Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

RESEARCH ARTICLE

Private Record Linkage with Linkage Maps

Shreya Patel | Rinku Dewri*

[1]Department of Computer Science, University of Denver, Colorado, USA

Correspondence
*Rinku Dewri Email: rdewri@cs.du.edu

Present Address
2155 E Wiley Av., Denver, CO 80210, USA

Summary

Private record linkage is an actively pursued research area to facilitate the linkage of database records under the constraints of regulations that do not allow linkage agents to learn sensitive identities of record owners. Recent works have shown that linkage using commutative ciphers, which were discarded earlier for efficiency concerns, can be made feasible by leveraging precomputations, data parallelism, and probabilistic key reuse approaches. In this work, we propose further optimizations that can be performed to improve the runtime efficiency of such an approach. We transition from modular exponentiation ciphers to elliptic curve operations to improve precomputation time, eliminate memory intensive comparisons of encrypted values, and introduce data structures to detect negative comparisons. We benchmark the proposed approach using real world demographics data, and provide an extensive study of the parametric aspects of the approach. We also supplement our execution time results with an assessment of the residual privacy risk left by the approach. The approach can perform a linkage of two datasets with 10^5 records each in 20 minutes in a commodity laptop. This is achieved by eliminating the need to compare more than 70% of the record pairs. By design, the linkage accuracy is also retained at the same level as a non-private record linkage procedure.

KEYWORDS:
private record linkage; ECDLP; linkage service; bigram matching

1 | INTRODUCTION

Data collection practices are now ubiquitous across many organizations. Pattern mining in the collected data is equally common in business practices, and enables novel discoveries and services. The value of such data is amplified when data sources can be merged to create richer content, collating multiple types of information gathered on an entity. Record linkage is the task of identifying records in different data sets that correspond to the same entity [1]. Potential usage of linked data range from understanding lifestyle choices, finding causes of diseases and their distribution patterns, fraud and crime detection, and exploring socio-behavioral phenomena [2,3,4].

Linked data widens an organization's view into an individual's life, either by bringing in raw information that was not available earlier, or by exposing patterns that were not identifiable before. Private record linkage aims to enable the linkage of distributed data sources using data attributes in formats that do not reveal their clear text values, yet enable the inference of matches and non-matches.

Addressing scalability and privacy in private record linkage has been a topic of extensive research for some time now. Approaches based on hash transformations of data can scale well[5,6], but lack privacy guarantees, or have been shown to be susceptible to attacks[7,8]. On the other hand, current approaches grounded on strong privacy properties, such as secure multi-party computation[9], are not yet scalable to handle data sets in the order of thousands or millions. Approaches based on cryptographic transformations of data have been discarded earlier due to computational and communication inefficiency. In 2003, Agrawal et al. proposed using a commutative encryption scheme based on modular exponentiations to dual encrypt data items (for example, all bigrams that make up a name)[10]. A set intersection on these encrypted data items can then be used to determine the similarity of data fields in two sources. It was estimated that around 200,000 exponentiations can be calculated in an hour on the hardware available at that time, and hence deemed impractical for large scale linkage. In 2016, Dewri et al. suggested modifications to this protocol and demonstrated its feasibility using optimizations driven by precomputations and data parallelism[11]. Since our method is strongly tied to this approach, we present the details of the approach (and associated background) before highlighting our contributions.

1.1 | System and threat model

A federated query processing architecture aggregates query results returned from distributed sources. The aggregation can involve deduplication of results, or a join of result records. Both of these tasks require the ability to determine if two records represent the same underlying entity. A (private) data linkage process is carried out to support this requirement. Our system model is focused on the entities and operations specific to this linkage process. The two entities in this model are *data holders* and a *linkage agent*. Data holders host data sets and are end points for a federated query processor. During the linkage process, data holders participate in the linkage process to establish identifiers for their records that can be later used during result aggregation. The linkage agent serves as a semi-trusted service provider to facilitate the entire linkage process, including peer discovery, data holder authentication, establishment of linkage attributes, dissemination of any data harmonization guidelines, forwarding any required data between parties (such as during precomputation), and access to linkage results. We begin our discussion after two authenticated data holders have agreed to link their data sets, and have standardized it as per the guidelines of the linkage agent. We assume that the following conditions hold at all times.

- All messages are communicated through secure channels.
- Messages between a data holder and another data holder moves via the linkage agent.
- The linkage agent is semi-trusted, meaning that the agent follows protocol steps but can attempt to reverse any introduced data anonymization.
- Data holders are semi-honest as well, but are governed by regulations that deter them from providing fake input into the linkage process, or from colluding with the linkage agent.
- Frequency of occurrence of bigrams in attributes of the data set are publicly known.

1.2 | Probabilistic record linkage

Consider two databases D_A and D_B, each containing n_f fields on which the linkage is performed. For example, we use demographic fields such as first name, last name, street address, city and zipcode to demonstrate linkage performance during our evaluation. Let R_A and R_B denote two records in D_A and D_B respectively. We access field i in a record, say R_A, using a standard array notation – $R_A[i]$. The preferred method for record linkage in the presence of data entry errors is a probabilistic linkage methodology[12], where strings are matched for similarity instead of equality. A bigram based approach in this context computes similarity based on the number of commonly occurring substrings of length 2.

Let $\mathcal{B}$ denote the set of all possible bigrams that can be composed using printable characters (letters, numbers and punctuations). Bigrams are uniquely identified using an index between 1 and $|\mathcal{B}|$. Considering only upper case versions of letters, there are 69 printable characters, giving us a total of 4761 possible bigrams. Given two strings $R_A[i]$ and $R_B[i]$ in field i, let $\alpha \subseteq \mathcal{B}$ and $\beta \subseteq \mathcal{B}$ respectively denote the bigrams present in the two strings. A frequently adopted similarity measure is the Dice coefficient metric, given as

$$\mathsf{Dice}(R_A[i], R_B[i]) = \frac{2|\alpha \cap \beta|}{|\alpha| + |\beta|}. \tag{1}$$

Similarity values between fields are combined into a weighted sum to obtain a record similarity measure. If Sim_f denotes a field similarity function, and w_i denotes a non-negative weight for field i, then the similarity between two records is given as,

$$\mathsf{Sim}(R_A, R_B) = \frac{\sum_{i=1}^{n_f} w_i \mathsf{Sim}_f(R_A[i], R_B[i])}{\sum_{i=1}^{n_f} w_i}. \tag{2}$$

A probabilistic record linkage scheme computes similarities between all record pairs, and designates a pair as a match when the similarity crosses a pre-specified threshold. Ties are broken arbitrarily.

1.3 | Notation

We use multiple storage structures in the following discussion. The data stored in these structures may be a single value, or a collection of values. Each structure enforces an implicit ordering of the elements stored, and is indexed using the [ ] array notation. Indexing may be single-dimensional ($[\cdot]$), two-dimensional ($[\cdot,\cdot]$), or three-dimensional ($[\cdot,\cdot,\cdot]$). For example, $L[1,2] = 3$ implies that the element 3 is stored at index $[1,2]$ in the structure named L. When the name of the structure is understood from context (as in a table column), we represent the same as *index: value* (e.g. $[1,2] : 3$).

1.4 | Private record linkage with key rings

A probabilistic record linkage scheme exposes the bigrams corresponding to a string to a linkage agent. Even if bigrams are transformed to an obfuscated representation using standard keyed hash functions, their frequency of occurrence can still leak substantial information for an accurate reconstruction. A commutative one-way function (hard to invert) can be used instead of a keyed hash function to eliminate the requirement for shared keys. Let $\mathsf{OWF}(k, m)$ denote the one-way transformation of a message m using key k. When OWF is commutative, we can assert that

$$\mathsf{OWF}(k_1, \mathsf{OWF}(k_2, m)) = \mathsf{OWF}(k_2, \mathsf{OWF}(k_1, m)). \tag{3}$$

Further, Dewri et al. suggested the use of a key ring to reduce the risk from frequency analysis [11]. A *key ring* is a fixed number of keys used by a party in such a transformation. Let $\mathcal{K}_A = \{k_1, ..., k_{s_A}\}$ and $\mathcal{K}_B = \{k'_1, ..., k'_{s_B}\}$ denote the key rings for two parties A and B. Each party also chooses permutations of the elements $\{1, ..., |\mathcal{B}|\}$, one for each key in its key ring. Let π^i_A and π^j_B denote the permutations chosen by the two parties, with $i = 1, ..., s_A$ and $j = 1, ..., s_B$. Given a bigram $b_t \in \mathcal{B}$ with index t, $\pi^i_\cdot(t)$ then gives the permuted index of t as per the permutation $\pi^i_\cdot$.

In the first step, using a common mapping of bigrams to the message space of the one-way function, each party computes the transformation of all bigrams in $\mathcal{B}$ using all keys in its key ring, and stores them permuted in a structure. We refer to these transformations as level-1 ($L1$) values.

$$L1_A[u, \pi^u_A(v)] = \mathsf{OWF}(k_u, b_v) \tag{4}$$

$$L1_B[u', \pi^{u'}_B(v)] = \mathsf{OWF}(k_{u'}, b_v) \tag{5}$$

where $u = 1, ..., s_A$, $u' = 1, ..., s_B$, and $v = 1, ..., |\mathcal{B}|$. The parties then exchange their $L1$ structures.

In the second step of the precomputation, each party computes the transformations of the $L1$ values it received from the other party. We refer to these transformations as level-2 ($L2$) values. Party A computes its $L2$ structure as

$$L2_A[u, v, w] = \mathsf{OWF}(k_u, L1_B[v, w]), \tag{6}$$

where $u = 1, ..., s_A$, $v = 1, ..., s_B$, and $w = 1, ..., |\mathcal{B}|$. Correspondingly, party B computes as

$$L2_B[v', u', w'] = \mathsf{OWF}(k_{v'}, L1_A[u', w']), \tag{7}$$

where $v' = 1, ..., s_B$, $u' = 1, ..., s_A$, and $w' = 1, ..., |\mathcal{B}|$.

The $L2$ structures are then sent to the linkage agent. Since OWF is commutative, the transformation of the same bigram in $L2_A$ and that in $L2_B$ would be equal for a given key pair from $\mathcal{K}_A$ and $\mathcal{K}_B$ (irrespective of order). We exemplify these steps in the context of our modified approach in Section 3.

A party transforms its database by converting each field value into a set of bigram encodings. Each bigram is encoded by a (key index, permuted bigram index) tuple. A key index is a value between 1 and the size of the key ring, and signifies a key chosen from the key ring. A permuted bigram index is the index of the bigram when $\mathcal{B}$ is permuted using the permutation corresponding to

the key. If the two parties have the same bigram, and choose encodings (i_A, j_A) and (i_B, j_B) respectively, then equality of the two encodings can be determined by comparing the value in $L2_A$ addressed by $[i_A, i_B, j_B]$ and that in $L2_B$ addressed by $[i_B, i_A, j_A]$.

1.5 | Our contributions

Earlier proposals to perform private record linkage using commutative ciphers, including the aforementioned approach by Dewri et al.[11], is based on using a modular exponentiation cipher as the one-way function. As an immediate improvement, we replace commutative transformations based on modular exponentiations with one-way transformations based on elliptic curve operations. The transition from modular exponentiations to elliptic curve point additions and scalar multiplications significantly reduces the precomputation time (to a matter of few minutes in a personal laptop). The second improvement we perform is the elimination of the requirement to send the $L2$ structures (large encryption values) to the linkage agent. Instead, a fast post-processing step (after $L2$ values are computed) is used to generate the data sufficient for the linkage agent to determine bigram equivalence. The linkage agent, with this improvement, does not have to directly compare (long) $L2$ values using expensive memory comparison operators. Since the same bigram may be encoded differently in a multi-key setting, set intersection is quadratic in the number of bigrams in the two sets. The third improvement that we perform allows for quickly checking if a bigram is absent in a set. Guaranteed with a zero error rate, this check eliminates the need to compare the bigram encodings of most pairs of elements in two bigram sets.

