Approximate Two-Party Privacy-Preserving String Matching with Linear Complexity

Martin Beck
Technische Universität Dresden
Institute of Systems Architecture
Dresden, Germany
Email: martin.beck1@tu-dresden.de

Florian Kerschbaum
SAP Research
Karlsruhe, Germany
Email: florian.kerschbaum@sap.com

***Abstract*—Consider two parties who want to compare their strings, e.g., genomes, but do not want to reveal them to each other. We present a system for privacy-preserving matching of strings, which differs from existing systems by providing a deterministic approximation instead of an exact distance. It is efficient (linear complexity), non-interactive and does not involve a third party which makes it particularly suitable for cloud computing. We extend our protocol, such that it only reveals whether there is a match and not the exact distance. Further an implementation of the system is evaluated and compared against current privacy-preserving string matching algorithms.**

***Keywords*-privacy-preserving string comparison, approximate string matching, homomorphic encryption, variable length grams, linear complexity**

I. Introduction

Storing and querying big data poses significant privacy risks. We present an approach to privately querying genomic sequences – a major source of big data. Particularly, we optimize performance, such that our approach scales to volumes frequently found in big data.

As technology for sequencing the human genome is developing at a fast pace and the number of sequenced genomes is rapidly growing, the need to process this highly personalized information in a privacy preserving way also increases. Several studies demonstrate how genomes can be linked to surnames [6] or even reveal the full identity [9, 19] of the individual. Many algorithms were presented which should protect the genomic information while it is being processed across untrusted parties.

These protocols however are either interactive, match only exact strings or require a third party to be involved. Our protocol is non-interactive, implements approximate string matching and does not require any third party. Our protocol is efficient and has linear complexity in computation and communication. It also has better resistance to an iterated differential attack proposed by Goodrich [7], that exploits the information gained by knowing the exact string distance (as proposed in other protocols), since it only reveals whether there is a match.

Our contributions:

- A new efficient privacy-preserving, non-interactive, two-party string matching protocol
- An analysis of our scheme in a genome matching setting using full mitochondrial DNA sequences
- We can privacy-preservingly, approximately compare real-world genomes in under 5 minutes on commodity hardware.

The remainder of this paper is structured as follows. Section II gives an overview over basic concepts and related work. In Section III the design will be presented and followed by Section IV, which gives a security analysis of our system. Section V describes some implementation details and results in comparison to related systems. Section VI concludes this work and points out further research directions.

II. Related Work

Research into string matching algorithms is defined by a long list of proposed algorithms over many years and for many different problems. String matching itself is closely related to the distance between strings, which can be measured by a large variety of means, ranging from generic and simple solutions like the Hamming distance [11] to more powerful algorithms like Smith-Waterman [24] solving local sequence alignment problems. A survey about current developments can be found in [18].

A. Approximate String Matching

As several tasks, for example checking whether a user profile is within a remote database, do not require the exact distance between two strings, data items or other entities, the notion of approximate matching was introduced to define levels of similarity, which in the most extreme way only output a single bit of information: if the input strings are similar or not. Due to these properties this class is called approximate string matching algorithms, which is not to be confused with the approximate string matching of [10], where the term “approximate” referred to the property of two strings being close in distance.

B. Privacy-Preserving String Matching

Two of the applications for string comparison algorithms which are often used for motivation are calculating the distance of genome or protein sequences in bioinformatics and checking if a person is present in a remote database. As these topics by design deal with very personal information, which must not be given to third parties, the necessity to build privacy-preserving matching algorithms arose. As these were not sufficient to protect privacy due to information leakage given by the exact distance results, just obtained in a privacy preserving manner, combinations of the above mentioned approximation and the privacy-preserving computational steps were developed. A survey of recently published algorithms together with benchmark results can be found in [2].

One of the more recent protocols introduced by Schnell et al. [23] uses Bloom filters to represent strings and transforms the notion of distances between strings into distances between similar Bloom filters. We will also use Bloom filters as set representation for our genomic strings and build the matching protocol upon them. However, we use a two-party technique for comparing the Bloom filters and therefore do not need a trusted third party for comparing the strings. Furthermore, our protocol can be size-hiding, by choosing appropriate Bloom filter sizes, that are not proportional to the string length.

