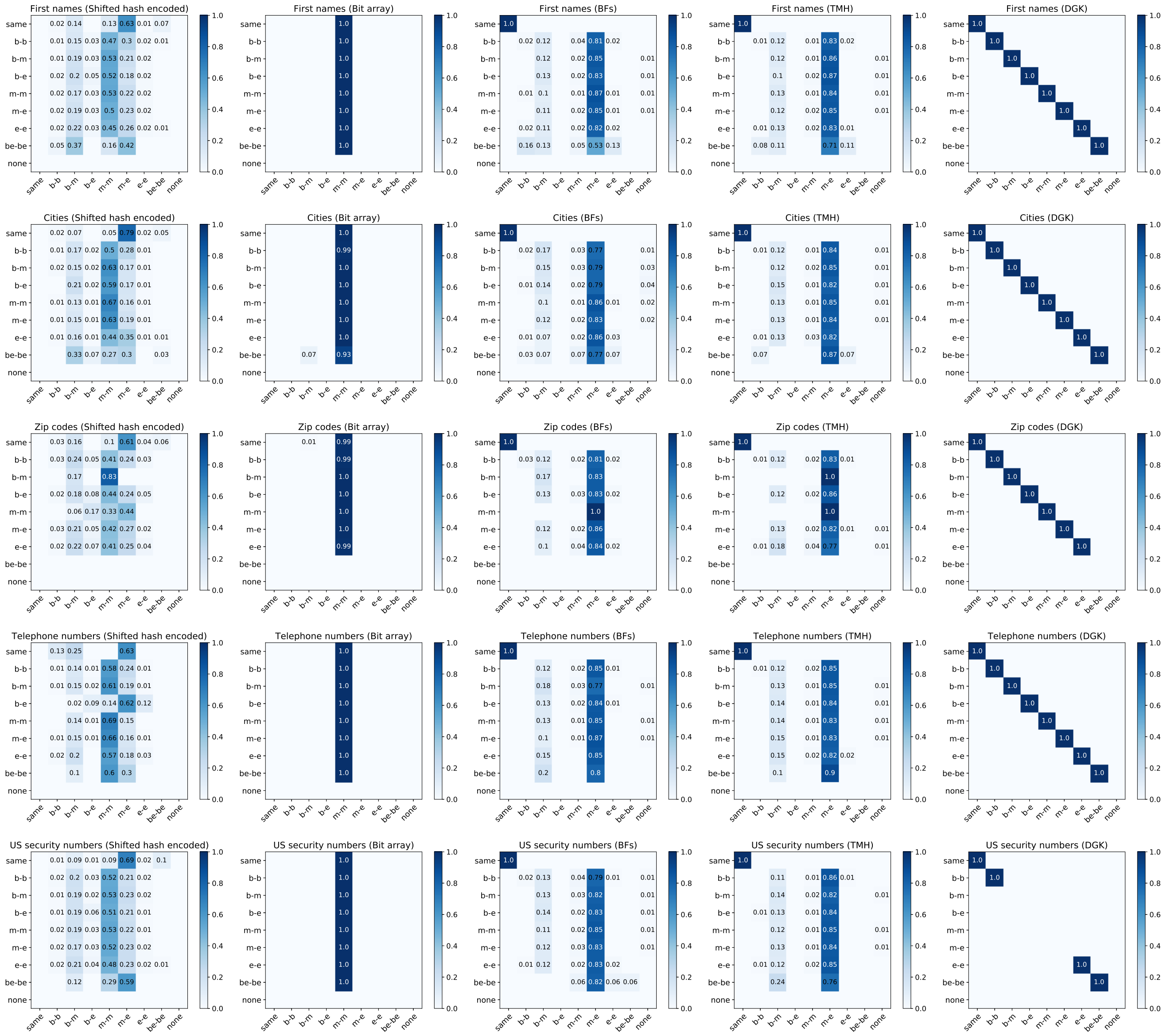


Fig. 7 Heatmap [22] plots of the NCVR and US social security number data set that are compared using different approaches. Each column shows shifted hash encoded q-gram, random bit arrays, BF encoding [45], TMH [52], and DGK [24] ordered from left to right. Each row shows common patterns of different real data sets. In each plot, the vertical axis shows the common pattern of unencoded (or unencrypted) string pairs and the horizontal axis shows the common pattern of encoded (or encrypted) string pairs. Higher percentages of unencoded and encoded string pairs are shown in dark blue, while lower percentages are shown in light blue colour.

