Pattern-Mining based Cryptanalysis of Bloom Filters for Privacy-Preserving Record Linkage*

Peter Christen[1], Anushka Vidanage[1], Thilina Ranbaduge[1], and Rainer Schnell[2]

[1] Research School of Computer Science, The Australian National University, Canberra, Australia. Contact: peter.christen@anu.edu.au
[2] Methodology Research Group, University Duisburg-Essen, Duisburg, Germany.

Abstract. Data mining projects increasingly require records about individuals to be linked across databases to facilitate advanced analytics. The process of linking records without revealing any sensitive or confidential information about the entities represented by these records is known as privacy-preserving record linkage (PPRL). Bloom filters are a popular PPRL technique to encode sensitive information while still enabling approximate linking of records. However, Bloom filter encoding can be vulnerable to attacks that can re-identify some encoded values from sets of Bloom filters. Existing attacks exploit that certain Bloom filters can occur frequently in an encoded database, and thus likely correspond to frequent plain-text values such as common names. We present a novel attack method based on a maximal frequent itemset mining technique which identifies frequently co-occurring bit positions in a set of Bloom filters. Our attack can re-identify encoded sensitive values even when all Bloom filters in an encoded database are unique. As our experiments on a real-world data set show, our attack can successfully re-identify values from encoded Bloom filters even in scenarios where previous attacks fail.

Keywords: Privacy; re-identification; Apriori algorithm; FPmax; data linkage.

1 Introduction

Applications in domains ranging from healthcare, business analytics and social science research all the way to fraud detection and national security increasingly require records about individual entities to be linked across several databases, and commonly across different organizations. Linked individual-level databases allow to improve data quality and enrich data, and open novel ways of data analysis and mining not possible on a single database [4].

Due to the lack of common unique entity identifiers (such as social security numbers or patient identifiers) across databases, linking records of individuals most often requires identifying personal details such as names, addresses and

* This work was funded by the Australian Research Council under DP130101801 and DP160101934. Peter Christen likes to acknowledge the support of ScaDS Dresden/Leipzig (BMBF grant 01IS14014B), where parts of this work were conducted.

dates of birth [4]. However, in many application areas growing concerns about privacy and confidentiality increasingly limit the use of sensitive personal details for linking databases across organizations [20].

Since the 1990s, the research area of *privacy-preserving record linkage* (PPRL) has aimed to develop techniques for linking records that correspond to the same entities across databases while protecting the privacy and confidentiality of these entities [20]. The general idea behind PPRL is to encode sensitive identifying attribute values (such as names, addresses, and dates of birth) and to conduct the linkage using these encoded values. At the end of the PPRL process the organizations involved only learn which of their records are matched with records from the other databases according to some decision model, but they cannot learn any sensitive information about the other databases, while any external attacker is not able to learn anything about these databases at all [20].

As surveyed by Vatsalan et al. [20], various encoding techniques have been proposed for PPRL. They can be categorized into secure multi-party computation (SMC) and perturbation based techniques, as well as hybrid techniques that combine aspects of both. While SMC techniques are accurate and provably secure, they often incur high computation and communication costs. Perturbation based techniques, on the other hand, are generally efficient and provide a trade-off between linkage quality, scalability, and privacy.

One perturbation technique that has attracted much interest is Bloom filter (BF) encoding [2,17]. As we describe in detail in the following section, a BF is a bit array (initialized to 0) into which elements of a set are mapped into by setting those positions to 1 that are selected by a set of hash functions. For PPRL, the elements of these sets are commonly character q-grams as extracted from the string values used for linking. The number of common 1-bits between two BFs approximates the number of common encoded q-grams and allows the calculation of the similarity between two BFs, as is illustrated in Fig. 1.

BF encoding has shown to allow accurate and efficient PPRL of large databases, and first practical PPRL systems based on BF encoding are now being deployed [3,16]. However, recent work has shown that BFs can be successfully attacked with the aim to re-identify the sensitive values encoded in them [5,10,11,12,13,15]. These cryptanalysis attacks assume that some bit patterns occur many times in an encoded BF database, allowing a mapping of frequent bit patterns to frequent plain-text values (such as common names) to identify the BF bit positions that possibly can correspond to certain q-grams in a plain-text value. Most existing attacks also require knowledge of certain parameters used in the BF encoding process, or they have high computational costs making them impractical for attacking large databases.

Contributions: We present a novel attack method that employs a maximal frequent itemset based pattern-mining approach (combining Apriori [1] and FP-Max [7]) to identify frequently co-occurring bit positions in a BF database that encode frequent q-grams in a plain-text database. Our attack can re-identify q-grams and plain-text values even when each BF in an encoded database is unique, and when an attacker has no knowledge of any parameters used in the

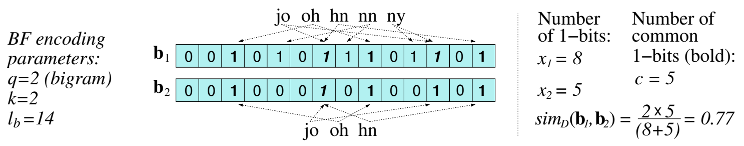
BF encoding parameters:
q=2 (bigram)
k=2
l_b=14
jo oh hn nn ny
$\mathbf{b}_1$ 0 0 1 0 1 0 1 1 1 0 1 1 0 1
$\mathbf{b}_2$ 0 0 1 0 0 0 1 0 1 0 0 1 0 1
jo oh hn
Number of 1−bits:
$x_1 = 8$
$x_2 = 5$
Number of common 1−bits (bold):
$c = 5$
$sim_D(\mathbf{b}_1, \mathbf{b}_2) = \frac{2 \times 5}{(8+5)} = 0.77$

Fig. 1. An example Dice coefficient similarity calculation of the first names 'johnny' and 'john' encoded in BFs, as described in Sect. 2. Hash collisions are shown in italics.