In addition to the above modifications, we benchmark a standard quadratic record linkage algorithm on eight different machine configurations, comprising of standalone and cloud-based (Amazon EC2) instances. The results we obtained suggest that a linkage on data sets with a million records each can be completed under 15 minutes on a small compute-optimized Amazon EC2 instance. In terms of privacy assessment, we propose an exposure risk index to assess the residual risk associated with identifying portions of an attribute's value using frequency analysis.

The subsequent sections in this manuscript are organized as follows. Section 2 presents an overview of some related work in the field. Sections 3, 4 and 5 respectively get into details of the precomputation phase, data encoding procedure and residual risk assessment, and the modified linkage process. Implementation details are provided in Section 6, followed by empirical results in Section 7. We conclude the paper in Section 8.

2 | RELATED WORK

Early proposals on private record matching were based on transformations of data values using secure hash functions[5,13,14]. Linkage under such transformations can only succeed if the data values do not contain errors, an aspect that is difficult to enforce. Hence, probabilistic data linkage was proposed where data values are broken down into smaller substrings, bigrams or trigrams, and similarity measures are used to determine if two data values match[15,16]. This method remains a popular choice in the data linkage community, and has been argued to perform at par to deterministic linkage[6,17]. However, hash functions being deterministic in nature, their application in data masking cannot hide frequencies[10]. Other approaches to perform approximate linkage centers around the creation of linkage keys from specific indices of data values[18], using phonetic codes for comparison[19,20], or embedding data values in a multi-dimensional space[21].

Bloom filters have received wide attention in the private record linkage community. A Bloom filter (BF) encoding inserts bigrams into hashed locations in a bit array. Two encodings from two bigram sets are then compared to approximate the intersection size of the sets[6,22]. This method has been cryptanalyzed in detail, and feasible attacks to reverse Bloom filter encodings have been demonstrated. Kuzu et al. present an attack where they formulated a constraint satisfaction problem to map strings from a global database to BF encodings in a private data set[7]. This is achieved by observing the frequency of occurrence of entire strings, or individual bits set by strings, in the global database, and then identifying BF encodings with similar frequencies. The authors report a 11% reidentification on 3500 first names taken from a North Carolina Voters Registration (NCVR) database (the same data source we use in this work). In a follow up study involving patient names in a medical database, the recall rate was found to be 20% when attempting to identify from the 20 most frequent names[23]. However, the authors note that the attack setup requires tuning to balance accuracy and performance. Subsequent works demonstrated reidentification rates of 12% on German names[24] and 76% on NCVR names[25], the latter attack being specific to the case when parties exchange BF encodings without relying on a third party for the linkage. To counter these attacks, multiple BF hardening techniques have been proposed: encode bigrams from more than one field to form composite BFs[26], combine attribute level BFs to record level BFs based on

the weights assigned to different attributes[27], use different number of hash functions to encode different attributes[28], or perform stretching/shrinking operations on the BF encodings to make the bit distributions uniform[29]. Kroll and Steinmetzer demonstrate a 44% reidentification rate (on 100,000 records) when composite Bloom filters are created from separate BF encodings of surnames and town names[8]. However, the attack itself could take days to complete. Recently, Christen et al. presented a targeted frequency-based cryptanalysis attack that aims to identify if the most frequent values from a given set has been encoded in a BF[30]. The attack could reidentify 10 out of 100 most frequent names in the NCVR database. The attack was later improved by the same team[31]. Even with BF hardening techniques in place, the improved attack can identify with more than 90% precision the positions in a Bloom filter that could not have been set by bigrams. This leakage significantly aids in the reidentification process; the authors report reidentification of more than 49,000 BF encodings (NCVR database with different attributes) out of 224,000 records. All attacks discussed here require that the adversary has access to a public database that has a distribution (with respect to the strings and bigrams) similar to the BF encoded database. Bloom filter based techniques are being heavily cryptanalyzed, and subsequently improved to resist against identified flaws. A characterization of the leakage induced by the use of Bloom filters has not been attempted in the area. We position this work as an alternative approach, present a preliminary method for residual leakage, and expect that the leakage analysis can be made more rigorous to direct well-informed parametric choices.

Data encryption is another technique explored for private record linkage. Similarity distances of data values can be tested using a fully homomorphic encryption scheme[32]. However, the cost to do so on large databases is prohibitively expensive. To address this issue, large data sets are divided into smaller ones using a blocking procedure[33,34]. A blocking procedure creates disjoint subsets of the data based on either the value of an attribute or the composition of partial information from multiple attributes. Blocking procedures also need privacy treatment since empty subsets corresponding to certain data values can leak information[35,36]. Homomorphic encryption has also been used to demonstrate fast private set intersection protocols, which is often the core of similarity metrics[37]. It is important to note that the efficiency improvements reported in private set intersection protocols are with respect to performing a few intersections of large sets; the computation and communications costs do not scale when millions of small set intersections have to be performed[38,39]. A more recent work by Khurram and Kerschbaum proposed using secure computation instead for private sorting of Hilbert curve-based locally computed scores, with the objective of reducing the number of comparisons to be performed (using a sliding window approach)[40]. The score comparisons are also performed using secure computation methods. While the security benefits of the method aligns with that of other secure computation methods, and performance is significantly improved over similar relevant works[36], the approach is still limited in scalability. The work reports being able to perform linkage of 4096 records per party in around 400 minutes (including both online and offline phases), which, by linear interpolation, will amount to ~7 days for a dataset of size 100,000 records per party. Comparatively, such a linkage can be done under 3 hours in our approach without multithreading.

3 | KEY RING PRECOMPUTATIONS

Consider a cyclic group $\mathbb{G}$ of (large) prime order n, and generator G. In the following, we realize such a group using an elliptic curve. Point manipulations in $\mathbb{G}$ are then defined using two operations – point addition ($P + Q$, where $P, Q \in \mathbb{G}$) and scalar multiplication (xP, where $P \in \mathbb{G}$ and x is a scalar). All points in $\mathbb{G}$ can also be represented as some multiple of the generator G. However, determining the multiple in polynomial time is believed to be a "hard" problem, and is an instance of the Elliptic Curve Discrete Logarithm Problem (ECDLP) – given $P, Q \in \mathbb{G}$, find k such that $Q = kP$. A one-way function can therefore be constructed based on ECDLP as $\mathsf{OWF} : \mathbb{Z} \times \mathbb{G} \rightarrow \mathbb{G}$, such that $\mathsf{OWF}(k, P) = kP$. Commutativity of OWF also holds in this case – $k_1 k_2 P = k_2 k_1 P$.

3.1 | Key ring generation

A key is an integer less than the order (n) of the group in use. As mentioned above, we use an elliptic curve to realize the cyclic group. The chosen elliptic curve and its parameters are considered public information, known to both parties A and B. Each party decides on the size (number of keys) of the key ring, denoted here as s_A and s_B, and accordingly samples integers uniformly at random to represent the keys in its key ring, denoted respectively by $\mathcal{K}_A$ and $\mathcal{K}_B$. Each party also chooses a pseudo-random permutation of $1, ..., |\mathcal{B}|$, namely π_A and π_B. Note that the use of separate permutations in each key is unwarranted. As we will see later, the precomputation output enables the linkage agent to associate two different encodings (using two keys) of a bigram, hence nullifying the effect of multiple permutations. We will exemplify the steps carried out in the subsequent steps using a toy

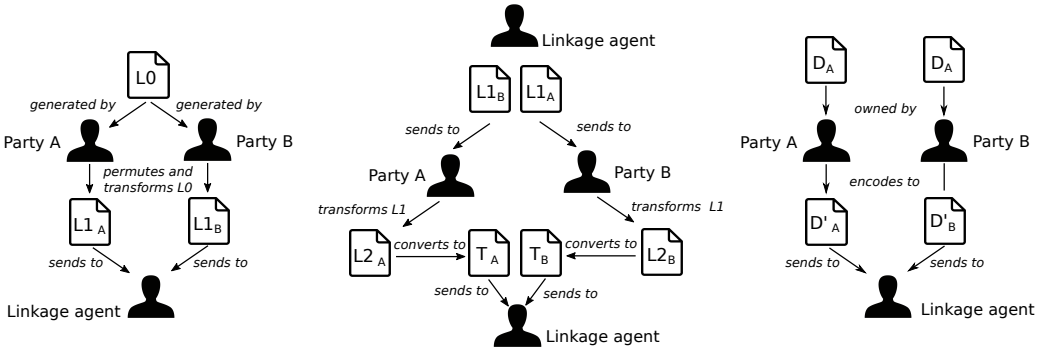


Figure 1 Schematic of the computation and communication workflow. The edge label '*sends to*' implies a network communication. The $L0$ file is a deterministic mapping of bigrams to elliptic curve points. The $L1$ and $L2$ files are results of elliptic curve point multiplications using keys as scalar. The T files are post-processed $L2$ files to reduce communication time. D_A and D_B are raw data files to be linked; D'_A and D'_B are their encoded versions.

example where the universe of characters is restricted to $\{a, b\}$. The complete example is available in Appendix A, and referred to in parts from the following discussion.

Example. Consider the universe of characters to be $\{a, b\}$*, which results in four potential bigrams* $\mathcal{B} = \{aa, ab, ba, bb\}$*. The bigrams are indexed using the numbers* $1, 2, 3$ *and* 4*. Let the key rings at sites A and B be* $\mathcal{K}_A = \{k_1 = 3, k_2 = 5\}$ *and* $\mathcal{K}_B = \{k'_1 = 7, k'_2 = 11\}$ *respectively. Keys can be referred to by their indices, e.g. key* 1 *in* $\mathcal{K}_A$ *is* 3*, key 2 in* $\mathcal{K}_B$ *is* 11*, and so on. Table A.1 shows the permutations that A and B uses for the four bigrams.*

Fig. 1 depicts the data exchanges that happen between the two parties (via the linkage agent) as part of the precomputation. At this point, both parties have a key ring of its own, and datasets with established fields on which the linkage will happen.

3.2 | Level-0 (L0)

The level-0 precomputation uses a mapping algorithm to deterministically map bigram strings to points on the elliptic curve. We store this mapping in the level-0 ($L0$) structure.

$$L0[i] = \langle b_i, P_i \rangle, \tag{8}$$

where $b_i \in \mathcal{B}$ is a bigram and P_i is the corresponding point. The $L0$ structure is indexed by the index of the bigram. We use the notations $L0[i].b$ and $L0[i].P$ to signify the respective values in the tuples stored in $L0$. Each point P_i is generated in a manner that prevents the linkage agent from determining a corresponding scalar t_i such that $P_i = t_i G$, G being the generator of the curve. This property has important security implications, as discussed in Section 3.5. We use the 'try-and-increment' procedure of hashing arbitrary strings to elliptic curve points for this step[41]. The procedure (i) uses a cryptographic hash function to map a bigram string b_i and a counter ctr_i (initialized to one), to an integer x_i in a finite field, (ii) computes y_i as per the elliptic curve equation, and then (iii) checks to see if the point $P_i = (x_i, y_i)$ is on the elliptic curve. If the point is not on the curve, the counter is incremented, and the process is repeated. Given a certain value of ctr_i, we generate the candidate x_i to have the same order as the field's size (number of bits) by using a hash function such as SHA256, and stretching the output as

$$\begin{aligned} X_0 &= \mathsf{SHA256}\big(b_i || ctr_i\big) \\ X_j &= \mathsf{SHA256}\big(X_{j-1} || (ctr_i + j)\big) \\ x_i &= X_0 || X_1 || X_2 || \ldots \text{ (as many times necessary).} \end{aligned}$$

This procedure is simpler to implement than other available methods[42], and is appropriate when there is no constant time execution requirement. The mapping is also a one-time activity for a party. Newer methods[1] are in the works currently, and adopting them remains an alternative in the future.