coded string pairs can have the *none* common pattern because when using these approaches the bits that encode q-grams are not located in sequential order. The bits of common q-gram between two strings can be located next to bits that encode not common q-grams, and the sequence of bits in a BF or TMH bit array is then a mix of common and not common q-grams. For example, assume the two BFs b_x = 10**1**100**0**100 and b_y = 00**0**101**0**110 have common bits encoding of common q-gram locating at positions 3 and 7 (as shown in bold) of the BFs. These two bits are located next to the bits encoded of not common q-grams. The encoding is a mix of bits encoded of common and not common q-grams. As can be seen, this BF pair cannot be categorised to any of common patterns (as illustrated in Table 3), and therefore, the common pattern of this BFs pair is *none*.

We also evaluated the privacy of our random bit arrays approach and the BF [45] and TMH [52] encoding baselines using two cryptanalysis attacks developed for BFs for PPRL [12,13,14]. A frequency based attack [12, 13] cannot reveal any information from our random bit

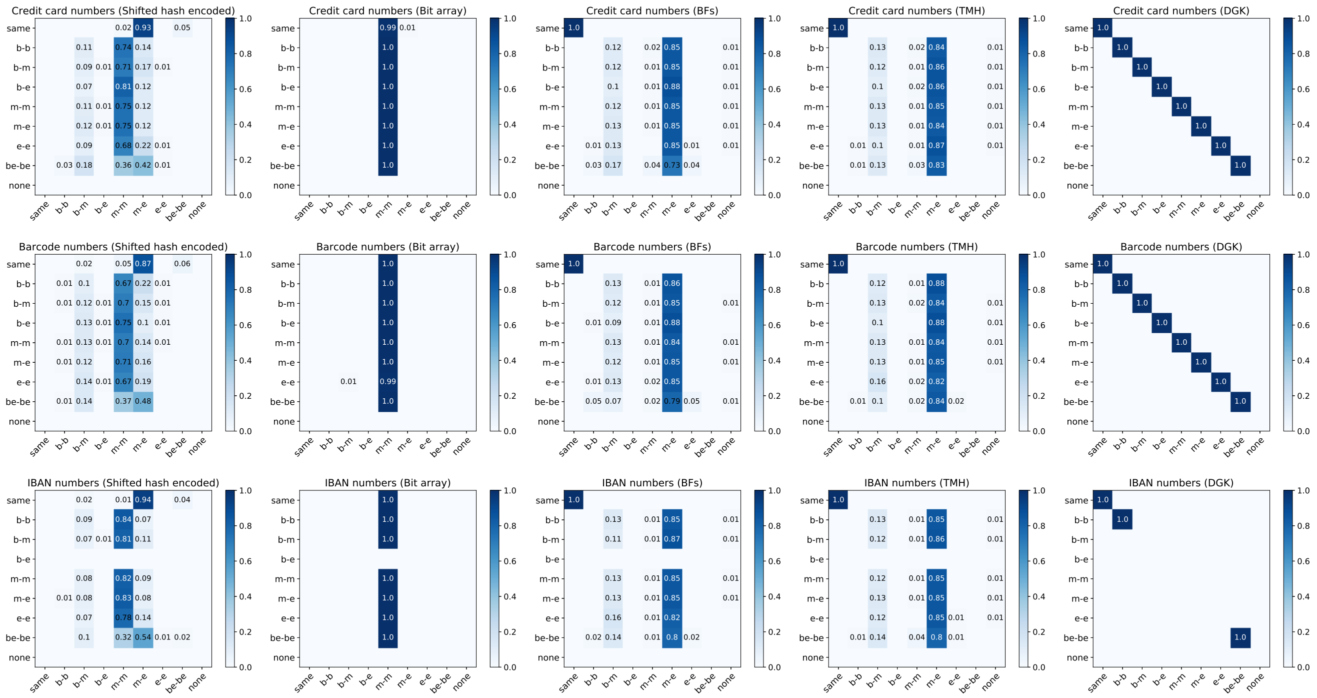


Fig. 8 Heatmap [22] plots of the synthetic data sets that are compared using different approaches. Each column shows shifted hash encoded q-gram, random bit arrays, BF encoding [45], TMH [52], and DGK [24] ordered from left to right. Each row shows common patterns of different synthetic data sets. The vertical axis shows the common pattern of unencoded (or unencrypted) string pairs and the horizontal axis shows the common pattern of encoded (or encrypted) string pairs. Higher percentages of unencoded and encoded string pairs are shown in dark blue, while lower percentages are shown in light blue colour.