BF encoding process. Our evaluation on real-world data sets shows that our attack can re-identify plain-text values even when several attributes have been encoded into a BF, a situation where previous attacks would not be successful.

2 Background and Related Work

We now describe Bloom filter (BF) encoding for PPRL in more detail, and then provide an overview of existing cryptanalysis attacks on encoded BF.

Bloom Filter Encoding for PPRL: BF encoding is a popular privacy technique for PPRL due to its ability to calculate similarities between BF encoded string [6,17,20] and numerical [9,19] values, as well as its efficiency and simplicity of implementation. As the recent study by Randall et al. [16] has shown, in real-world applications PPRL based on encoded BFs can achieve linkage quality similar to traditional linkage methods on unencoded values.

BFs were proposed by Bloom [2] to efficiently represent sets and test for set membership. A BF $\mathbf{b}$ is a bit array of length l_b initialized to 0. k independent hash functions, $h_1, \ldots, h_k$, are used to map the elements s in a set $\mathbf{s}$ into $\mathbf{b}$ by setting the bit positions $\mathbf{b}[h_j(s)] = 1$, with $1 \leq j \leq k$, $1 \leq h_j(s) \leq l_b$, and $\forall s \in \mathbf{s}$.

In PPRL, where most attributes used for linking contain strings, the set $\mathbf{s}$ can be generated from character q-grams (sub-strings of length q [4]) extracted from a string value using a sliding window approach [17]. As illustrated in Fig. 1, the Dice coefficient (which is insensitive to many matching zeros) [4] can then be used to calculate the similarity between two BFs $\mathbf{b}_1$ and $\mathbf{b}_2$, as: $sim_D(\mathbf{b}_1, \mathbf{b}_2) = 2c/(x_1 + x_2)$, where c is the number of common bit positions that are set to 1 in both BFs, and x_1 and x_2 are the number of bit positions set to 1 in $\mathbf{b}_1$ and $\mathbf{b}_2$, respectively. Note that the hashing process can lead to collisions (where several q-grams are hashed to the same position), leading to false positives [14,17].

Different methods of how to encode records into BFs have been proposed. Generating one BF per attribute, known as *attribute level* BF (ABF), is one approach which has the advantage of allowing one similarity to be calculated per attribute. However, ABFs are susceptible to attacks due to the frequency distribution of common attribute values [5]. Alternatively, two methods that hash several attribute values of a record into one combined BF are the *cryptographic long term key* (CLK) [18] and *record level* BFs (RBF) [6]. With CLK, q-grams from several attributes are hashed into one BF, while in RBF the values from different attributes are first hashed into individual ABFs, and then bits are selected from each ABF into one RBF according to weights assigned to attributes.

A double hashing scheme was initially proposed for BF encoding for PPRL [17], where the k hash functions are based on the sum of the integer representation of two independent hash functions. This approach is however vulnerable to attacks [10,15], as discussed below. An alternative to prevent against these attacks is random hashing, where an integer representation of an element s to be hashed is used as the seed of a random generator used for hashing [18].

Once the databases to be linked are encoded into BFs by their owners, they are either sent to a linkage unit to calculate the similarity between pairs of BFs and classify them as matches or non-matches, or BFs are exchanged among the database owners to distributively calculate the similarities between BFs [17,20].

Cryptanalysis Attacks on Bloom Filter based PPRL: Because BF encoding is now being used in practical PPRL applications [3,16], it is highly important to identify the limits of this technique with regard to the privacy protection it provides. As we now describe, BFs are prone to different attacks.

The first attack method proposed by Kuzu et al. [6,11] was based on a constraint satisfaction problem (CSP) solver which assigns values to variables such that a set of constraints is satisfied. The attack is achieved by a frequency alignment of a set of BF encodings of records and sensitive plain-text values, where it requires access to a global database where the encoded records and their frequencies are drawn from. The attack was applied on a real patient database where it was successful in re-identifying four out of 20 frequent names correctly [12].

More recently, Niedermeyer et al. [15] proposed an attack on BFs based on the counts of q-grams extracted from frequent German surnames. From $7,580$ unique surnames encoded into $10,000$ BFs, the authors manually re-identified the 934 most frequent ones. This work was extended by Kroll and Steinmetzer [10] into a cryptanalysis of several attributes, which was able to re-identify 44% of plain-text values correctly. Both these attack methods are however only applicable with the double hashing approach used by Schnell et al. [17].

Christen et al. [5] recently proposed a novel efficient attack method that does not require any knowledge of the BF encoding function and its parameters used. The method aligns frequent BFs with frequent plain-text values and identifies sets of q-grams that could have been hashed to certain bit positions or not. The attack was successfully applied on large real databases and was able to correctly re-identify the most frequent plain-text values within a few minutes.

The most recent attack by Mitchell et al. [13] depends on the strong assumption that the adversary knows all parameters used in the BF encoding process. First, a brute-force attack is used to identify all possible q-grams encoded in a BF, then a graph is built which represents possible plain-text values that can be generated from the identified q-grams in a BF. The evaluation on real-world databases showed a 76.8% accuracy of correct one-to-one re-identifications.

The drawbacks of existing attacks on encoded BFs for PPRL are that they require knowledge about certain parameters used during BF encoding, are computationally expensive, and/or are only applicable if there are BFs and plain-text values that occur frequently such that their frequency alignments can be exploited. Our novel attack, presented next, overcomes these limitations.