[1] https://datatracker.ietf.org/doc/draft-irtf-cfrg-hash-to-curve/

Example. The four bigrams in our example are numerically mapped as $L0[1] = \langle aa, 10\rangle$, $L0[2] = \langle ab, 20\rangle$, $L0[3] = \langle ba, 30\rangle$ *and* $L0[4] = \langle bb, 40\rangle$. *This simplified representation is for demonstration only; the bigrams would typically be represented as points on an elliptic curve.*

3.3 | Level-1 (L1)

As part of the first step in the precomputation, each party computes one-way transformations of the curve points representing the bigrams in $L0$, and stores them in the level-1 ($L1$) structure in a permuted manner. Transformations are performed using the one-way function based on elliptic curve point multiplication with a scalar.

$$L1_A[u, \pi_A(v)] = \mathsf{OWF}(k_u, L0[v].P) \tag{9}$$

$$L1_B[u', \pi_B(v)] = \mathsf{OWF}(k_{u'}, L0[v].P) \tag{10}$$

where $u = 1, ..., s_A$, $u' = 1, ..., s_B$, and $v = 1, ..., |\mathcal{B}|$.

Example: Table A.2 shows the $L1$ *precomputations for A and B. The* OWF *value is computed in the example as the product of the key and a message; typically, this would be a point multiplication operation that is difficult to invert. The* $L1$ *structures are obtained by permuting the* OWF *values using the permutations from Table A.1.*

Both parties exchange their $L1$ structures using the linkage agent as a communication channel.

3.4 | Level-2 (L2)

Upon receiving a $L1$ structure, a party repeats the same procedure, but without applying any permutations. The computation is performed on the $L1$ values as per Eqs. (6) and (7). Owing to the commutative property of OWF, for any bigram $b_i \in \mathcal{B}$, we have $L2_A[u, v, w] = L2_B[v', u', w']$ when

$$\begin{aligned} u &= u', v = v', \\ w &= \pi_B(i), \text{ and } w' = \pi_A(i). \end{aligned} \tag{11}$$

Therefore, if party A provides a key index and a permuted bigram index (u', w') in lieu of a bigram, and party B provides the same as (v, w), a linkage agent can check for equivalence of the represented bigrams by comparing the two values $L2_A[u, v, w]$ and $L2_B[v', u', w']$.

At this point, the two parties can send their $L2$ structures to the linkage agent and finish the precomputation. However, comparisons of transformations (long memory bytes representing the curve points) are relatively more expensive than performing a lookup to determine equivalence. As such, we post-process the $L2$ values by sorting them based on the points they store. Sorting is done on a serialized byte representation of the elliptic curve points provided by the OpenSSL library. For a sufficiently large group order, there is negligible probability for two $L2$ values to represent the same point unless the two keys in the nested OWF transformations and the represented bigrams are the same. Hence, independent sorting of $L2_A$ and $L2_B$ result in an ordering of the points such that, if $L2_A[u, v, w]$ and $L2_B[v', u', w']$ are the values at the same position after the sortings, then it should hold (except with negligible probability) that the two values are equal. Alternatively, it holds that $u = u'$, $v = v'$, and both values represent the same bigram. Therefore, instead of sending the $L2$ values to the linkage agent, each party sends the indexing triples ordered based on the values (points) stored in them. We denote these indexing triples as T_A and T_B for A and B respectively. The indexing triples are independent of any data sets, and can be used for the linkages of multiple data sets between two parties.

Example. Tables A.3 shows the computation of the $L2$ *structures for A and B. During post-processing, the* $L2$ *values are sorted, and their position prior to sorting are also noted. Observe that the sorted* $L2$ *values are indeed the same in both subtables. The indexing triples are shown in the last column. The last two columns in Table A.4 confirm the assertion that two triples at the same position are indeed encoding the same bigram.* $\bar{\pi}$ *represents a reverse permutation that, given a permuted bigram index, returns the true index.*

Parties A and B send the respective indexing triple structures T_A and T_B to the linkage agent. Observe that, in the example given in Table A.4, every bigram encoding from a party (say (1,3) from A) has an equivalence relationship established with every 'possible' encoding of the same bigram from the other party. Therefore, by transitivity, a linkage agent can infer which encodings from a certain party refer to the same bigram, irrespective of using a separate permutation for each key. The use of multiple keys do have an impact on the data encoding and exposure risk (Section 4).

3.5 | Security

The level-0 precomputation describes the mapping between bigrams and points on the elliptic curve. Consider a mapped point $\langle b_i, P_i \rangle$. We require that the linkage agent is unable to determine a scalar t_i such that $P_i = t_i G$. Given P_i as the point (x_i, y_i) on the curve, it is intractable to determine t_i due to the difficulty of ECDLP.

In the level-1 precomputations, the permutations remove any association between the order of values in the $L1$ structures and the transformed bigrams. The inability to reverse these permutations depends strongly on the hardness of ECDLP. For a key k, the $L1$ values for two arbitrary bigram points $P_i = t_i G$ and $P_j = t_j G$ are $M_i = k t_i G$ and $M_j = k t_j G$ respectively. If t_i and t_j are easily recoverable from the respective bigram points, then an adversary (the linkage agent or another data holder) can check if $\frac{t_j}{t_i} M_{i'} \stackrel{?}{=} M_{j'}$, for all pairs of $L1$ values $M_{i'}$ and $M_{j'}$. There is a non-negligible probability that the condition will be satisfied only when $i' = i$ and $j' = j$, thereby revealing the encoded bigram.

In general, we require that an adversary is unable to distinguish between the tuples (P_i, P_j, kP_i) and (P_i, P_j, kP_j) when k is secret; otherwise, the distinction reveals the correspondence between a $L1$ value and its underlying bigram. Due to the difficulty of ECDLP, one cannot simply solve for k in either of the two tuples, and then verify on the other tuple. Nonetheless, assume that the adversary has some algorithm that, given the two tuples in a random order, can output which of the two is of the form (P_i, P_j, kP_i). Without loss of generality, say the algorithm outputs that the first input tuple is of the form (P_i, P_j, kP_i). When the elliptic curve is over a large non-trivial (sub)group of prime order, then each point is a generator. Using P_i as the generator, we can then express the point kP_j as $k'P_i$, for some scalar k'. Hence, the second tuple is equally likely to be of the output form. Therefore, the output of the adversary's algorithm is equivalent to a random guess at best.

The linkage agent receives a set of indexing triples from each party after the level-2 precomputations. Under our threat model, a data holder does not collude with the linkage agent. However, if a collusion is present, the linkage agent would know the permutation used by the colluding party. Thereafter, by equivalence of the indexing triples in T_A and T_B, the linkage agent also learns the permutations used by the non-colluding party.

4 | DATA ENCODING

Data encoding involves representing the bigrams in a record's field to a set of key index and permuted bigram index pairs. For a field value $R_A[i]$, the string is first converted to a multiset α of its bigrams. For each bigram $b_i \in \alpha$, a key k_u is chosen from the key ring $\mathcal{K}_A$, and the bigram is encoded as the pair $(u, \pi_A(i))$. In addition, to achieve frequency smoothing, a small number of randomly chosen bigram encodings may be inserted for each string. The encoded bigrams are then shuffled to form the encoded record field. Party B follows a similar procedure using its own key ring and permutation.

4.1 | Frequency smoothing

The number of keys used to encode a specific bigram has direct implications on the ability to infer the bigrams based on how frequently a specific encoding is observed. Therefore, a frequency smoothing procedure is used to distribute high frequency bigrams over a larger number of keys such that an encoded value appears at a uniform rate across the entire data set. Frequency smoothing is done on a per field basis to account for differences in apriori distributions across the vocabularies of different fields (e.g. a person's name versus a city's name).

Frequency smoothing within a field begins by first computing the maximum frequency of a bigram in the field. Let $f_i(b)$ denote the normalized frequency (between 0 and 1) of a bigram $b \in \mathcal{B}$ in field i. Then, given a key ring $\mathcal{K}$, a party can choose keys in the encoding such that a key is used for

$$\delta = \frac{\max_{b \in \mathcal{B}} f_i(b)}{|\mathcal{K}|} \tag{12}$$

fraction of occurrence of a bigram. This is accomplished by uniformly choosing between key index 1 to $\kappa = \lceil f_i(b) / \delta \rceil$ (round up) during the encoding. Note that, when requiring κ keys, we choose them uniformly from the first κ keys – key numbers $1, 2, ..., \kappa$. Keys are not chosen randomly since the bigrams will then get proportionately distributed across the keys.

Example. Consider an attribute where any given bigram appears at most 10% in all bigram occurrences. With a key ring of size 50, we then have $\delta = \frac{0.1}{50} = 0.002$. *All bigrams that appears with 0.2% or lower frequency will always use key 1. A bigram that appears with 10% frequency will use keys 1 through* $\lceil 0.1 / 0.002 \rceil = 50$. *Another bigram which appears, say with 3.5% frequency, will use keys 1 through* $\lceil 0.035 / 0.002 \rceil = 18$.

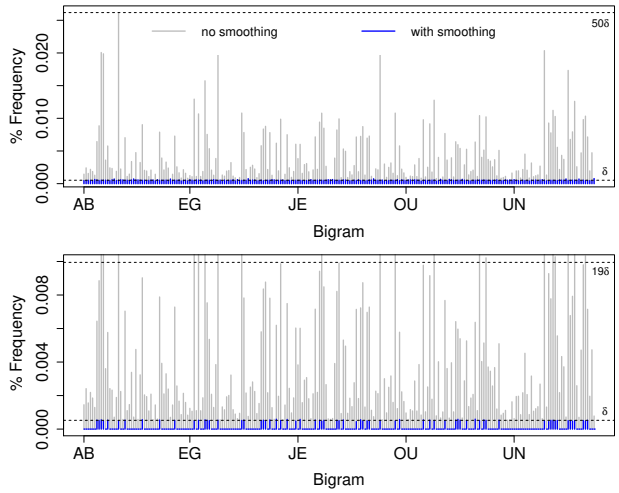


Figure 2 Frequency smoothing on bigrams with frequencies greater than δ. Top plot shows frequencies in key 1 (out of 50); bottom plot is for key 10 (shows limited portion of y-axis for clarity). Frequencies before smoothing depict values seen in typical first names of individuals.