array based approach as well as the two baselines because the frequency of bit arrays or BFs equals the frequency of strings (all have frequency of 1). Therefore, the attack cannot identify any pairs of unencoded and encoded values. A pattern mining based attack [14] cannot re-identify any information in our random bit array based approach and the two baselines either, because of the random bit arrays which result in encodings of the same q-gram in different strings being located at different positions. It also cannot attack the two baselines because too many hash collisions occur in encodings which means the attack cannot re-identify any information about individual q-grams.

9.5 Scalability Results

To be comparable between our approaches and the baselines, we use a three-party protocol for all approaches [11]. We evaluated the runtime of the encoding process by a DO and the string comparison process by the LU, as shown in Fig. 9. We report the average times for one string or string pair in milliseconds.

As can be seen in Fig. 9, our shifted hash encoded q-gram based approach is the fastest encoding technique while the DGK approach [24] is the slowest encoding technique. In our random bit arrays based approach, the encoding of letters is performed faster than the encoding of numbers. This is because the size of the alphabet, $|\Sigma|$, affects the runtime when generating the unique random bit arrays. A small $|\Sigma|$ leads to shorter q-gram bit array length, l_q, and results in longer runtimes to generate unique random bit arrays for the two DOs. Furthermore, encoding also uses more time for longer strings, such as IBAN numbers (as shown in Table 5). However, the size of the alphabet and the length of strings do not affect the other encoding approaches.

For the comparison process, we applied q-gram based blocking to our shifted hash encoded q-gram based approach and the three baselines, while we applied HLSH based blocking to our random bit array based approach and the BF [45] and TMH [52] baselines. Table 7 shows the number of string pair comparisons of the different data set pairs and approaches, where we show only the number of comparisons based on q-gram based blocking for the three baselines.

As can be seen in Fig. 9, in the comparison process, our approaches consume similar runtimes to the DGK approach [24] and have longer runtimes than BF [45] and TMH [52] encoding, where these two baselines have similar runtimes. This is because the comparison process of our approaches is more complicated, where we find all sequences of common encodings that occur in

Table 7 Number of string pair comparisons for different data set pairs and encoding approaches.

Data set pair	Data set	Shifted hash encoded q-gram	Bit array based	BF [45]	TMH [52]	DGK [24]
NCVR 2011-2020	First names	19,462	20,573	45,215	45,215	124,246
”	Cities	33,029	33,662	46,954	46,954	168,494
”	Zip codes	55,454	59,515	389,603	389,603	396,020
”	Telephone numbers	176,863	177,372	573,053	573,053	574,920
Extracted-Corrupted	US security numbers	12,884	13,175	29,625	29,625	81,024
Synthetic-Corrupted	Credit card numbers	18,594	22,153	27,606	27,606	27,606
Synthetic-Corrupted	Barcode numbers	9,864	12,139	11,706	11,706	11,706
Synthetic-Corrupted	IBAN numbers	22,818	23,156	112,872	112,872	112,872