Alternatively, techniques from private set intersection (PSI) [13] could be used. However, revealing the content of the intersection is not appropriate for a privacy preserving protocol. Based on these security concerns, protocols for private set intersection cardinality (PSI-CA) were developed [5]. Yet, these solutions still reveal the intersection cardinality, whereas we only reveal whether there is a match.

Privacy-preserving protocols designed for approximate string comparisons can also be found in literature [25, 15], but rely on interactive techniques like oblivious transfers or secure computation. This excludes these protocols from off-line execution, e.g., in the cloud. Further [3] presents a more efficient solution, but which only matches exact strings, whereas we compare approximate strings.

III. Protocol Design

Let the client (Alice) have a string, e.g. a genome, and the server have a string. After the execution of our protocol Alice will have learned whether the two strings are approximately close, but not Bob's string nor the approximate distance to Bob's string. Bob will learn nothing.

First the transformation into grams of variable length and their representation through a Bloom filter is specified, upon which the generic string matching algorithm is given. Following this generic matching algorithm a privacy-preserving version is constructed and then enhanced to only reveal whether there is a match.

A. Bloom Filter Representation

A Bloom filter is a data structure fixed in size to which element representations can be added and on which member tests can be performed. Checking for an element is probabilistic due to the design of the filter.

Let b be an array of bits of length n and $b[i]$ the ith value within the array with $i \in [1, n]$. Further let $h_1() \dots h_k()$ be k hash functions, with uniformly distributed output in $[1, n]$. For initialization set $\forall i \in [1, n] : b[i] = 0$.

To add an element e to the filter, all k hash functions are evaluated on e and the results are taken as indexes for b to set these positions to one. Set $\forall j \in [1, k] : b[h_j(e)] = 1$.

A member test for element e' is performed by also evaluating all k hash functions and checking the referenced positions in b. If at least one of the positions $b[h_j(e')]$ is set to zero, the element has not been added to the Bloom filter before. If all bits are set to one, however, one cannot be sure if the exact element was inserted, or one or more different elements had these positions set to one.

Using these operations a set is represented by adding all set elements to the filter. Depending on the filter parameters, the probability that a false-positive member test occurs, i.e. that an element is falsely identified as being added to the filter before, is given by:

$$p = \left(1 - \left(1 - \frac{1}{n}\right)^{kl}\right)^k \tag{1}$$

Where $\left(1 - \frac{1}{n}\right)^{kl}$ is the probability that a single bit is still zero after l elements were added to the filter of length n using k hash functions. To calculate the required length of a Bloom filter n given the false-positive rate and the number of elements to be inserted l, the equation (1) can be transposed to:

$$n = \frac{-1}{\left(1 - p^{1/k}\right)^{1/(k*l)} - 1} \tag{2}$$

B. String Matching Using Bloom Filters

A typical string comparison algorithm is the Levenshtein distance [16], which is often also referred to as edit distance and describes the minimum number of insertions, deletions and substitutions needed to transform one string S_1 into another S_2. The result is a distance measure d, which can easily be converted into a similarity score s between zero and one by: $s = 1 - \frac{d}{d_{max}}$

d_{max}, i.e. the maximum distance between two strings, equals the length of the longer string and can thus be replaced by $d_{max} = \max(|S_1|, |S_2|)$ regarding the Levenshtein distance.

$$s = 1 - \frac{d}{\max(|S_1|, |S_2|)}$$

As a Bloom filter is a set representation, the input strings first need to be converted into sets. This has to be done in a way, that a distance measure can later be formulated upon the constructed set which, loosely spoken, correlates with the Levenshtein distance measure.

The sets are build from q-grams, which are substrings of length q from input string S. Let $n = |S|$ be the number of characters in S and s_i the q-gram starting at position i with

$i \in [1, n-q+1]$. As a result $n-q+1$ q-grams are generated out of S using a sliding window for all possible i. If this set would be used to represent a string and measure similarity upon, the positional information of the substrings would not be included, which is important to build the similarity measure. To keep this information positional q-grams are used, which are pairs (i, s_i) with i being the position in S and s_i the actual q-gram starting at that position.