Therefore, high frequency bigrams get distributed over more keys than low frequency bigrams. Fig. 2 shows the effect of the smoothing process on bigram distributions corresponding to typical first names. As expected, the distribution within a key is close to uniform for those bigrams that use the key; however, not all bigrams use all keys (bottom figure shows that only a few bigrams use key 10). We also require that the frequency of occurrence of a bigram b using its κ keys is close to $\kappa\delta$. This ensures that all bigrams using a specific number of keys are not distinguishable based on their total number of occurrences. For a bigram appearing with frequency $f_i(b)$ and using κ keys, the gap in frequency $g(b) = \kappa\delta - f_i(b)$ is covered using random insertion of the bigram across the field values. This can be achieved by inserting an arbitrary bigram b as part of a string with probability $g(b)/\sum_{b_j \in \mathcal{B}} g(b_j)$. Note that $g(b)$ is the distance of $f_i(b)$ from the closest multiple of δ (greater than $f_i(b)$); therefore, smaller the δ, or bigger the key ring size $|\mathcal{K}|$, the smaller is the gap. For a typical first name field, δ is observed to be $\approx 0.05\%$ with a key ring of size 50. Under this setting, 75% of the first names had one or no additional bigram added to them; in fact, our record matching results are exactly the same with or without the random bigram insertion.

4.2 | Exposure risk

While frequency smoothing flattens the frequency of occurrence of bigram encodings, information can still leak through the number of keys used for a specific bigram. We present here our analysis framework to assess the residual exposure risk after frequency smoothing is applied. Our adversarial model assumes an adversary that is aware of the field-wise distribution of bigram frequencies specific to the database used by a party. The key ring size is also assumed to be known to the adversary. Hence, the adversary can compute the δ value, thereby the exact number of key indices that are used in a bigram encoding. We compute the exposure risk in terms of the probability of identifying a bigram from its encoded values, and subsequently the chances of identifying a specific fraction of bigrams that are present in a field's value.

Note that if all bigrams naturally appear with equal (or similar) frequencies, it is not possible to differentiate between bigrams based on their frequency of occurrence. Let $f(b)$ denote the occurrence frequency of bigram b known to the adversary in an arbitrary field. We therefore measure the prior exposure of a bigram in the field as the reciprocal of the number of bigrams that occurs with the same frequency.

$$\Pr_{pre}(b_i) = \begin{cases} \frac{1}{|\{b_j | f(b_j) = f(b_i)\}|} & , f(b_i) \neq 0 \\ 0 & , f(b_i) = 0 \end{cases} \tag{13}$$

Therefore, if all bigrams appear with unique frequency values (the notion of similar frequency is hereby discarded), they can be trivially identified irrespective of any deterministic transformation applied on them.

For bigrams where only one key index will be used during encoding (includes bigrams with frequencies less than δ), the posterior exposure (after frequency smoothing) remains the same as the prior exposure. All remaining encodings will be observed

with a frequency of $\approx \delta$, thereby making it difficult to differentiate them based on frequencies. Instead the differentiation is made based on the number of key indices used. For a bigram b_i using κ_i keys, the uncertainty faced by the adversary depends on how many other bigrams uses the same number of keys. Hence, we define the posterior exposure of b_i as

$$\Pr_{post}(b_i) = \begin{cases} \frac{1}{|\{b_j | \kappa_j = \kappa_i\}|} & , \kappa_i > 1 \\ \Pr_{pre}(b_i) & , \kappa_i = 1 \end{cases}. \tag{14}$$

Given a string s in a field made of bigrams $b_{i_1}, b_{i_2}, ..., b_{i_l}$ arranged in decreasing order of their exposure probability, we compute the exposure risk associated with identifying the m ($\leq l$) highest exposed bigrams as

$$\text{Exposure}_m(s) = \prod_{j=1,...,m} \Pr_{mode}(b_{i_j}), \tag{15}$$

where *mode* is '*pre*' or '*post*' depending on prior or posterior exposure risk. We acknowledge that this analysis does not account for incremental inferences where identification of one bigram in a string changes subsequent probabilities of occurrence of other bigrams, and higher order n-grams in general.

5 | LINKING WITH HASH MAPS

A basic linkage procedure considers one record at a time in a data set, computes its similarity with every other record in the other data set, and stores the identifier (index) of the record with the maximum similarity along with the similarity score. For similarity computation, a private set intersection is performed on the bigram sets of corresponding fields using the bigram encodings and the precomputed indexing tuples.

5.1 | Linkage map

We begin the private linkage of two datasets by first merging the precomputed indexing triples T_A and T_B into a single lookup table T, henceforth called the *linkage map*. Given entries $T_A[i] = (u, v, w)$ and $T_B[i] = (v', u', w')$ (recall that they represent the same bigram, and $u = u'$ and $v = v'$), an entry is created in the linkage map as

$$T[u, v, w] = w'. \tag{16}$$

Two arbitrary bigrams with encodings (u_A, w_A) from party A and (u_B, w_B) from party B are then the same if $T[u_A, u_B, w_B]$ is equal to w_A.

Example. The third column in Table A.4 shows the linkage map constructed for our example. The last two columns show the equivalence of bigrams based on their encodings.

The advantage of the linkage map over comparing $L2$ values can now be seen. $L2$ values are long sequences of bytes and expensive to compare (linear in the number of words in a serialized elliptic curve point, or a hash of the representation), especially when it has to be done possibly millions of times during the entire linkage process. The linkage map is a simple lookup table with a constant cost of reading one word. Large portions of the linkage map can also fit in different levels of the cache, thereby making their access faster.

5.2 | Hash maps

A majority of the checks for equivalence of two bigrams is likely to return a negative answer. Even when the strings (of length l) are exactly the same in a field, the number of bigram matches is equal to $(l - 1)$, whereas the number of comparisons required is l^2 owing to the shuffling of the elements after encoding. Hence, the ability to quickly decide whether a bigram from one encoded set can at all match any of the bigrams in the other set can have vast implications on the linkage time. We propose here an optimization to facilitate such an ability.

Let $\mathcal{B}_A$ and $\mathcal{B}_B$ denote the set of bigram encodings that parties A and B would use. Consider a hash function $Hash$ that outputs integers in the range $[1, h]$. For each $(u, w) \in \mathcal{B}_A$, we create a bitmap $H_{(u,w)}$ of size h, such that bit t is set (= 1) if there is any $(v', w') \in \mathcal{B}_B$ with $Hash((v', w')) = t$ and $T[u, v', w'] = w$. In other words, a bitmap is created for every possible bigram encoding from party A; a bit (not necessarily unique) is set in the bitmap corresponding to each of party B's bigram encodings

that represent the same bigram as A's. We refer to these bitmaps as *hash maps.* Each hash map can be computed during creation of the linkage map.

Example: Figure A.1 shows the creation of the hashmaps corresponding to each of the eight (2 keys, 4 bigrams) possible bigram encodings in A. According to the linkage map in Table A.4, the bigram $(1,1)$ *in A is equivalent to the bigrams* $(1,3)$ *and* $(2,3)$ *in B. Hence,* $H_{(1,1)}$ *is created by hashing* $(1,3)$ *and* $(2,3)$ *and setting the corresponding bits. The other hash maps are similarly constructed. Although eight hash maps are shown, there are only four unique hash maps in this case.*

Consider the instance when a record from party A is being compared for similarity against the records from party B. Since party A's record is static in these comparisons, the set of bigram encodings used in each field also stays the same. Let $\alpha_1, \alpha_2, ..., \alpha_{n_f}$ be the bigram encoding sets for the n_f fields in party A's record. For each α_i, we create the bitmap $\mathcal{H}_i$ by merging (bitwise-or) the hash maps corresponding to the bigram encodings in α_i.

$$\mathcal{H}_i = \vee_{(u,w)\in\alpha_i} H_{(u,w)} \tag{17}$$

$\mathcal{H}_i$ is then a composite hash map for all possible bigram encodings from B that can match some bigram encoding from A in field i of the specific record. Therefore, when performing the set intersection in field i, we consider bigram encodings one at a time from B, and compare it with the encodings from A only if a membership query on $\mathcal{H}_i$ (using B's bigram) is positive. A membership query asks the question $\mathcal{H}_i[Hash((v', w'))] \stackrel{?}{=} 1$ for some encoding (v', w') from B. If a query returns a negative answer (bit is not set), then by construction, no bigram in A's set can match B's bigram in question. The false negative rate is therefore zero. However, if the query returns a positive answer (bit is set), a match is not guaranteed since collisions in the hash function might have led to the bit being set. Nonetheless, we would want that if the query result is positive, a match is indeed present in most cases, i.e. the true positive rate should be high. The composite hash maps $\mathcal{H}_1, ..., \mathcal{H}_{n_f}$ are created once for every record of A, used during the similarity computation of the record with all records in B, and then discarded.

Example. Figure A.1 shows the comparison of the bigram sets {ab, bb, bb} and {aa, aa, ab} using hash maps. First, using the encodings used by A for the three bigrams ab, bb, and bb, we identify three hash maps and create a composite hash map by OR-ing them. Therefore, the composite hash map has bits set corresponding to hashes of $(1,3)$*,* $(2,3)$*,* $(1,4)$ *and* $(2,4)$*, the different ways B can encode ab and bb. We then use a bigram encoding from B's string, say* $(2,3)$*, hash it, and see if the corresponding bit is set in the composite hash map. The process is repeated for the other bigrams. If no collision happened during creation of the composite hash map, the lookup will immediately reveal that bigrams* $(2,2)$ *and* $(1,2)$ *in B, i.e. aa in both cases, do not appear in A's string.*

5.3 | Hash map size

The true positive rate for a composite hash map depends on the size h (bits) of the hash maps. If h is too small, too many collisions will occur, thereby resulting in queries returning a positive answer in most instances. Making h arbitrarily large will increase the memory demand. A total of $|\mathcal{K}_A| \times |\mathcal{B}|$ hash maps are created, and at most $|\mathcal{K}_B|$ bits will be set in each hash map. A few of these hash maps will be combined to form the composite hash maps.

Let q be the average number of bigrams in a field. q hash maps of size h bits will then be merged to form a composite hash map of size h bits. A composite hash map is queried at q locations. Collisions in the hash maps at locations other than those q locations do not impact the query answer. Hence, a hash map needs to have a size large enough that the composition of q hash maps can avoid at least q collisions. We compute the probability of this event under the assumption that the hash function produces a uniformly distributed output. Assuming the worst case where at most $q(|\mathcal{K}_B| - 1)$ locations may already have been set by encodings that are not queried, the first of the q queried locations has no collision with probability $\frac{h-q(|\mathcal{K}_B|-1)}{h}$, the second with probability $\frac{h-q(|\mathcal{K}_B|-1)-1}{h}$, and so on. Therefore, the probability that each of the q queried locations is not set by multiple encodings is computed as

$$\Pr(\text{no collision} \geq q) \geq \prod_{i=0}^{q-1} \frac{h - q(|\mathcal{K}_B| - 1) - i}{h}. \tag{18}$$

We set the hash map size h as a multiple of $|\mathcal{K}_B|$, i.e $h = x|\mathcal{K}_B|$ for some positive integer x, and determine the smallest x such that

$$\prod_{i=0}^{q-1} \frac{x|\mathcal{K}_B| - q(|\mathcal{K}_B| - 1) - i}{x|\mathcal{K}_B|} \geq p \tag{19}$$

for a desired true positive rate p. We find x by progressively increasing its value until the condition is met.

Table 1 Configurations of machines used to benchmark execution times.