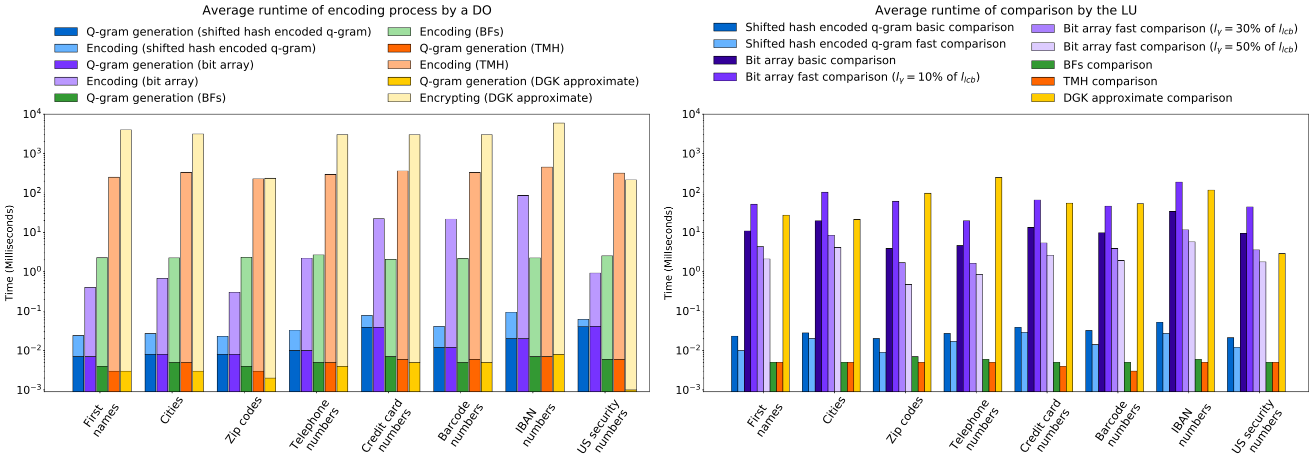


Fig. 9 Runtime comparison of the encoding processes by a DO (left) and encoded string comparison by a LU (right) between our approaches, BF encoding, and TMH. Shown are average times for encoding one string and matching one string pair.

the encoded strings pair and then find the LCS between them, while BFs [45] and TMH [52] both only calculate approximate similarities based on the set intersection of 1-bits that occur in a pair of encoded strings.

Overall, as expected, the runtimes of our fast comparison algorithms are faster than the basic comparison algorithms. However, in the random bit arrays based approach, the fast algorithm is slower than the basic algorithm when we use $l_\gamma = 10\%$ of l_{lcb}. This is because of the overhead by the fast algorithm which needs to generate segments and find the common sequences of bits to the left and right of segments.

In terms of privacy, the common patterns of the original string pairs are distributed to different patterns when strings are encoded using our approaches, while with the BF, TMH, and DGK baselines the common patterns of string pairs are not distributed to other common patterns. This implies that our approaches will make it more difficult for an adversary to re-identify the original string pairs based on a frequency analysis than with the three baselines because less common patterns are available for an attack. Overall, our approaches provide high accuracy and privacy, at the cost of increased comparison times if compared to the three baselines.

10 Discussion

Our approaches provide accurate string comparisons and outperform Bloom filter (BF) encoding [45], tabulation based hashing (TMH) [52], and the DGK approximate string matching (DGK) [24] approaches, where all of these baselines calculate approximate similarities between string pairs. Our approaches use more time for the comparison step than the BF [45] and TMH [52] baselines, while our random bit arrays based approach uses similar runtimes to the DGK [24] approach. For the encoding step, our approaches are faster than the TMH [52] and DGK [24] baselines.

11 Conclusions and Future Work

We have presented two new privacy-preserving string matching techniques that allow the accurate and efficient calculation of the longest common sub-string between strings. Our approaches encode sensitive input strings such that no re-identification is possible, while also preventing frequency attacks on individual character encodings. Our experimental evaluation has shown that both our approaches result in the same string similarities as on the original unencoded strings, while commonly used Bloom filter encoding [45], tabulation based hashing [52], and DGK approximate string matching [24] approaches will lead to potentially much higher or lower similarities between encoded strings.

As future work we aim to improve the runtime of the comparison step of our random bit arrays based approach by generating blocks based on the consecutive order of bit segments, and conduct more extensive scalability experiments on larger databases.

Conflict of interest

On behalf of all authors, the first author states that there is no conflict of interest.

Acknowledgements This work was partially funded by the Australian Research Council (ARC) under DP160101934. The authors like to thank Kee Siong Ng for helpful discussions.