Further as characters at the beginning and at the end of S are underrepresented over all q-grams, the input string S is extended by $q-1$ identical symbols, which are not part of the alphabet of S at the beginning and end of S. Gravano et al. [8] introduced this definition of positional q-grams on extended strings. Without the extension we would only see the first character in the first q-gram, whereas in the middle of the string each character is found within q q-grams.

These positional grams are not used directly, but a technique called VGRAM [17] is employed to generate grams with variable lengths within a previously defined range $[q_{min}, q_{max}]$. To choose which length to select at what position, a gram dictionary is build prior to running the string comparison algorithm. As source to build this dictionary, the Human Mitochondrial Genome Database [14] is used. Afterwards the generated dictionary is published and available to all participants described in this string matching algorithm.

Both parties use the VGRAM algorithm to build a set of variable length grams based on the published dictionary, following the description in [17]. The number of variable length grams generated for a string S is depicted by n_v.

As these sets cannot be used directly to compare strings in a privacy-preserving way, which would reveal the original data, we represent them using Bloom filters. A single Bloom filter is used for all grams generated by a single string S. Papapetrou et al. [22] conclude, that the optimal number of hash functions to do cardinality estimation using Bloom filters is 1. As our string similarity measure will mainly use the estimated set cardinality for the Bloom filters, we thus fix $k = 1$ and only use a single hash function to build and query Bloom filters throughout the rest of the paper. The length l of the Bloom filter and the used hash function $h()$ is also fixed and set to be equal across all participants.

Determining an appropriate value for l can be done using the formula (2) with the simplification $k = 1$ as introduced above. This results in l being calculated as:

$$l = \frac{-1}{(1-p)^{n_v^{-1}} - 1}$$

Under these constraints, that k, l and $h()$ are identical, set union $\cup$ and intersection $\cap$ can also be performed upon Bloom filters B_1, B_2 by applying the binary OR or AND operator.

Li et al. [17] describe the effect of a single edit operation on the set of variable length grams and propose an algorithm to calculate the maximum number of affected grams by applying a number of edit operations upon an initial string. Based on this number a lower bound on the number of common grams for two strings which are within a certain edit distance can be calculated.

As this lower bound calculation on the set intersection cardinality uses the input sequences to generate a baseline for the lower bound, it cannot be applied in our privacy-preserving scheme directly. To be independent of such a baseline, we do not use the absolute set intersection cardinality directly, but the difference between the union cardinality and the intersection cardinality. This results in an approximate distance measure, which follows the explanation for the upper bound on the Hamming distance between bit vectors in [17]. Our distance measure equals the Hamming distance between the Bloom filter bit vectors.

$$d = |B_1 \cup B_2| - |B_1 \cap B_2|$$

Thus having $d = 0$ equals to having identical strings, as the set union and intersection sizes are identical.

C. Encrypting the Bloom Filter

We constructed string representations using Bloom filters in Section III-B and used them to define an appropriate distance measure. However the Bloom filters themselves cannot be exchanged between the participants directly, as they can be used to possibly reconstruct the original strings by guessing substrings and checking if they were added to the filter. For preserving privacy of the filter content, an additively homomorphic cryptosystem is used.

A homomorphic cryptosystem uses at least one homomorphic property to evaluate an operation $\oplus$ on the ciphertext, which translates into applying the equivalent operation $+$ on the plaintext. We will use the additively homomorphic system introduced by Naccache and Stern [20], which is also probabilistic. Alice generates a key pair and shares the public key with Bob. Let $E(x, r)$ denote the encryption of a value x using a fresh random value r for each encryption, this additively homomorphic system has the following properties:

$$\begin{aligned} E(x,r) \cdot E(y,s) &= E(x+y, rs) \\ E(x,r)^y &= E(xy, r^y) \end{aligned}$$

Further let $E(x, r)^{-1}$ denote the calculation of the multiplicative inverse upon $E(x, r)$, found through executing the extended euclidean algorithm, which is by the homomorphism definition the encryption of the additively inverse plaintext. This results in $E(x, r)^{-1} = E(-x, r)$ and can be used to calculate a difference between two encrypted values.