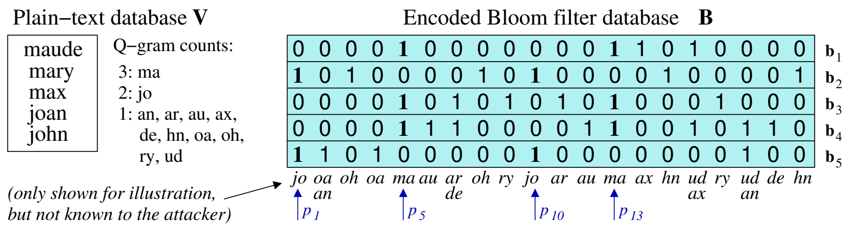
Plain−text database **V**
maude
mary
max
joan
john
Q−gram counts:
3: ma
2: jo
1: an, ar, au, ax, de, hn, oa, oh, ry, ud
Encoded Bloom filter database **B**
0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 0 0 0 0 $\mathbf{b}_1$
1 0 1 0 0 0 0 1 0 1 0 0 0 0 1 0 0 0 0 1 $\mathbf{b}_2$
0 0 0 0 1 0 1 0 1 0 1 0 1 0 0 0 1 0 0 0 $\mathbf{b}_3$
0 0 0 0 1 1 1 0 0 0 0 1 1 0 0 1 0 1 1 0 $\mathbf{b}_4$
1 1 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 $\mathbf{b}_5$
jo oa oh oa ma au ar oh ry jo ar au ma ax hn ud ry ud de hn
an de ax an
(only shown for illustration, but not known to the attacker)
p_1
p_5
p_{10}
p_{13}

Fig. 2. Example of our proposed attack method, as described in Sect. 3. Using a frequent pattern-mining approach, we first identify that bit positions p_5 and p_{13} have co-occurring 1-bits in the *same three BFs* ($\mathbf{b}_1$, $\mathbf{b}_3$ and $\mathbf{b}_4$) and therefore must encode 'ma' which is the only q-gram that occurs in *three plain-text values*. Next, we find that positions p_1 and p_{10} must encode 'jo' as they have co-occurring 1-bits in the *same two BFs* ($\mathbf{b}_2$ and $\mathbf{b}_5$) and 'jo' is the only q-gram that occurs in *two plain-text values*. Based on the identified q-grams and their bit positions, we learn that BFs $\mathbf{b}_2$ and $\mathbf{b}_4$ can only encode 'john' and 'joan', while $\mathbf{b}_1$, $\mathbf{b}_3$ and $\mathbf{b}_4$ can encode 'maude', 'mary' or 'max'.

3 Pattern-Mining based Cryptanalysis Attack

We now discuss the ideas behind our attack method, as we illustrate in Fig. 2. In Sects. 3.1 and 3.2 we then describe the two main phases of our method in detail, and analyze their complexities. For notation we use bold letters for BFs, sets and lists (with upper-case bold letters for sets/lists of BFs/sets/lists) and normal type letters for integers and strings. Lists are shown with square and sets with curly brackets, where lists have an order while sets do not.

Previous attack methods exploit the bit patterns in and between BFs that occur frequently in an encoded BF database. Our new method is the first to exploit the co-occurrence between BF bit positions without requiring frequent BFs. As with existing attacks on BFs for PPRL, we assume an attacker has access to an encoded BF database, $\mathbf{B}$, where it is unknown which plain-text value(s) are encoded in a BF; and a plain-text database, $\mathbf{V}$, that contains values from one or several attributes. An attacker can guess which attributes are encoded in $\mathbf{B}$ based on the distribution of the number of 1-bits in BFs in $\mathbf{B}$ [18]. However, unlike with other attacks [10,11,12,13,15], the attacker does not require any information used in the BF encoding, such as the number and type of hash functions.

In the first phase of our attack (Sect. 3.1), we identify sets of frequently co-occurring bit positions (columns) in $\mathbf{B}$ that can encode q-grams that are frequent in $\mathbf{V}$. In the second phase (Sect. 3.2) we then re-identify possible plain-text values $v \in \mathbf{V}$ that can be encoded in a BF $\mathbf{b} \in \mathbf{B}$ based on these identified frequent q-grams. Our attack exploits the way BFs are constructed as follows:

Proposition 1 *Assuming a q-gram q occurs in $n_q < n$ records in a plain-text database $\mathbf{V}$ that contains $n = |\mathbf{V}|$ records, and $k \geq 1$ independent hash functions are used to encode q-grams from $\mathbf{V}$ into the encoded database $\mathbf{B}$ of n BFs, i.e. $|\mathbf{V}| = |\mathbf{B}|$. Then, (i) each BF bit position that can encode q must contain a 1-bit in at least n_q BFs in $\mathbf{B}$, and (ii) if $k > 1$ then up to k bit positions must contain a 1-bit in the same subset of BFs $\mathbf{B}_q \subseteq \mathbf{B}$, with $n_q = |\mathbf{B}_q|$, that encode q.*

Proof. For (i), based on the BF construction principle [2,17], if a BF contains a 0-bit in a bit position p where any of the k hash functions have mapped a q-gram q into, then this BF cannot encode q. Formally, $\exists\ h_j, 1 \leq j \leq k : (h_j(q) = p) \wedge (\mathbf{b}[p] = 0) \Rightarrow \mathbf{b}$ cannot encode q. For (ii), given two distinct BF bit positions, p_x and p_y (with $1 \leq p_x, p_y \leq l_b$ and $p_x \neq p_y$), for a given BF $\mathbf{b}$, if $\mathbf{b}[p_x] \neq \mathbf{b}[p_y]$ then $\mathbf{b}$ cannot encode a q-gram q because $\forall h_i, h_j, 1 \leq i \leq k, 1 \leq j \leq k, i \neq j$: $(h_i(q) = p_x \wedge h_j(q) = p_y) \Rightarrow (\mathbf{b}[p_x] = 1) \wedge (\mathbf{b}[p_y] = 1)$.