References

1. Ash, R.B.: Basic probability theory. Courier Corporation (2008)
2. Benford, F.: The law of anomalous numbers. Proceedings of the American philosophical society pp. 551–572 (1938)
3. Bezawada, B., Liu, A.X., Jayaraman, B., Wang, A.L., Li, R.: Privacy preserving string matching for cloud computing. In: 2015 IEEE 35th International Conference on Distributed Computing Systems, pp. 609–618. IEEE (2015). DOI 10.1109/ICDCS.2015.68
4. Bonomi, L., Xiong, L., Chen, R., Fung, B.C.: Frequent grams based embedding for privacy preserving record linkage. In: ACM Conference on Information and Knowledge Management, pp. 1597–1601 (2012)
5. Broder, A.Z.: On the resemblance and containment of documents. In: Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171), pp. 21–29. IEEE (1997). DOI 10.1109/SEQUEN.1997.666900
6. Chase, M., Shen, E.: Pattern matching encryption. IACR Cryptol. ePrint Arch. **2014**, 638 (2014)
7. Chen, F., Wang, D., Li, R., Chen, J., Ming, Z., Liu, A.X., Duan, H., Wang, C., Qin, J.: Secure hashing-based verifiable pattern matching. IEEE Transactions on Information Forensics and Security **13**(11), 2677–2690 (2018). DOI 10.1109/TIFS.2018.2825141
8. Chi, L., Zhu, X.: Hashing techniques: A survey and taxonomy. ACM Computing Surveys (CSUR) **50**(1), 1–36 (2017). DOI 10.1145/3047307
9. Christen, P.: Data Matching. Springer, Heidelberg (2012). DOI 10.1007/978-3-642-31164-2
10. Christen, P.: Preparation of a real voter data set for record linkage and duplicate detection research. Australian Nat. Univ., Canberra, Australia (2013)
11. Christen, P., Ranbaduge, T., Schnell, R.: Linking Sensitive Data: Methods and Techniques for Practical Privacy-Preserving Information Sharing. Springer International Publishing AG (2020). DOI 10.1007/978-3-030-59706-1
12. Christen, P., Ranbaduge, T., Vatsalan, D., Schnell, R.: Precise and fast cryptanalysis for Bloom filter based privacy-preserving record linkage. Transactions on Knowledge and Data Engineering (2018)
13. Christen, P., Schnell, R., Vatsalan, D., Ranbaduge, T.: Efficient cryptanalysis of Bloomfilters for privacy-preserving record linkage. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 628–640. Springer (2017). DOI 10.1007/978-3-319-57454-7_49
14. Christen, P., Vidanage, A., Ranbaduge, T., Schnell, R.: Pattern-mining based cryptanalysis of Bloom filters for privacy-preserving record linkage. In: Pacific-Asia Conference on Knowledge Discovery and Data Mining, pp. 530–542. Springer (2018). DOI 10.1007/978-3-319-93040-4_42
15. Conrad, K.: Stirling's formula. Available in http://www.math. uconn. edu/kconrad/blu rbs/analysis/stirling. pdf (2016). DOI 10.1002/0471667196.ess2579.pub2
16. Culnane, C., Rubinstein, B.I., Teague, V.: Options for encoding names for data linking at the australian bureau of statistics. arXiv preprint arXiv:1802.07975 (2018)
17. Damgård, I., Geisler, M., Krøigaard, M.: Efficient and secure comparison for on-line auctions. In: Australasian conference on information security and privacy, pp. 416–430. Springer (2007)
18. Dong, C., Chen, L., Wen, Z.: When private set intersection meets big data: an efficient and scalable protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 789–800 (2013). DOI 10.1145/2508859.2516701
19. Dong, X.L., Srivastava, D.: Big data integration. Synthesis Lectures on Data Management **7**(1), 1–198 (2015). DOI 10.2200/S00578ED1V01Y201404DTM040
20. Durham, E.A.: A framework for accurate, efficient private record linkage. Ph.D. thesis, Faculty of the Graduate School of Vanderbilt University, Nashville, TN (2012)
21. Dwork, C.: Differential privacy. Automata, Languages and Programming pp. 1–12 (2006). DOI 10.1007/11787006_1
22. Eisen, M.B., Spellman, P.T., Brown, P.O., Botstein, D.: Cluster analysis and display of genome-wide expression patterns. Proceedings of the National Academy of Sciences **95**(25), 14863–14868 (1998)
23. ElGamal, T.: A public key cryptosystem and a signature scheme based on discrete logarithms. IEEE transactions on information theory **31**(4), 469–472 (1985). DOI 10.1109/TIT.1985.1057074
24. Essex, A.: Secure approximate string matching for privacy-preserving record linkage. IEEE Transactions on Information Forensics and Security **14**(10), 2623–2632 (2019)
25. Ferragina, P., Manzini, G.: Opportunistic data structures with applications. In: Proceedings 41st Annual Symposium on Foundations of Computer Science, pp. 390–398. IEEE (2000)
26. Ferrer, J.D.: A new privacy homomorphism and applications. Information Processing Letters **60**(5), 277–282 (1996). DOI 10.1016/S0020-0190(96)00170-6
27. Franklin, M.K., Reiter, M.K.: Fair exchange with a semi-trusted third party. In: Proceedings of the 4th ACM Conference on Computer and Communications Security, pp. 1–5 (1997). DOI 10.1145/266420.266424
28. Goldreich, O.: Secure multi-party computation. Tech. rep., Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Israel (2002)
29. Graham, R.L., Knuth, D.E., Patashnik, O., Liu, S.: Concrete mathematics: a foundation for computer science. Computers in Physics **3**(5), 106–107 (1989)