Name	Type	CPU	vCPUs	Memory
S1	Apple Mac Pro	Intel Xeon E5-1620v2 3.7GHz	8	16GB
S2	Dell Precision T5810	Intel Xeon E5-1620v4 3.5GHz	8	8GB
S3	Dell Precision T5820	Intel Xeon W-2155 3.3GHz	20	16GB
S4	Dell PowerEdge R731	Intel Xeon E5-2695v4 2.1GHz	36	128GB
C1	AWS EC2 c5.2xlarge	Intel Xeon Platinum 8124M 3.0 GHz	8	16GB
C2	AWS EC2 c5.4xlarge		16	32GB
C3	AWS EC2 c5.9xlarge		36	72GB
C4	AWS EC2 c5.18xlarge		72	144GB

6 | IMPLEMENTATION DETAILS

We implemented the precomputation logic, data encoding, and linkage procedure as three separate multi-threaded C/C++ programs. The precomputation program uses the OpenSSL library (libcrypto 1.1.0) for elliptic curve operations under the secp224r1 curve[43]. Since we perform multiplications of different scalars with the same point, we custom implemented the w-ary non-adjacent form (wNAF) point multiplication procedure to avoid repeated precomputations within the method. Random number generation is performed using OpenSSL's RAND_bytes function, and the Knuth shuffle algorithm[44] is used to generate permutations. OpenSSL's EC_POINT_point2oct function is used to serialize elliptic curve points to an octet representation. The data encoding program reads data sets stored in CSV format, and field weights stored in a text file, and generates the encoded version with frequency smoothing applied. Bigram encodings (a pair of values) are represented using a bit string of length $\lceil \log_2(\text{key ring size}) \rceil + 13$ ($2^{13} > |\mathcal{B}|$) rounded to the nearest multiple of 4. The linkage program performs linkage map and hash map setups before linking two specified data sets. The 128-bit MurmurHash3 hash function is used for hash map related computations. The precomputation and data encoding programs are client side, while the linkage program is server side. We have not implemented any networking to transfer data between the client and the server at this point. The file sizes produced are relatively small for any bandwidth concern. All programs are compiled with the -Ofast and -march = native flags in g++ version 5.4 , with pthreads as the threading library. Threading is implemented using the Threadpool pattern.

7 | EMPIRICAL EVALUATION

The North Carolina Voters Registration (NCVR) database contains more than 7 millions records with demographics data that includes individual names, their mailing address, phone number, gender, and age among other attributes. We create subsets of this database in our empirical evaluation. The subsets are created by random sampling of records with five attributes – first name, last name, street address, city and zip code. The underscore character '_' is added as a prefix and a suffix to all values. The default data set contains 100,000 records for each party, with a 25% overlap. Other subsets of varying sizes – 1000, 5000, 10,000, 50,000, 500,000, and 1,000,000 records – with 25% overlap are also created to study the variation in execution time. All attributes except the city and zip code in one data set of each pair are modified to simulate data entry errors such as erroneous character insertion/deletion, missing attribute, character substitution, transposition of adjacent characters, and entry of data in the wrong field. These errors are inserted with probabilities of 15%, 15%, 35%, 5%, 20% and 10% respectively. For a relatively clean, moderately clean and a dirty data set, we introduce such errors in 5%, 30% and 50% of the records respectively. The file sizes of the encoded data sets ranged from 200KB to 230MB between the smallest and the largest sets; the files are compressed further to approximately half the size. A default key ring size of 50 and hash map true positive rate of $p = 0.9$ is used. Fields are assigned equal weight, and we use the Dice coefficient metric for similarity computations.

We run linkage tasks on the data sets in four standalone machines and four Amazon AWS EC2 compute optimized instances. Standalone machines run linkage tasks only (besides standard operating system processes), and EC2 instances are configured as on-demand virtual machines on a shared platform.

Table 1 lists the basic configuration of the eight machines. All systems run Ubuntu Server 18.04 as the operating system. All client side tasks (precomputation and data encoding) are carried out using 4 threads on a 2015 Macbook Pro laptop with a 3.1 GHz Intel Core i7 CPU and 16 GB memory. Execution time results are averaged over three independent runs.

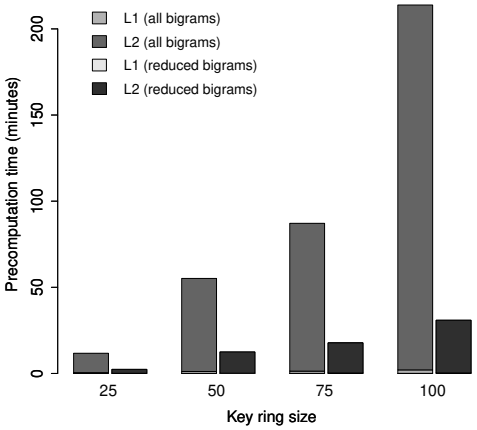


Figure 3 Run time of level-1 ($L1$) and level-2 ($L2$) precomputations. Reduced bigrams indicate bigrams composed only of letters and numbers.

7.1 | Computational/communication cost

7.1.1 | Precomputation

The level-0 ($L0$) file is created once by the linkage agent, taking approximately 3.35 seconds using a single thread in machine S1. The size of the generated file is 319 KB. The maximum counter value observed during the mapping is 17, with most mapping happening successfully with a value of 1. Fig. 3 shows the execution time for level-1 ($L1$) and level-2 ($L2$, including post-processing) precomputation for varying key ring sizes. The level-1 precomputation is relatively small and does not benefit much from threading. The level-2 precomputation repeatedly performs the same amount of work as in a level-1 precomputation for each key in a key ring, and benefits significantly from parallelization. The $L1$ set file sizes produced are in the range of 15-70 MB for 25 to 100 keys, and between 10-190 MB for the indexing triples. Assuming an upload bandwidth of 20 Mbps, the transfer time of $L1$ and $L2$ precomputation results (100 keys) to the linkage agent would be around 28 seconds and 40 seconds respectively, plus network latency time. For comparisons, a similar precomputation based on a modular exponentiation based one-way transformation takes 9.6 minutes for the $L1$ phase and 8.4 hours for the $L2$ phase; the $L2$ files are also much larger in size (>5 GB), which may be heuristically reducible by bit stripping [11]. The precomputation we performed considers all possible bigrams that can be created from printable characters. However, a large fraction of those bigrams (especially ones with combinations of punctuation characters) are rarely seen in data strings. For example, only 600 out of the 4761 possible bigrams cover 96% or more of the bigrams seen in all street addresses (in our data sets). Therefore, if we restrict the bigrams to a reduced set, such as consisting of only letters and numbers, the precomputation time is further reduced.

7.1.2 | Linkage

Table 2 lists the linkage times corresponding to different data sets. Note that the largest of the data sets we use (a million records) can be uploaded to a linkage agent in less than a minute using a 20 Mbps connection. With 16 to 20 vCPUs (S3 and C2), a quadratic linkage of $10^5 \times 10^5$ record pairs is achieved under 9 minutes. With 8 vCPUs (S2 and C1), we achieve run times in the neighborhood of 16 minutes. Improvement in individual processor features (speed, cache size, and bus speed) provide a significant boost as well, such as in S4 versus C3, or S1 versus S2, where the number of vCPUs are equal. Small data sets do not benefit much from the parallelization; threadpool creation time in fact dominates in such cases. Our implementation also reads the entire encoded data sets into memory before starting the linkage process. Hence, once the linkage routines begin (after data loading and setup), we do not have to make any system calls except for join operations on threads. This allows the linkage process to run at native speed even in the virtualized platforms. Linkage of larger data sets, such as a million records each, can be achieved in reasonable time by using more vCPUs (more cost), but may not scale (linkage time may not be acceptable) when data sets reach sizes of 10 or more millions. However, such massively large datasets often require further treatment before linking. We discuss this in Section 7.5.

Table 2 Linkage time in different machine configurations. Data sets have 25% overlap and 30% of the records in one data set has errors. s: seconds, m: minutes, h: hours.

	Number of records in a dataset						
Machine	1000	5000	10,000	50,000	100,000	500,000	1,000,000
S1	0.39 s	3.86 s	13.65 s	5.82 m	20.61 m	7.43 h	28.94 h
S2	0.37 s	5.32 s	18.58 s	6.03 m	17.68 m	6.47 h	24.7 h
S3	0.33 s	4.04 s	8.43 s	2.37 m	6.73 m	2.71 h	8.83 h
S4	0.50 s	2.89 s	8.91 s	2.21 m	6.95 m	2.64 h	8.91 h
C1	0.34 s	3.97 s	11.76 s	4.24 m	15.67 m	5.38 h	20.34 h
C2	0.29 s	2.42 s	9.07 s	2.76 m	8.95 m	3.11 h	11.02 h
C3	0.30 s	1.79 s	6.34 s	1.96 m	3.29 m	1.78 h	5.35 h
C4	0.54 s	2.82 s	6.99 s	1.48 m	3.57 m	1.05 h	3.81 h

Table 3 Comparative linkage time on machines S2 and C1. Data sets have 25% overlap and 30% of the records in one data set has errors. PHMK and PHSK are the Pohlig-Hellman based multi-key and single-key approaches in[11] using 2048-bit primes. s: seconds, m: minutes, h: hours.

		Number of records in a dataset						
Approach	Machine	1000	5000	10,000	50,000	100,000	500,000	1,000,000
PHMK	S2	5.19 s	49.82 s	197.32 s	58.7 m	229.8 m	70.6 h	not attempted
PHSK	S2	1.74 s	5.66 s	22.35 s	6.1 m	20.3 m	7.51 h	32.7 h
this work	S2	0.37 s	5.32 s	18.58 s	6.03 m	17.68 m	6.47 h	24.7 h
PHMK	C1	4.19 s	39.41 s	130.1 s	41.67 m	156.49 m	60.32 h	not attempted
PHSK	C1	0.49 s	4.73 s	13.84 s	4.77 m	19.34 m	6.5 h	25.57 h
this work	C1	0.34 s	3.97 s	11.76 s	4.24 m	15.67 m	5.38 h	20.34 h

In comparison, an approach that does not use hash maps and compares $L2$ values for bigram equivalence[11] takes close to 4 hours to complete for data sets with 100,000 records (5 fields) in each. This runtime is for a key ring of size 50, and on the standalone machine S2. The linkage time in the same setting clocks around 20 minutes when using a single key (10 clusters). Table 3 lists the linkage times for other data set sizes, as well as on the C1 EC2 instance. The case of a million records in each data set was not explored as the run time is estimated to take more than a week. The current approach clearly provides a significant improvement in linkage time when compared with the multi-key approach in[11]. The improvement is not as dramatic when compared with the single-key approach. Recall that the single key approach requires choosing clustering parameters and relies on dummy records to reduce the risk of statistical attacks. Further, based on the asymptotic bounds provided in[45], we conclude that state-of-the-art private set intersection (PSI) methods would take more than a month to complete a 10^{10} record pair linkage task.

7.2 | Hash map efficiency

In order to demonstrate the improvements produced by using hash maps, we collected four values during a linkage run on the default data set – (i) the number of bigram comparisons that would be performed if hash maps are not used, (ii) the number of comparisons performed while hash maps are in use, (iii) the number of times a query on a composite hash map returned a positive answer, and (iv) the number of times a positive query resulted in actually finding a match for a bigram. The first two values allow us to compute the percentage of comparisons that we avoided by using hash maps, and the last two values provide the observed true positive rate of the queries. Fig. 4 shows these two statistics for different settings of the desired true positive rate (parameter p). The hash map size is computed as discussed in Section 5.3 assuming an average of 10 bigrams per field. The number of comparisons avoided ranges from 72% to 89% as the hash map size is increased. The absolute number of comparisons that would have been performed without hash maps in this case is 454,249,862,241. As expected, the observed

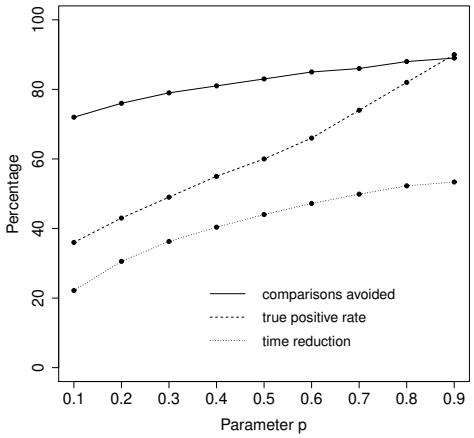


Figure 4 Efficiency of hash maps in avoiding bigram comparisons and reducing the execution time. Parameter p is the desired true positive rate of a membership query in a composite hash map.