Because of possible collisions of the k hash functions used to map q-grams into BFs [14], potentially less than k bit positions will encode a certain q-gram, and thus less than k bit positions might be co-occurring frequently. We can calculate the probability of this to happen using the birthday paradox [14], which, for example for the commonly used BF settings $l_b = 1,000$ and $k = 30$ [18], leads to a probability of 0.64 of no collision for a single q-gram, and a probability of less than 0.07 of more than one collision. For $k = 20$ the probability of no collision is 0.83 while for $k = 10$ it is 0.96. Our attack should therefore be able to clearly identify frequently co-occurring bit positions that encode the same q-gram with high accuracy, as we validate experimentally on a real data set in Sect. 4.

3.1 Identifying Co-occurring Bit Positions in Bloom Filters

The first phase of our attack, as detailed in Algo. 1, identifies the sets of co-occurring bit positions (columns) in the BF database $\mathbf{B}$ that correspond to the frequent q-grams in the plain-text database $\mathbf{V}$.

The algorithm first converts $\mathbf{B}$ from its row storage (one BF per record) into a column-wise format $\mathbf{B}_c$ of l_b bit arrays each of length $|\mathbf{B}|$ to allow efficient access to individual bit positions. Then, in line 3, the plain-text values $v \in \mathbf{V}$ are converted into one set of q-grams per record, where $\mathbf{Q}$ is the list of q-gram sets of all records in $\mathbf{V}$. Various lists and sets are then initialized, as is the queue Q in line 8 with the first partition that contains all BFs and all bit positions. A partition is a subset of BFs (rows) and bit positions (columns) in $\mathbf{B}$.

The main loop of the algorithm starts in line 9, where the largest partition in the queue Q will be processed. In each iteration, we use a tuple containing $\mathbf{c}_i$ (the column filter of which bit positions in $\mathbf{B}_c$ to consider), $\mathbf{r}_i$ (the row filter of which BFs in $\mathbf{B}_c$ to consider), and $\mathbf{m}_i$ and $\mathbf{n}_i$ (the set of must and cannot contain q-grams, respectively, that a q-gram set in $\mathbf{Q}$ must / cannot contain in order to be considered in this iteration). The function **GetTwoMostFreqQGrams()** called in line 11 returns the two q-grams, q_1 and q_2, and their respective frequencies, f_1 and f_2 (with $f_1 \geq f_2$), that occur in most q-gram sets in $\mathbf{Q}$, conditional on only considering q-gram sets in $\mathbf{Q}$ that contain all q-grams in $\mathbf{m}_i$ and no q-gram from $\mathbf{n}_i$.