30. Hahn, F., Loza, N., Kerschbaum, F.: Practical and secure substring search. In: Proceedings of the 2018 International Conference on Management of Data, pp. 163–176 (2018). DOI 10.1145/3183713.3183754
31. Hall, R., Fienberg, S.E.: Privacy-preserving record linkage. In: International conference on privacy in statistical databases, pp. 269–283. Springer (2010). DOI 10.1007/978-3-642-15838-4_24
32. Juels, A., Sudan, M.: A fuzzy vault scheme. Designs, Codes and Cryptography **38**(2), 237–257 (2006)
33. Karakasidis, A., Verykios, V.S., Christen, P.: Fake injection strategies for private phonetic matching. In: Data Privacy Management and Autonomous Spontaneus Security, pp. 9–24. Springer (2011). DOI 10.1007/978-3-642-28879-1_2
34. Karapiperis, D., Gkoulalas-Divanis, A., Verykios, V.S.: Federal: A framework for distance-aware privacy-preserving record linkage. IEEE Transactions on Knowledge and Data Engineering **30**(2), 292–304 (2017). DOI 10.1109/TKDE.2017.2761759
35. Karapiperis, D., Verykios, V.S.: A fast and efficient hamming lsh-based scheme for accurate linkage. Knowledge and Information Systems **49**(3), 861–884 (2016). DOI 10.1007/s10115-016-0919-y
36. Kerschbaum, F.: Frequency-hiding order-preserving encryption. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 656–667 (2015). DOI 10.1145/2810103.2813629
37. Lindell, Y., Pinkas, B.: Secure multiparty computation for privacy-preserving data mining. Journal of Privacy and Confidentiality **1**(1) (2009)
38. McCreight, E.M.: A space-economical suffix tree construction algorithm. Journal of the ACM (JACM) **23**(2), 262–272 (1976). DOI 10.1145/321941.321946
39. Mitzenmacher, M., Upfal, E.: Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis. CUP (2005)
40. Mullaymeri, X., Karakasidis, A.: A two-party private string matching fuzzy vault scheme. In: ACM Symposium on Applied Computing, pp. 340–343 (2021)
41. Nakagawa, Y., Ohata, S., Shimizu, K.: Efficient privacy-preserving variable-length substring match for genome sequence. In: Workshop on Algorithms in Bioinformatics. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2021)
42. Randall, S., Wichmann, H., Brown, A., Boyd, J., Eitelhuber, T., Merchant, A., Ferrante, A.: A blinded evaluation of privacy preserving record linkage with Bloom filters. BMC Medical Research Methodology **22**(1), 1–7 (2022)
43. Randall, S.M., Ferrante, A.M., Boyd, J.H., Bauer, J.K., Semmens, J.B.: Privacy-preserving record linkage on large real world datasets. Journal of biomedical informatics **50**, 205–212 (2014). DOI 10.1016/j.jbi.2013.12.003
44. Schneier, B., et al.: Applied cryptography-protocols, algorithms, and source code in c (1996)
45. Schnell, R., Bachteler, T., Reiher, J.: Privacy-preserving record linkage using Bloom filters. BMC medical informatics and decision making **9**(1), 1–11 (2009). DOI 10.1186/1472-6947-9-41
46. Schnell, R., Borgs, C.: Encoding hierarchical classification codes for privacy-preserving record linkage using Bloom filters. In: Joint European Conference on Machine Learning and Knowledge Discovery in Databases, pp. 142–156. Springer (2019). DOI 10.1007/978-3-030-43887-6_12
47. Schnell, R., Borgs, C.: Encoding diagnostic codes for privacy-preserving record linkage. International Journal of Population Data Science **5**(5) (2020). DOI 10.23889/ijpds.v5i5.1461