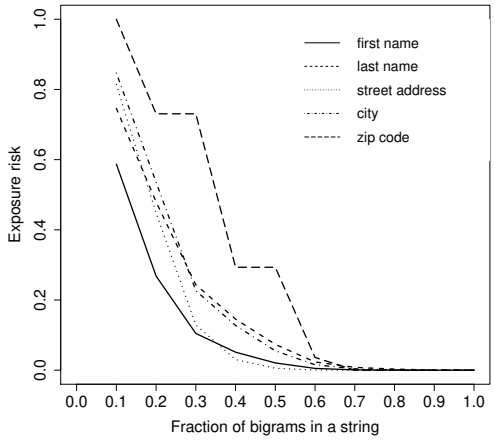
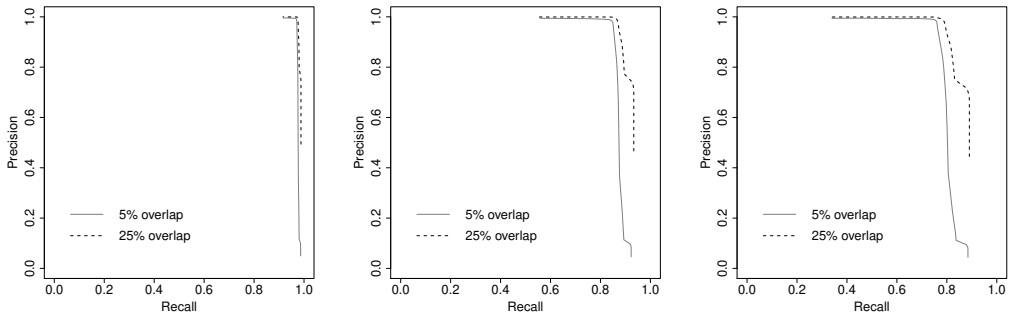


Figure 5 Exposure risk associated with identifying a given fraction of bigrams in a field's value. Key ring size = 50.

true positive rate also grows as larger hash maps are used, and is better than the desired rate in most cases. Overall, the usage of hash maps provides us a 22% to 53% reduction in run time over the case where they are not used (only the linkage map is used). The correlation between time reduction and hash map size are consistent when using different number of vCPUs. The factor to consider when using larger hash maps (e.g. $p = 0.9$) is the resulting increase in memory requirements.

7.3 | Exposure risk

Fig. 5 shows the exposure risk associated with identifying a varying fraction of bigrams in a field's value. The key ring size is set to 50 in this case. In the NCVR data set, the five fields have an average length of 6, 7, 15, 9 and 5 characters. As is evident, correct identification of 60% or more of the bigrams in a string is improbable in all the fields. However, there are high chances that 10% of the bigrams in a string could be identified based simply on the number of keys used in a bigram encoding. Given the string lengths, this amounts to identifying one bigram in the strings, which in our opinion is not sufficient to reconstruct the field's value. For shorter strings made from a smaller alphabet (e.g. zipcode), identification of 40-50% of the bigrams can be considered significant; hence, a lower exposure risk is desirable. Exposure risk can be traded for increased precomputation time by using a larger key ring.



(a) Low error (5% of records) **(b)** Moderate error (30% of records) **(c)** High error (50% of records)

Figure 6 Precision and recall of a quadratic record linkage procedure for varying similarity thresholds in [0, 1]. (a) Low error (5% of records). (b) Moderate error (30% of records). (c) High error (50% of records). Equal weights are assigned to all fields.

7.4 | Linkage accuracy

Linkage performance is typically measured using precision and recall. Precision is the fraction of correct matches in the total number of pairs returned as a match. Recall is the fraction of record pairs from the ground truth that have been identified as matches by the linkage algorithm. Fig. 6 shows the precision-recall trend for matching thresholds in the range of 0 to 1. A record pair is declared a match if the similarity score is at least the threshold value. We use data sets with 100,000 records in each for these plots. Precision and recall can be maintained at more at 96% for data sets with low error rates by using a high threshold (0.9 in this case). However, recall rates drop to around 83% for such a high threshold when date errors are moderate. We observed that maintaining both a high precision and a high recall is more difficult in the data sets with low overlap. For example, a threshold of 0.8 results in 86% recall and 99% precision for the data sets with 25% overlap (moderate error), while the same threshold results in a lower 83% precision when having 5% overlap in the records. A similar observation also holds for data sets with high error rates; the threshold must be lowered further to retain similar precision and recall.

In comparison, a non-private approach that computes similarity scores directly from the bigram sets, also retains a precision of 99% and recall of 86% on the same dataset (threshold of 0.8). This is expected since our approach only generates cryptographic transformations for the bigrams, and does not alter the linkage process. Except for the injection of a small number of random bigrams (frequency smoothing), all optimizations performed in the approach are lossless, i.e. they do not introduce the possibility of not comparing a record pair that would have otherwise been compared in the non-private method. For another comparison, we implemented the Cryptographic Long-term Keys (CLK) record-level Bloom filter approach[26] with LSH blocking[46], hardened using the RAPPOR permanent randomized response method to provide differential privacy on the filters[47,29]. In this approach, given a probability value p, each bit of a Bloom filter is set to one with probability $\frac{p}{2}$, or set to zero with probability $\frac{p}{2}$, or kept unchanged. This provides ϵ-differential privacy with $\epsilon = 2h\ln(\frac{2}{p} - 1)$, where h is the number of hash functions used in the filter construction. We observe that using 1024-bit Bloom filters with 25 hash functions and $p = 0.02$, the method can provide a precision and recall of 98.7% and 98.9% respectively (threshold 0.9). However, a value of $p = 0.02$ implies a privacy budget of $\epsilon = 229.8$, which can be unacceptably large for privacy guarantees. The linkage accuracy degrades quickly as the budget is lowered; with $p = 0.2$ ($\epsilon = 109.9$), the precision and recall become 98.9% and 64.7% respectively (threshold 0.8).

7.5 | Linkage with blocking

We now revisit the task of linking significantly larger data sets, say on the order of millions, in an acceptable time frame. A blocking operation is often performed on larger data sets to reduce linkage time. Blocking on a field, or derivatives thereof, is the process of dividing a data set into smaller subsets such that all records within a subset are in agreement with respect to the blocking value. For example, we can divide a data set with a million records into subsets where the zip code value is equal for all records. As another example, we can also perform the grouping based on the initials of individual names. Linkage is then carried out for subsets from the two parties where the blocking value matches.

Blocking of our large data set (1,000,000 records) on zip code produces 860 subset pairs (one from each party) for linkage, each subset averaging around 1150 records, with the maximum size at 7200 records. Blocking using name initials produces 725

Table 4 Execution time to link two data sets with 1,000,000 records in each using blocking. Machine S3 is used. m: minutes.

	Number of vCPUs		
Blocking variable	8	4	1
Zip code	13.96 m	18.39 m	25.93 m
First and last initials	14.87 m	19.21 m	28.89 m

subset pairs to be linked, each subset averaging around 1370 records. We independently link each subset pair, and then combine the results to produce the final linkage result on the larger data set. Table 4 shows the total time required to link the data set in this manner. Since each individual linkage task is now on a much smaller data set, we can finish the linkage in a significantly smaller time frame. Further, since smaller data sets do not benefit much from more vCPUs, the linkage can be completed using fewer resources. If data sets with 10 million records are blocked into 1000 subsets of 10,000 records each, then extrapolating from the data in Table 2, the linkage can be completed in 3 hours in machine C1.

A side effect of blocking is the potential loss in precision/recall of the linkage algorithm, especially when the blocking variable is prone to high error rates. In our data preparation, we excluded the zip code from error insertion. As a result, precision and recall retain similar values when using zip code as the blocking variable, but deviates when using the name initials. With a threshold of 0.9, the no-blocking approach, as well as the blocking based linkage using zip code, has a precision and recall of 98.6% and 82.9% respectively, while blocking based linkage using the name initials have them at 98.7% and 81.1% respectively. Blocking on low error fields, or fields where values can be auto corrected (e.g. a county name), is therefore advised. We leave the treatment of linkage under multi-attribute blocking, each possibly with a different error likelihood, as a case for future study.

The introduction of blocking in the context of private record linkage requires the linkage agent to identify the subsets of data to be linked. If a shared secret key is present with the parties, both parties can mark the blocks with a keyed-hash transformation of the key values, which will be sufficient for the linkage agent to determine which blocks from one party are to be compared against which other blocks from the other party. In the absence of such shared secrets, the data encoder can perform the blocking operation, and separately provide as part of each subset an encoding for the bigrams that make up the blocking value. A linkage agent can then first perform a linking of data subsets based on the bigrams of the blocking value, before proceeding to perform record level linkage in each identified subset pair. This only adds the overhead of linking one more small dataset. Other private methods of performing the blocking can also be considered[48].

7.6 | Multiparty linkage

A multiparty record linkage requires finding common records that appear in the data sets of all the involved parties. A multiparty linkage can be performed using a sequence of two-party linkages. The first linkage is performed between parties 1 and 2. Each subsequent linkage is performed using the data records found common in the previous linkage and the data set of another party. In the end, the matched data records would be common across all parties and satisfy the similarity threshold requirement. We provide an estimate of the linkage time in such a linkage procedure in Table 5. Machine C1 is assumed in the estimations. A similar ring protocol utilizing counting Bloom filters is also reported to take ~100 seconds on a data set with 10,000 records at each party and 10 parties[49]. Other works around this form of multiparty linkage also exists, but can only work in the absence of data entry errors (exact matching)[20], or have exponential complexity[50]. Another variant of multiparty linkage aims to find common records in any two or more of the parties. Specialized methods to perform this form of multiparty linkage with cryprographic security guarantees do not exist yet. However, methods using counting Bloom filters and clustering techniques have recently appeared and solves this variant in quadratic time[51]. The multiparty linkage algorithms referenced here are all specifically designed to address one or the other variant, and are not simple iterative applications of a two-party algorithm. We leave the specialization of our optimizations for such variants for future work.

8 | CONCLUSION

The linkage times we achieve here are suggestive of the possibility that private record linkage can be offered as a service using available cloud computing resources. For scenarios where queries on distributed databases yield a few thousand records, and

Table 5 Estimated run time for multiparty linkage (extrapolated from the C1 data in Table 2). s: seconds, m: minutes, h: hours.

Number of parties	Number of records in a dataset		
	1000	10000	100000
2	0.32 s	10.84 s	14.49 m
3	0.64 s	21.68 s	28.98 m
5	1.28 s	43.36 s	57.96 m
10	2.88 s	1.62 m	2.17 h
20	6.08 s	3.43 m	4.59 h

requires private join operations for collated results, a near real-time linkage seems plausible. Larger data sets would require using techniques such as blocking, but can be reasonably handled.