The percentage difference between f_1 and f_2 is then calculated as d_i in line 12, and only if d_i is at least the user provided minimum threshold d (i.e. q-gram q_1 occurs a certain times more often than q_2) will the current partition i be processed. This ensures q_1 is clearly more frequent than q_2 in the partition for it to be assigned to the set of co-occurring bit positions $\mathbf{f}_i$ as described next.

```
Algorithm 1: Identify frequent q-grams co-occurring with frequent bit positions – Phase 1
Input:
- $\mathbf{B}$: List of Bloom filters (BFs) from the sensitive encoded database, one BF per record
- $\mathbf{V}$: List of plain-text values from a public database, one string value per record
- $l_q$: Length of sub-strings to extract from plain-text values
- $d$: Minimum percentage difference between two most frequent q-grams in a partition
- $m$: Minimum partition size (a subset of BFs and bit positions in $\mathbf{B}$) as a number of BFs
Output:
- $\mathbf{F}$:        List of frequent q-grams and their identified sets of co-occurring BF bit positions
- $\mathbf{A}^+$, $\mathbf{A}^-$: Lists of must have and cannot have q-gram sets assigned to each BF in $\mathbf{B}$
1:  $\mathbf{B}_c = ConvColWise(\mathbf{B})$        // One bit array per BF bit position (column) for efficient access
2:  $l_b = |\mathbf{B}_c|$                           // Number of BF bit positions (BF length)
3:  $\mathbf{Q} = GenQGramSets(\mathbf{V}, l_q)$ // Convert plain-text values into q-gram sets, one set per record
4:  $\mathbf{F} = [\,]$, $\mathbf{A}^+ = [\,]$, $\mathbf{A}^- = [\,]$     // Initialize lists to be generated
5:  $\mathbf{r} = \{b : 1 \leq b \leq |\mathbf{B}|\}$            // Initialize row partition filter (so all BFs are considered)
6:  $\mathbf{c} = \{p : 1 \leq p \leq l_b\}$              // Initialize column filter (so all BF bit positions are considered)
7:  $\mathbf{m} = \{\}$, $\mathbf{n} = \{\}$                   // Initialize empty sets of must and cannot include q-grams
8:  $Q = [(|\mathbf{B}|, \mathbf{c}, \mathbf{r}, \mathbf{m}, \mathbf{n})]$            // Initialize queue with first partition (the full BF data set)
9:  while $|Q| > 0$ do:                          // Main loop: As long as the queue is not empty
10:     $(ps_i, \mathbf{c}_i, \mathbf{r}_i, \mathbf{m}_i, \mathbf{n}_i) = Q.pop()$   // Start iteration $i$: Get first tuple from queue
11:     $q_1, f_1, q_2, f_2 = \mathbf{GetTwoMostFreqQGrams}(\mathbf{Q}, \mathbf{m}_i, \mathbf{n}_i)$      // Most frequent in partition
12:     $d_i = 2(f_1 - f_2)/(f_1 + f_2) \cdot 100$ // Percentage difference between two most frequent q-grams
13:     if $d_i \geq d$ then:                        // Only continue if large enough percentage difference
14:         $s_i = |\mathbf{B}| \cdot (f_1 + f_2)/(2|\mathbf{V}|)$   // Minimum support for frequent pattern-mining
15:         $\mathbf{f}_i = \mathbf{GetLongFreqCoOccurBitPos}(\mathbf{B}_c, s_i, \mathbf{c}_i, \mathbf{r}_i)$         // Run pattern-mining
16:         if $\mathbf{f}_i \neq \emptyset$ then:   // A set of frequent co-occurring bit positions was identified
17:             $\mathbf{F}[q_1] = \mathbf{f}_i$        // Assign bit positions to most frequent q-gram $q_1$ (from line 11)
18:             $\mathbf{c}_i = \mathbf{c}_i \setminus \mathbf{f}_i$       // Remove identified bit positions so they are not considered anymore
19:             $\mathbf{r}_i^+, \mathbf{r}_i^- = \mathbf{UpdateRowFilter}(\mathbf{f}_i, \mathbf{r}_i)$    // Update row filter based on bit positions in $\mathbf{f}_i$
20:             $\forall b \in \mathbf{r}^+ : \mathbf{A}^+[b] = \mathbf{A}^+[b] \cup \{q_1\}$         // Assign $q_1$ to must have q-gram sets of BFs
21:             $\forall b \in \mathbf{r}^- : \mathbf{A}^-[b] = \mathbf{A}^-[b] \cup \{q_1\}$         // Assign $q_1$ to cannot have q-gram sets of BFs
22:             if $|\mathbf{r}_i^+| \geq m$ then:                       // Partition containing $q_1$ is large enough
23:                 $Q.add((|\mathbf{r}_i^+|, \mathbf{c}_i, \mathbf{r}_i^+, \mathbf{m}_i \cup \{q_1\}, \mathbf{n}_i))$ // New partition of BFs containing $q_1$
24:             if $|\mathbf{r}_i^-| \geq m$ then:                       // Partition not containing $q_1$ is large enough
25:                 $Q.add((|\mathbf{r}_i^-|, \mathbf{c}_i, \mathbf{r}_i^-, \mathbf{m}_i, \mathbf{n}_i \cup \{q_1\}))$ // New partition of BFs not containing $q_1$
26:             $Q.sort()$                                        // Sort queue by partition size with largest first
27: return $\mathbf{F}$, $\mathbf{A}^+$, $\mathbf{A}^-$
```

In line 14 the average frequency of q_1 and q_2 is converted into a support count s_i of required 1-bits in $\mathbf{B}_c$ to be used for frequent pattern-mining. This is based on Proposition 1, because any set of co-occurring bit positions in $\mathbf{B}_c$ that potentially can encode q_1 must have 1-bits in at least s_i common BFs (rows) in $\mathbf{B}_c$. The function **GetLongFreqCoOccurBitPos()** (line 15) employs the Apriori [1] and FPmax [7] algorithms on the bit positions in $\mathbf{B}_c$, where bit positions correspond to items and sets of co-occurring bit positions to itemsets. The aim of frequent pattern-mining is to find the longest set (likely of size k) of co-occurring bit positions, $\mathbf{f}_i$, that have a 1-bit in at least s_i common BFs.

Because the length of the longest pattern is likely large (values of $15 \leq k \leq 30$ have been used [6,17,18]), employing only the Apriori algorithm would generate too many candidate bit position sets. On the other hand, running the FPmax algorithm would generate a very large FPtree with a number of branches in the order of $O(|\mathbf{B}|)$. To reduce the size of the tree generated by FPmax, we therefore first run Apriori to find those bit positions that occur in frequent triplets (i.e. frequent 3-itemsets) of co-occurring bit positions that have a 1-bit in at least s_i common BFs. The FPmax algorithm is then run on only those bit positions, and only the longest set of found frequent bit positions is returned in $\mathbf{f}_i$.

The parameters $\mathbf{c}_i$ and $\mathbf{r}_i$ are the column and row filters that select a subset of BF bit positions (columns) and rows (BFs) from $\mathbf{B}_c$ to be considered in this partition i (as described below). If a non-empty set of co-occurring bit positions $\mathbf{f}_i$ is returned, it is assigned in line 17 to the identified most frequent q-gram q_1 of this partition, as we assume that the bit positions in $\mathbf{f}_i$ must encode q_1.

In line 18, we remove the identified bit positions $\mathbf{f}_i$ from the set $\mathbf{c}_i$ of positions to be considered in the following iterations. In the function **UpdateRowFilter()**, based on the current set $\mathbf{r}_i$ of BFs that have been considered in this iteration, we generate the two new subsets $\mathbf{r}_i^+$ of BFs that do contain q_1 (have 1-bits in all positions in $\mathbf{f}_i$) and $\mathbf{r}_i^-$ of BFs that cannot contain q_1, where $\mathbf{r}_i^- = \mathbf{r}_i \setminus \mathbf{r}_i^+$. We then add the frequent q-gram q_1 to the sets of must have (line 20) and cannot have (line 21) q-gram sets, $\mathbf{A}^+$ and $\mathbf{A}^-$, for all BFs in $\mathbf{r}_i^+$ and $\mathbf{r}_i^-$.