48. Schnell, R., Klingwort, J., Farrow, J.M.: Locational privacy-preserving distance computations with intersecting sets of randomly labeled grid points. International Journal of Health Geographics **20**(1), 1–16 (2021). DOI 10.1186/s12942-021-00268-y
49. Shannon, C.: A mathematical theory of communication, bell systems technol. J **27**(3), 379–423 (1948). DOI 10.1002/j.1538-7305.1948.tb01338.x
50. Sheikh, R., Mishra, D.K.: Protocols for getting maximum value for multi-party computations. In: 2010 Fourth Asia International Conference on Mathematical/Analytical Modelling and Computer Simulation, pp. 597–600. IEEE (2010). DOI 10.1109/AMS.2010.120
51. Shimizu, K., Nuida, K., Rätsch, G.: Efficient privacy-preserving string search and an application in genomics. Bioinformatics **32**(11), 1652–1661 (2016). DOI 10.1093/bioinformatics/btw050
52. Smith, D.: Secure pseudonymisation for privacy-preserving probabilistic record linkage. Journal of Information Security and Applications **34**, 271–279 (2017). DOI 10.1016/j.jisa.2017.01.002
53. Sudo, H., Jimbo, M., Nuida, K., Shimizu, K.: Secure wavelet matrix: Alphabet-friendly privacy-preserving string search for bioinformatics. IEEE/ACM transactions on computational biology and bioinformatics **16**(5), 1675–1684 (2018)
54. Sun, S., Qian, Y., Zhang, R., Wang, Y., Li, X.: An improved chinese string comparator for Bloom filter based privacy-preserving record linkage. Entropy **23**(8), 1091 (2021)
55. Ukkonen, E.: Approximate string-matching over suffix trees. In: Annual Symposium on Combinatorial Pattern Matching, pp. 228–242. Springer (1993). DOI 10.1007/BFb0029808
56. Vatsalan, D., Christen, P.: Scalable privacy-preserving record linkage for multiple databases. In: ACM Conference on Information and Knowledge Management. Shanghai (2014)
57. Vatsalan, D., Christen, P.: Privacy-preserving matching of similar patients. Journal of biomedical informatics **59**, 285–298 (2016). DOI 10.1016/j.jbi.2015.12.004
58. Vatsalan, D., Christen, P., Verykios, V.S.: A taxonomy of privacy-preserving record linkage techniques. Information Systems **38**(6), 946–969 (2013). DOI 10.1016/j.is.2012.11.005
59. Vatsalan, D., Sehili, Z., Christen, P., Rahm, E.: Privacy-preserving record linkage for big data: Current approaches and research challenges. In: Handbook of Big Data Technologies, pp. 851–895. Springer (2017). DOI 10.1007/978-3-319-49340-4_25
60. Wandelt, S., Deng, D., Gerdjikov, S., Mishra, S., Mitankin, P., Patil, M., Siragusa, E., Tiskin, A., Wang, W., Wang, J., et al.: State-of-the-art in string similarity search and join. ACM SIGMOD Record **43**(1), 64–76 (2014). DOI 10.1145/2627692.2627706
61. Wang, J., Yang, X., Wang, B., Liu, C.: An adaptive approach of approximate substring matching. In: International Conference on Database Systems for Advanced Applications, pp. 501–516. Springer (2016). DOI 10.1007/978-3-319-32025-0_31
62. Zarezadeh, M., Mala, H., Ladani, B.T.: Efficient secure pattern matching with malicious adversaries. IEEE Transactions on Dependable and Secure Computing (2020). DOI 10.1109/TDSC.2020.3009595
63. Zipf, G.: Human Behavior and the Principle of Least Effort. Addison-Wesley Press (1949)