This study was centered around improving linkage times and has resulted in some relevant source code. However, a complete tool chain is often desired by stakeholders with interest in such services, for example the biomedical research community. As such multiple components remains to be developed. The exposure risk we compute here is based on a static frequency analysis, i.e. frequencies are considered independently. A dynamic analysis can also be carried out by considering frequency distributions of bigrams in the presence of partial information (such as one or more known bigrams). Similar to how a frequency attack on a substitution cipher can make use of a dictionary, inference of bigrams from their encodings may also benefit from demographics dictionaries. Assessment of the exposure risks under such an adversarial model is an attractive future direction.

References

1. Elmagarmid AK, Ipeirotis PG, Verykios VS. Duplicate record detection: A survey. *IEEE Transactions on Knowledge and Data Engineering* 2007; 19(1): 1–16.

2. Brook E, Rosman D, Holman C. Public good through data linkage: Measuring research outputs from the Western Australian data linkage system. *Australian and New Zealand Journal of Public Health* 2008; 32(1): 19–23.

3. Phua C, Smith-Miles K, Lee V, Gayler R. Resilient identity crime detection. *IEEE Transactions on Knowledge and Data Engineering* 2012; 24(3): 533–546.

4. Kum HC, Krishnamurthy A, Machanavajjhala A, and SCA. Social genome: Putting big data to work for population informatics. *Computer* 2013; 47(1): 56–63.

5. Dusserre L, Quantin C, Bouzelat H. A one way public key cryptosystem for the linkage of nominal files in epidemiological studies. *MedInfo* 1995; 8 (Pt 1): 644–647.

6. Durham E, Xue Y, Kantarcioglu M, Malin B. Private medical record linkage with approximate matching. In: AMIA Annual Symposium Proceedings. ; 2010: 182–186.

7. Kuzu M, Kantarcioglu M, Durham E, Malin B. A constraint satisfaction cryptanalysis of Bloom filters in private record linkage. In: International Conference on Privacy Enhancing Technologies. ; 2011: 226–245.

8. Kroll M, Steinmetzer S. Who is 1011011111...1110110010? Automated cryptanalysis of bloom filter encryptions of databases with several personal identifiers. In: Proceedings of the International Joint Conference on Biomedical Engineering Systems and Technologies. ; 2015: 341–356.

9. Archer DW, Bogdanov D, Pinkas B, Pullonen P. Maturity and performance of programmable secure computation. *IEEE Security & Privacy* 2016; 14(5): 48–56.

10. Agrawal R, Evfimievski A, Srikant R. Information sharing across private databases. In: ACM SIGMOD International Conference on Management of Data. ; 2003: 86-97.

11. Dewri R, Ong T, Thurimella R. Linking health records for federated query processing. *Proceedings on Privacy Enhancing Technologies* 2016; 2016(3): 4–23.

12. Sayers A, Ben-Shlomo Y, Blom AW, Steele F. Probabilistic record linkage. *International Journal of Epidemiology* 2015; 45(3): 954–964.

13. Bouzelat H, Quantin C, Dusserre L. Extraction and anonymity protocol of medical file. In: AMIA Annual Fall Symposium. ; 1996: 323–327.

14. Grannis SJ, Overhage JM, McDonald C. Analysis of identifier performance using a deterministic linkage algorithm. In: AMIA Annual Symposium Proceedings. ; 2002: 305–309.

15. Churches T, Christen P. Some methods for blindfolded record linkage. *BMC Medical Informatics and Decision Making* 2004; 4: 9.

16. Churches T, Christen P. Blind data linkage using n-grams similarity comparisons. In: Advances in Knowledge Discovery and Data Mining. ; 2004: 121-126.

17. Doidge JC, Harron K. Demystifying probabilistic linkage: Common myths and misconceptions. *International Journal of Population Data Science (2018) 3:1 International Journal of Population Data Science* 2018; 3(1): 1–8.

18. al. v. E. eE. Evaluation of the encryption procedure and record linkage in the Belgian National Cancer Registry. *Archives of Public Health* 2000; 50(6): 281–294.

19. Karakasidis A, Verykios VS. Privacy preserving record linkage using phonetic codes. In: Balkan Conference in Informatics. ; 2009: 101-106.

20. Karapiperis D, Vatsalan D, Verykios VS, Christen P. Large-scale multi-party counting set intersection using a space efficient global synopsis. In: Proceedings of the International Conference on Database Systems for Advanced Applications. ; 2015: 329–345.

21. Bonomi L, Xiong L, Chen R, Fung B. Frequent grams based embedding for privacy preserving record linkage. In: Proceedings of the 21st ACM International Conference on Information and Knowledge Management. ; 2012: 1597–1601.

22. Schnell R, Bachteler T, Reiher J. Privacy-preserving record linkage using Bloom filters. *BMC Medical Informatics and Decision Making* 2009; 9: 41.

23. Kuzu M, Kantarcioglu M, Durham E, Toth C, Malin B. A practical approach to achieve private medical record linkage in light of public resources. *Journal of the American Medical Informatics Association* 2013; 20(2): 285–292.

24. Niedermeyer F, Steinmetzer S, Kroll M, Schnell R. Cryptanalysis of basic bloom filters used for privacy preserving record linkage. *Journal of Privacy and Confidentiality* 2014; 6(2): 59–79.

25. Mitchell W, Dewri R, Thurimella R, Rosckhe M. A graph traversal attack on bloom filter based medical data aggregation. *International Journal of Big Data Intelligence* 2017; 4(4): 217–226.

26. Schnell R, Bachteler T, Reiher J. A novel error-tolerant anonymous linking code. Tech. Rep. WP-GRLC-2011-02, German Record Linkage Center; 2011.

27. Durham EA, Kantarcioglu M, Xue Y, Toth C, Kuzu M, Malin B. Composite bloom filters for secure record linkage. *IEEE Transactions on Knowledge and Data Engineering* 2013; 26(12): 2956–2968.

28. Vatsalan D, Christen P, Verykios VS. An efficient two-party protocol for approximate matching in private record linkage. In: Proceedings of the 9th Australasian Data Mining Conference. ; 2011: 125–136.

29. Schnell R, Borgs C. Randomized response and balanced bloom filters for privacy preserving record linkage. In: Proceedings of the 16th International Conference on Data Mining Workshops. ; 2016: 218–224.

30. Christen P, Schnell R, Vatsalan D, Ranbaduge T. Efficient cryptanalysis of Bloom filters for privacy-preserving record linkage. In: Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining. ; 2017: 628–640.

31. Christen P, Ranbaduge T, Vatsalan D, Schnell R. Precise and fast cryptanalysis for Bloom filter based privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering* 2018; Preprints: 1–1.

32. Inan A, Kantarcioglu M, Bertino E, Scannapieco M. A hybird approach to private record linkage. In: International Conference in Data Engineering. ; 2008: 496-505.

33. Ioannou E, Nejdi W, Niederee C, Velegrakis Y. On-the-fly entity-aware query processing in the presence of linkage. *Proceedings of the VLDB Endowment* 2010; 3(1-2): 429–438.

34. Karapiperis D, Verykios VS. An LSH-based blocking approach with a homomorphic matching technique for privacy-preserving record linkage. *IEEE Transactions on Knowledge and Data Engineering* 2014; 27(4): 909–921.

35. Inan A, Kantarcioglu M, Ghinita G, Bertino E. Private record matching using differential privacy. In: International Conference on Extending Database Technology. ; 2010: 123-134.

36. He X, Machanavajjhala A, Flynn C, Srivastava D. Composing differential privacy and secure computation: A case study on scaling private record linkage. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ; 2017: 1389–1406.

37. Chen H, Laine K, Rindal P. Fast private set intersection from homomorphic encryption. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. ; 2017: 1243–1255.

38. Kamara S, Mohassel P, Raykova M, Sadeghian S. Scaling private set intersection to billion-element sets. Tech. Rep. MSR-TR-2013-63, Microsoft Research; 2013.

39. Pinkas B, Schneider T, Zoner M. Faster private set intersection based on OT extension. In: 23rd USENIX Conference on Security Symposium. ; 2014: 797–812.

40. Khurram B, Kerschbaum F. Sfour: A protocol for cryptographically secure record linkage at scale. In: Proceedings of the 36th International Conference on Data Engineering. ; 2020: 277–288.

41. Boneh D, Lynn B, Shacham H. Short signatures from the weil pairing. *Journal of Cryptology* 2004; 17(4): 213–229.

42. Icart T. How to hash into elliptic curves. In: Proceedings of the 29th Annual International Cryptology Conference on Advances in Cryptology. ; 2009: 303–316.

43. Certicom . SEC 2: Recommended elliptic curve domain parameters. tech. rep., Certicom Research; 2000.

44. Durstenfeld R. Algorithm 235: Random permutation. *Communications of the ACM* 1964; 7(7): 420.

45. Pinkas B, Schneider T, Zoner M. Scalable private set intersection based on OT extension. *ACM Transactions on Privacy and Security* 2018; 21(2): 1–35.

46. Franke M, Sehili Z, Rahm E. Parallel privacy-preserving record linkage using LSH-based blocking. In: Proceedings of the 3rd International Conference on Internet of Things, Big Data and Security. ; 2018: 195–203.

47. Erlingsson U, Pihur V, Korolova A. RAPPOR: Randomized aggregatable privacy-preserving ordinal response. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ; 2014: 1054–1067.

48. Steorts RC, Ventura SL, Sadinle M, Fienberg SE. A comparison of blocking methods for record linkage. In: Proceedings of the International Conference on Privacy in Statistical Databases. ; 2014: 253–268.

49. Vatsalan D, Christen P, Rahm E. Scalable privacy-preserving linking of multiple databases using counting Bloom filters. In: Proceedings of the 16th International Conference on Data Mining Workshops. ; 2016: 882–889.

50. Lai P, Yiu S, Chow K, Chong C, Hiu L. An efficient Bloom filter based solution for multiparty private matching. In: Proceedings of 2006 Security and Management. ; 2006.

51. Vatsalan D, Christen P, Rahm E. Incremental clustering techniques for multi-party privacy-preserving record linkage. *Data & Knowledge Engineering* 2020; 128: 101809.

APPENDIX

A TOY EXAMPLE

Consider the universe of characters to be $\{a, b\}$, which results in four potential bigrams – aa, ab, ba, and bb. The bigrams are indexed using the numbers $1, 2, 3$ and 4. Further, they are numerically represented as $10, 20, 30$ and 40 respectively. This simplified representation is for demonstration only; the bigrams would typically be represented as points in an elliptic curve. Let the key rings at sites A and B be $\mathcal{K}_A = \{k_1 = 3, k_2 = 5\}$ and $\mathcal{K}_B = \{k'_1 = 7, k'_2 = 11\}$ respectively. Keys can be referred to by their indices, e.g. key 1 in $\mathcal{K}_A$ is 3, key 2 in $\mathcal{K}_B$ is 11, and so on. We use the notation *index:value* to signify a value and the (multi-dimensional array) index where the value is stored. Table A.1 shows the permutations that A and B uses for the four bigrams. Given a bigram index and a key index, a site applies the permutation to decide the permuted index – the index where the OWF value is stored in the $L1$ files. We use the notation $\pi_{\text{site name}}$ for the forward permutation (bigram index to permuted index), and $\bar{\pi}_{\text{site name}}$ for the reverse permutation (permuted index to bigram index). The OWF value is computed here as the product of the key and a message; typically, this would be a point multiplication operation that is difficult to invert. Tables A.2 and A.3 show the precomputation steps performed by the two parties (refer to Section 3 for details). The generation of the linkage map is shown in Table A.4. Fig. A.1 shows the generation of the hash maps and a hypothetical comparison of two strings using the hash maps.

Table A.1 Permutations used by A and B.