Finally, we generate two new partitions (if their size is at least the minimum partition size, m), where in line 23 the new partition contains those BFs that contain the frequent q-grams q_1 (so q_1 is added to the must contain q-gram set $\mathbf{m}_i$) and in line 25 the new partition contains those BFs that do not contain q_1 (and therefore q_1 is added to the set $\mathbf{n}_i$ of not contained q-grams). In line 26 we sort the queue q such that the largest partition is first.

To estimate the complexity of Algo. 1, we assume $n = |\mathbf{B}| = |\mathbf{V}|$ is the number of BFs and plain-text values, respectively. The initialization steps and function calls in lines 1, 3, 11 and 19 are of complexity $O(n)$ as they require linear scans over $\mathbf{B}$ or $\mathbf{V}$. Assuming in each iteration of the main loop k bit positions are assigned to a frequent q-gram in line 17, then the expected number of iterations is $O(l_b/k)$. The complexities of the Apriori and FPMax algorithms in line 15 are known to be linear in the number of transactions (in our case BFs) and quadratic in the number of items (bit positions) [8] and therefore of $O(l_b^2 n)$. In the worst case the FPmax tree has a height of l_b and contains n branches. The overall complexity of Algo. 1 is therefore $O(l_b^3 n/k)$.

3.2 Plain-text Value Re-identification

As detailed in Algo. 2, the second phase of our attack aims to re-identify plain-text values from $\mathbf{V}$ that could have been encoded into BFs in $\mathbf{B}$ according to the lists of must have and cannot have q-gram sets for BFs, $\mathbf{A}^+$ and $\mathbf{A}^-$ (from Algo. 1). The algorithm only considers plain-text values and BFs that contain at least n_m must have q-grams assigned to them, because considering a single or only a few frequent q-grams would result in too many possible plain-text values that could match a BF (for example, nearly $4,000$ surnames from our experimental data set of $224,061$ records contain the q-gram ‘sm’).

The algorithm consists of three main steps, where in the first (lines 3 to 6) we find all BFs from $\mathbf{A}^+$ that (according to phase 1 of our attack) contain at least n_m identified q-grams. We build an inverted index, $\mathbf{I}_B$ where q-gram sets $\mathbf{q}_b$ are index keys, each with a list of BFs that contain $\mathbf{q}_b$ (line 6). In the second step (lines 7 to 10), the function **GetLongQGramSetInVal()** finds for each value $v \in \mathbf{V}$ the longest q-gram set $\mathbf{q}_v$ from $\mathbf{I}_B$. If a value v contains at least n_m identified q-grams then we add it to the inverted index list of $\mathbf{q}_v$ in $\mathbf{I}_V$.

Algorithm 2: ***Re-identify plain-text values in Bloom filters*** – Phase 2

```
Input:
- $\mathbf{V}$:        List of plain-text values from a public database, one string value per record
- $\mathbf{A}^+$, $\mathbf{A}^-$: Lists of must have and cannot have q-gram sets assigned to each BF in $\mathbf{B}$
- $n_m$:       Minimum number of identified q-grams in a BF from $\mathbf{B}$
Output:
- $\mathbf{R}$: List of re-identified plain-text values from $\mathbf{V}$ for BFs from $\mathbf{B}$
1:  $\mathbf{I}_B = [\,]$, $\mathbf{I}_V = [\,]$            // Initialize inverted indexes of BFs and plain-text values
2:  $\mathbf{R} = [\,]$                       // Initialize list of re-identified plain-text values to be generated
3:  for $b \in \mathbf{A}^+$ do:                // Step 1: Find all BFs in $\mathbf{A}^+$ with enough identified q-gram sets
4:    $\mathbf{q}_b = \mathbf{A}^+[b]$                 // Get the set of must have q-grams for this BF
5:    if $|\mathbf{q}_b| \geq n_m$ then:           // There are enough must have q-grams for this BF
6:      $\mathbf{I}_B[\mathbf{q}_b].add(b)$            // Add the BF identifier to the inverted index list of this q-gram set
7:  for $v \in \mathbf{V}$ then:                 // Step 2: Find longest identified q-gram set for plain-text values
8:    $\mathbf{q}_v$ = GetLongQGramSetInVal($\mathbf{I}_B, v$)   // Get longest known q-gram set in value $v$
9:    if $|\mathbf{q}_v| \geq n_m$ then:           // There are enough must have q-grams for this value
10:     $\mathbf{I}_V[\mathbf{q}_v].add(v)$            // Add the value to the list of this q-gram set in the inverted index
11: for $\mathbf{q}_b \in \mathbf{I}_B$ do:                // Step 3: Assign identified values to BFs
12:   if $\mathbf{q}_b \in \mathbf{I}_V$ then:             // Q-gram set occurs both in BFs and plain-text values
13:     for $b \in \mathbf{I}_B[\mathbf{q}_b]$ do:         // All BFs that contain this q-gram tuple
14:       for $v \in \mathbf{I}_V[\mathbf{q}_b]$ do:             // All plain-text values that contain this q-gram tuple
15:         if $\forall q \in \mathbf{A}^-[b] : q \notin v$ then: // If the value does not contain any cannot have q-grams
16:           $\mathbf{R}[b] = \mathbf{R}[b] \cup \{v\}$            // Add to set of re-identified values for BF identifier $b$
17: return $\mathbf{R}$
```

In the final step (lines 11 to 16), we loop over all q-gram sets $\mathbf{q}_b$ that have both BFs from $\mathbf{A}^+$ and plain-text values from $\mathbf{V}$. For each $\mathbf{q}_b$ and each of its BF identifiers b from $\mathbf{I}_B$, we find all possible corresponding values v from $\mathbf{I}_V$ that do not contain any cannot have q-grams for this BF according to $\mathbf{A}^-$ (line 15). All possible values v are added to the set $\mathbf{R}[b]$ of re-identified values for b.

The computational complexity of the first two steps of Algo. 2 is $O(n)$ because they loop over $\mathbf{A}^+$ and $\mathbf{V}$, respectively. In the third step, we loop over the identified q-gram tuples, and for each over its associated BFs and plain-text values. The maximum number of unique q-gram tuples in the inverted indexes $\mathbf{I}_B$ and $\mathbf{I}_V$ will be n, which would be the case where every value and every BF would contain a different q-gram tuple. In this case, each list in $\mathbf{I}_B$ and $\mathbf{I}_V$ would contain one BF identifier. The minimum number of q-gram tuples would be 1, the case where all values and BFs contain the same q-gram tuple. In this case the length of the corresponding list in $\mathbf{I}_B$ and $\mathbf{I}_V$ is n BF identifiers. Based on this, the overall complexity of step 3 of Algo. 2 is $O(n^2)$.

4 Experiments and Results

We evaluated our attack method using real data from the North Carolina Voter Registration (NCVR) database (`http://dl.ncsbe.gov/data/`). We used one snapshot of NCVR from April 2014 as $\mathbf{B}$ and another snapshot from June 2014 as $\mathbf{V}$. We extracted pairs of records that correspond to the same voter but had name and/or address changes over time, resulting in two files of 222,251 and 224,061 records, respectively. Using the CLK approach [18] discussed in Sect. 2, we encoded combinations of between two and four of the attributes *first name*, *last name*, *street address* and *city* into BFs. For combinations of three and four attributes, all plain-text values in $\mathbf{V}$ and bit patterns in $\mathbf{B}$ were unique.

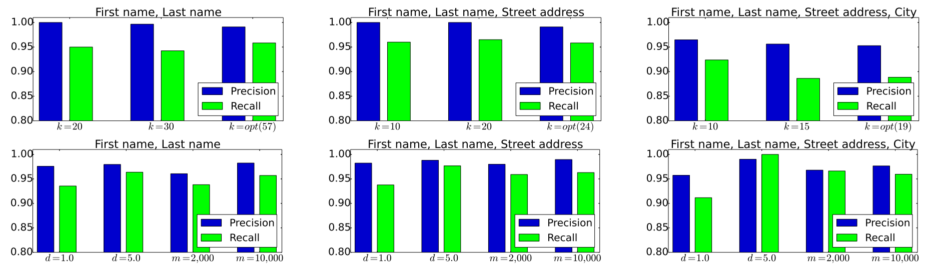

Fig. 3. Precision and recall results for the re-identified frequent q-grams from Algo. 1 (see Sect. 3.1), with different numbers of hash functions k (top), and different minimum partition sizes m and different q-gram frequency percentage differences d (bottom).

We used the following BF encoding settings: q=*2*, l_b=*1,000*, double and random hashing [18] as described in Sect. 2, and different values for k (top row in Fig. 3). We calculate the optimal number, *opt*, of hash functions such that the average number of 1-bits in a BF is 50% to minimize the false positive rate [19]. In Algo. 1 we set the values for the minimum percentage difference as d=*[1.0, 5.0]* and the minimum partition size as m=*[2,000, 10,000]* (bottom row in Fig. 3); and in Algo. 2 we set the minimum q-gram tuple size n_m=*3*, as these values provided good results in setup experiments.

We present the quality of the identified frequent q-grams from Algo. 1 as the precision and recall of how many bit positions were correctly identified for a q-gram in **F** averaged over all q-grams in **F**. For Algo. 2, we evaluated the quality of re-identified values in **R** as the percentages of (1) *exact* matches of a plain-text value with the true value encoded in a BF, (2) *partial* matches where not all words matched (for example, *first* and *last name* matched but *city* was different), and (3) *wrong* matches. We only considered BFs with 10 or less plain-text values assigned to them in **R**, and we separately present averaged results for *1-to-1* ($|\mathbf{R}[b]| = 1$) and *1-to-many* ($1 < |\mathbf{R}[b]| \leq 10$) re-identifications.

We compared our attack method with the recently proposed attack by Christen et al. [5] (the only other attack that does not require knowledge of the BF encoding parameters) which aligns frequent BFs and plain-text values to allow re-identification of the most frequent plain-text values. We implemented both attack methods using Python 2.7 and ran experiments on a server with 64-bit Intel Xeon 2.4 GHz CPUs, 128 GBytes of memory and running Ubuntu 14.04. The programs and data sets are available from: `https://dmm.anu.edu.au/pprlattack`.

Discussion: In Fig. 3 we show the results for the first phase of our attack (Algo. 1). As can be seen, both precision and recall of the identified frequent q-grams are very high, above 0.88, for all settings of parameters. This validates that our attack can successfully identify bit positions of q-grams with high accuracy even when no frequent BFs are available in an encoded database. Both precision and recall decrease slightly as more attributes are encoded into BFs, which is due to the increased number of unique encoded q-gram sets that make accurate frequent pattern-mining more difficult. A smaller difference d between the two

Table 1. Re-identification percentages of exact (E), partial (P) and wrong (W) matches of plain-text values from Algo. 2 (Sect. 3.2), averaged over the settings used in Fig. 3.

		Two attributes			Three attributes			Four attributes		
		E	P	W	E	P	W	E	P	W
Christen	1-to-1	5.0	0	95.0	0	0	100	0	0	100
et al. [5]	1-to-many	6.0	0	94.0	0	0	100	0	0	100
Our	1-to-1	20.7	30.9	48.4	0.2	61.0	38.8	0.5	73.2	26.3
approach	1-to-many	27.5	46.5	26.0	0.4	83.5	16.1	0.5	87.9	11.6

most frequent q-grams lowers re-identification accuracy, because the chance of a wrong frequent q-gram being identified is increased. The minimum partition size, m, seems to have no strong effect upon the q-gram re-identification accuracy.