	Bigram	Bigram index	Permuted index
A	aa	1	3
	ab	2	1
	ba	3	4
	bb	4	2

	Bigram	Bigram index	Permuted index
B	aa	1	2
	ab	2	3
	ba	3	1
	bb	4	4

Table A.2 $L1$ precomputation. OWF is computed as the product of the key and the numerical representation of a bigram. $L1$ values in the last column are shown as *index:value*.

(a) $L1_A$ precomputation.

(Key,Bigram) index	Key	Bigram	OWF	$L1_A$
(1,1)	3	10	30	[1,1]: 60
(1,2)	3	20	60	[1,2]: 120
(1,3)	3	30	90	[1,3]: 30
(1,4)	3	40	120	[1,4]: 90
(2,1)	5	10	50	[2,1]: 100
(2,2)	5	20	100	[2,2]: 200
(2,3)	5	30	150	[2,3]: 50
(2,4)	5	40	200	[2,4]: 150

(b) $L1_B$ precomputation

(Key,Bigram) index	Key	Bigram	OWF	$L1_B$
(1,1)	7	10	70	[1,1]: 210
(1,2)	7	20	140	[1,2]: 70
(1,3)	7	30	210	[1,3]: 140
(1,4)	7	40	280	[1,4]: 280
(2,1)	11	10	110	[2,1]: 330
(2,2)	11	20	220	[2,2]: 110
(2,3)	11	30	330	[2,3]: 220
(2,4)	11	40	440	[2,4]: 440

Table A.3 $L2$ precomputation and post-processing.

(a) $L2$ precomputation and post-processing at A. LK is key index of A, EK is key index of B, and PM is permuted bigram index created by B.

(LK,EK,PM) index	Local Key	$L1_B$	OWF	$L2_A$	Sorted $L2_A$ values (post-processing)	Indexing triples T_A (sent to linkage agent)
(1,1,1)	3	[1,1]: 210	630	[1,1,1]: 630	(1,1,2), 210	(1,1,2)
(1,1,2)	3	[1,2]: 70	210	[1,1,2]: 210	(1,2,2), 330	(1,2,2)
(1,1,3)	3	[1,3]: 140	420	[1,1,3]: 420	(2,1,2), 350	(2,1,2)
(1,1,4)	3	[1,4]: 280	840	[1,1,4]: 840	(1,1,3), 420	(1,1,3)
(1,2,1)	3	[2,1]: 330	990	[1,2,1]: 990	(2,2,2), 550	(2,2,2)
(1,2,2)	3	[2,2]: 110	330	[1,2,2]: 330	(1,1,1), 630	(1,1,1)
(1,2,3)	3	[2,3]: 220	660	[1,2,3]: 660	(1,2,3), 660	(1,2,3)
(1,2,4)	3	[2,4]: 440	1320	[1,2,4]: 1320	(2,1,3), 700	(2,1,3)
(2,1,1)	5	[1,1]: 210	1050	[2,1,1]: 1050	(1,1,4), 840	(1,1,4)
(2,1,2)	5	[1,2]: 70	350	[2,1,2]: 350	(1,2,1), 990	(1,2,1)
(2,1,3)	5	[1,3]: 140	700	[2,1,3]: 700	(2,1,1), 1050	(2,1,1)
(2,1,4)	5	[1,4]: 280	1400	[2,1,4]: 1400	(2,2,3), 1100	(2,2,3)
(2,2,1)	5	[2,1]: 330	1650	[2,2,1]: 1650	(1,2,4), 1320	(1,2,4)
(2,2,2)	5	[2,2]: 110	550	[2,2,2]: 550	(2,1,4), 1400	(2,1,4)
(2,2,3)	5	[2,3]: 220	1100	[2,2,3]: 1100	(2,2,1), 1650	(2,2,1)
(2,2,4)	5	[2,4]: 440	2200	[2,2,4]: 2200	(2,2,4), 2200	(2,2,4)

(b) $L2$ precomputation and post-processing at B. LK is key index of B, EK is key index of A, and PM is permuted bigram index created by A.

(LK,EK,PM) index	Local Key	$L1_A$	OWF	$L2_B$	Sorted $L2_B$ values (post-processing)	Indexing triples T_B (sent to linkage agent)
(1,1,1)	7	[1,1]: 60	420	[1,1,1]: 420	(1,1,3), 210	(1,1,3)
(1,1,2)	7	[1,2]: 120	840	[1,1,2]: 840	(2,1,3), 330	(2,1,3)
(1,1,3)	7	[1,3]: 30	210	[1,1,3]: 210	(1,2,3), 350	(1,2,3)
(1,1,4)	7	[1,4]: 90	630	[1,1,4]: 630	(1,1,1), 420	(1,1,1)
(1,2,1)	7	[2,1]: 100	700	[1,2,1]: 700	(2,2,3), 550	(2,2,3)
(1,2,2)	7	[2,2]: 200	1400	[1,2,2]: 1400	(1,1,4), 630	(1,1,4)
(1,2,3)	7	[2,3]: 50	350	[1,2,3]: 350	(2,1,1), 660	(2,1,1)
(1,2,4)	7	[2,4]: 150	1050	[1,2,4]: 1050	(1,2,1), 700	(1,2,1)
(2,1,1)	11	[1,1]: 60	660	[2,1,1]: 660	(1,1,2), 840	(1,1,2)
(2,1,2)	11	[1,2]: 120	1320	[2,1,2]: 1320	(2,1,4), 990	(2,1,4)
(2,1,3)	11	[1,3]: 30	330	[2,1,3]: 330	(1,2,4), 1050	(1,2,4)
(2,1,4)	11	[1,4]: 90	990	[2,1,4]: 990	(2,2,1), 1100	(2,2,1)
(2,2,1)	11	[2,1]: 100	1100	[2,2,1]: 1100	(2,1,2), 1320	(2,1,2)
(2,2,2)	11	[2,2]: 200	2200	[2,2,2]: 2200	(1,2,2), 1400	(1,2,2)
(2,2,3)	11	[2,3]: 50	550	[2,2,3]: 550	(2,2,4), 1650	(2,2,4)
(2,2,4)	11	[2,4]: 150	1650	[2,2,4]: 1650	(2,2,2), 2200	(2,2,2)

Table A.4 Generation of linkage map from the indexing triplets received from A and B. The last two columns are to demonstrate that the bigrams represented using the specific (key,permuted)-index values are indeed the same.

T_A: indexing triples (u, v, w) from A	T_B: indexing triples (v', u', w') from B	Linkage map $T[u, v, w] = w'$	Bigram equivalence: B's (v, w) = A's (u', w')	Reverse permutation (not computable by linkage agent)
(1,1,2)	(1,1,3)	[1,1,2]: 3	(1,2) = (1,3)	$\bar{\pi}_B(2) = \bar{\pi}_A(3) = 1$ (aa)
(1,2,2)	(2,1,3)	[1,2,2]: 3	(2,2) = (1,3)	$\bar{\pi}_B(2) = \bar{\pi}_A(3) = 1$ (aa)
(2,1,2)	(1,2,3)	[2,1,2]: 3	(1,2) = (2,3)	$\bar{\pi}_B(2) = \bar{\pi}_A(3) = 1$ (aa)
(1,1,3)	(1,1,1)	[1,1,3]: 1	(1,3) = (1,1)	$\bar{\pi}_B(3) = \bar{\pi}_A(1) = 2$ (ab)
(2,2,2)	(2,2,3)	[2,2,2]: 3	(2,2) = (2,3)	$\bar{\pi}_B(2) = \bar{\pi}_A(3) = 1$ (aa)
(1,1,1)	(1,1,4)	[1,1,1]: 4	(1,1) = (1,4)	$\bar{\pi}_B(1) = \bar{\pi}_A(4) = 3$ (ba)
(1,2,3)	(2,1,1)	[1,2,3]: 1	(2,3) = (1,1)	$\bar{\pi}_B(3) = \bar{\pi}_A(1) = 2$ (ab)
(2,1,3)	(1,2,1)	[2,1,3]: 1	(1,3) = (2,1)	$\bar{\pi}_B(3) = \bar{\pi}_A(1) = 2$ (ab)
(1,1,4)	(1,1,2)	[1,1,4]: 2	(1,4) = (1,2)	$\bar{\pi}_B(4) = \bar{\pi}_A(2) = 4$ (bb)
(1,2,1)	(2,1,4)	[1,2,1]: 4	(2,1) = (1,4)	$\bar{\pi}^2_B(2) = \bar{\pi}^1_A(4) = 3$ (ba)
(2,1,1)	(1,2,4)	[2,1,1]: 4	(1,1) = (2,4)	$\bar{\pi}_B(1) = \bar{\pi}_A(4) = 3$ (ba)
(2,2,3)	(2,2,1)	[2,2,3]: 1	(2,3) = (2,1)	$\bar{\pi}^2_B(3) = \bar{\pi}^2_A(4) = 2$ (ab)
(1,2,4)	(2,1,2)	[1,2,4]: 2	(2,4) = (1,2)	$\bar{\pi}_B(4) = \bar{\pi}_A(2) = 4$ (bb)
(2,1,4)	(1,2,2)	[2,1,4]: 2	(1,4) = (2,2)	$\bar{\pi}_B(4) = \bar{\pi}_A(2) = 4$ (bb)
(2,2,1)	(2,2,4)	[2,2,1]: 4	(2,1) = (2,4)	$\bar{\pi}_B(1) = \bar{\pi}_A(4) = 3$ (ba)
(2,2,4)	(2,2,2)	[2,2,4]: 2	(2,4) = (2,2)	$\bar{\pi}_B(4) = \bar{\pi}_A(2) = 4$ (bb)

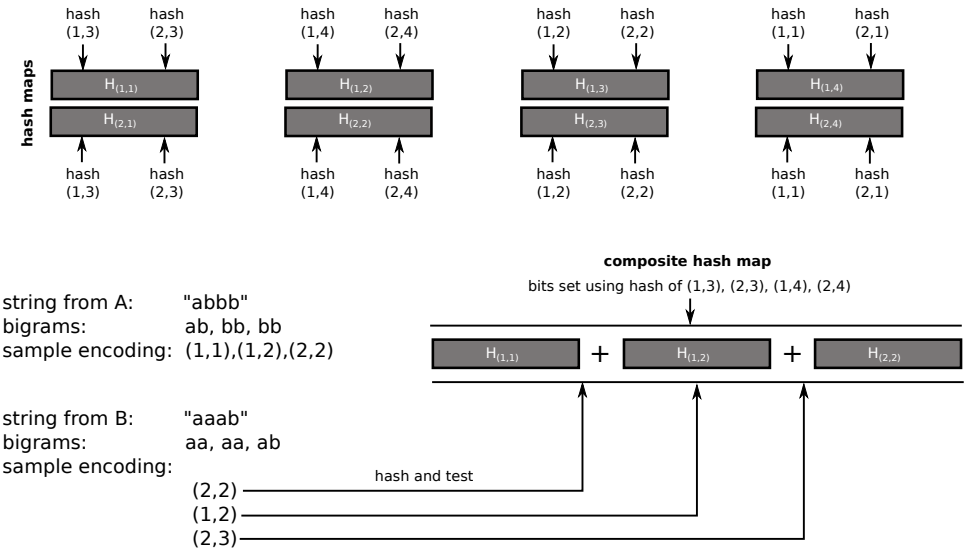


Figure A.1 Hash map for each possible bigram encoding of A (using 2 keys and 4 bigrams). Composite hash map is created based on bigrams in a string from A, and used to check for membership of B's bigrams. + implies bitwise OR.