As Table 1 shows, the re-identification results for the second phase (Algo. 2) of our attack has led to around 51% to 74% of 1-to-1 matches to be exact or partially correct, which means in the majority of cases where only one plain-text value $v \in \mathbf{V}$ was identified to match one BF $\mathbf{b} \in \mathbf{B}$, an attacker has information that likely allows re-identification of the individual represented by v. If between 2 and 10 values v match a BF, then the over 74% of re-identifications are exact or partially correct. The small values of exact matches for three and four attributes is because less than $1,800$ of over $222,200$ ($< 1\%$) combined values between the two NCVR snapshots are exact matches. In future work, we will concentrate on improving the accuracy and efficiency of this second phase of our approach.

As can also be seen from Table 1, the frequency based attack by Christen et al. [5] is only able to correctly re-identify a very small percentage of values when two attributes are encoded into BFs, because with three and four attributes no frequency information is available that could be exploited by this attack.

Our experiments show that basic BFs, even when each BF in an encoded database is unique, can successfully be attacked by identifying frequently co-occurring BF bit positions. These results highlight the need for improved BF encoding, as well as new PPRL encoding methods that do not exhibit the weaknesses of basic BFs that allow the re-identification of encoded values.

5 Conclusions and Future Work

We have presented a pattern-mining based attack method on BF encoding as used for PPRL. Our attack can successfully re-identify encoded q-grams and plain-text values even when all BFs in an encoded database are unique. Given that BF based PPRL is now employed in real-world applications [3,16], it is vital to study the limits of BF encoding. We believe our attack is important for PPRL because it allows data custodians to understand security flaws in BF encodings and ensure their encoded databases are not vulnerable to such attacks.

As future work we plan to improve the second phase of our attack by analyzing the differences in bit patterns between BFs to identify additional encoded q-grams which will allow an improved re-identification of plain-text values. We furthermore aim to formalize different attack scenarios for PPRL, and to conduct extensive experiments across a variety of parameter settings (such as different values of q, k, and l_b) and different encoding methods, and using data sets from a variety of domains to identify the limitations of our attack method. Finally, we will also explore how BF *hardening* techniques, such as balancing and XOR-folding [18], will influence the feasibility of our pattern-mining based attack.

References

1. Agrawal, R., Srikant, R.: Fast algorithms for mining association rules. In: VLDB. Santiago de Chile (1994)
2. Bloom, B.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
3. Boyd, J., Randall, S., Ferrante, A.: Application of privacy-preserving techniques in operational record linkage centres. In: Medical Data Privacy Handbook (2015)
4. Christen, P.: Data Matching – Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. Springer (2012)
5. Christen, P., Schnell, R., Vatsalan, D., Ranbaduge, T.: Efficient cryptanalysis of Bloom filters for privacy-preserving record linkage. In: PAKDD. Jeju, Korea (2017)
6. Durham, E.A., Kantarcioglu, M., Xue, Y., Toth, C., Kuzu, M., Malin, B.: Composite Bloom filters for secure record linkage. IEEE TKDE 26(12), 2956–2968 (2014)
7. Grahne, G., Zhu, J.: Fast algorithms for frequent itemset mining using FP-trees. IEEE TKDE 17(10), 1347–1362 (2005)
8. Hegland, M.: The Apriori algorithm – A tutorial. Mathematics and Computation in Imaging Science and Information Processing 11, 209–262 (2005)
9. Karapiperis, D., Gkoulalas-Divanis, A., Verykios, V.S.: FEDERAL: A framework for distance-aware privacy-preserving record linkage. IEEE TKDE 30(2) (2017)
10. Kroll, M., Steinmetzer, S.: Automated cryptanalysis of Bloom filter encryptions of databases with several personal identifiers. In: BIOSTEC. Lisbon (2015)
11. Kuzu, M., Kantarcioglu, M., Durham, E., Malin, B.: A constraint satisfaction cryptanalysis of Bloom filters in private record linkage. In: PET. Waterloo (2011)
12. Kuzu, M., Kantarcioglu, M., Durham, E., Toth, C., Malin, B.: A practical approach to achieve private medical record linkage in light of public resources. JAMIA (2013)
13. Mitchell, W., Dewri, R., Thurimella, R., Roschke, M.: A graph traversal attack on Bloom filter-based medical data aggregation. IJBDI 4(4), 217–226 (2017)
14. Mitzenmacher, M., Upfal, E.: Probability and Computing: Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, Cambridge (2005)
15. Niedermeyer, F., Steinmetzer, S., Kroll, M., Schnell, R.: Cryptanalysis of basic Bloom filters used for privacy preserving record linkage. JPC 6(2), 59–79 (2014)
16. Randall, S., Ferrante, A., Boyd, J., Bauer, J., Semmens, J.: Privacy-preserving record linkage on large real world datasets. JBI 50, 205–212 (2014)
17. Schnell, R., Bachteler, T., Reiher, J.: Privacy-preserving record linkage using Bloom filters. BMC Med Inform Decis Mak 9(1) (2009)
18. Schnell, R., Borgs, C.: Randomized response and balanced Bloom filters for privacy preserving record linkage. In: ICDMW DINA. Barcelona (2016)
19. Vatsalan, D., Christen, P.: Privacy-preserving matching of similar patients. JBI 59, 285–298 (2016)
20. 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)