To multiply an encrypted plaintext with a negative factor $-z$, first the multiplicative inverse of the encrypted value is calculated and then multiplied using the positive factor.

$$E(x,r)^{-z} = E(x,r)^{-1 \cdot z} = E(-x,r)^z = E(-xz, r^z)$$

To increase readability, $E(x) = E(x, r)$ is used, which also always uses a fresh r.

An encryption of a Bloom filter B with length l is constructed by encrypting every bit in B separately, storing the resulting l values in a new array C with equal length.

$$\forall i \in [1,l], \text{ fresh } r: \quad C[i] = E(B[i], r)$$

This encryption is not to be confused with "encrypted Bloom filter", which were presented in [4]. Encryption of the Bloom filter is only performed by Alice, who wants to compare a string privately to one that Bob holds. Bob also slices his string down into variable length grams, which are then added to a new Bloom filter using the previously agreed upon parameters k, l and $h()$.

Alice sends the encryption of her Bloom filter to Bob, together with her public key. Recall that the Bloom filters just contained zeros and ones. So calculating the cardinality of a filter, denoted by $|B|$, cannot just be done by counting all bits set to one, but also by calculating the sum over all values $|B| = \sum_{i=1}^{l} B[i]$.

Bob can use this property to calculate the encrypted sum over all values in the encrypted Bloom filter and thus the encrypted cardinality.

$$E(|B|, r) = E(\sum_{i=1}^{l} B[i], r) = \prod_{i=1}^{l} C[i]$$

However as Bob is not interested in the encrypted cardinality of Alice's filter $|B_A|$, he only adds up values at those positions, that are set to one on his own Bloom filter B_B. This is equivalent to building the intersection using binary AND and calculating the resulting cardinality.

$$E(|B_A \cap B_B|) = \prod_{i, B_B[i]=1} C[i]$$

Further Alice encrypts the cardinality of her Bloom filter $E(|B_A|)$ and sends it to Bob, who also encrypts the cardinality of his own Bloom filter $E(|B_B|)$. Using these values, the union cardinality is calculated as follows:

$$\begin{aligned} |B_A \cup B_B| &= |B_A \cap B_B| + (|B_A| - |B_A \cap B_B|) \\ &\quad + (|B_B| - |B_A \cap B_B|) \\ &= |B_A| + |B_B| - |B_A \cap B_B| \end{aligned}$$

$$E(|B_A \cup B_B|) = E(|B_A|) \cdot E(|B_B|) \cdot E(|B_A \cap B_B|)^{-1}$$

This way the encrypted distance $E(d)$ for the measure presented in Section III-B is calculated as such:

$$\begin{aligned} E(d) &= E(|B_A \cup B_B|) \cdot E(|B_A \cap B_B|)^{-1} \\ &= E(|B_A|) \cdot E(|B_B|) \cdot E(|B_A \cap B_B|)^{-2} \end{aligned}$$

D. Privacy-Preserving Similarity

The calculated approximate distance value $E(d)$ between both compared strings S_A and S_B in Section III-C, could be returned to Alice for decryption, for her to learn the actual computed value. This would however result in increased sensitivity to the Mastermind attack described in [7]. To circumvent this attack, we restrict the information Alice gains from executing this protocol. Instead of learning the exact result of the comparison, the result is manipulated to give Alice only the information whether the distance is smaller than a previously defined threshold t_{max}.

Recall that the calculated distance value equals the Hamming distance between the Bloom filters and that [17] describes how to calculate an upper bound for the Hamming distance in equation (4). The calculation however involves the number of affected grams for both input strings S_A and S_B. As S_B is not available to Alice, she uses the revised Cambridge Reference Sequence (rCRS) [1] S_{rCRS} as a reference to replace S_B in the calculation of the upper Hamming distance bound. This replacement is a good approximation for small edit distances. Following [17] the upper bound for a maximum edit distance e_{max} is calculated as $t_{max} = NAG(S_A, e_{max}) + NAG(S_{rCRS}, e_{max})$, where $NAG(S, e)$ describes the maximum number of affected grams for e edit operations on string S. Further, as $NAG(S_{rCRS}, e_{max})$ is very close to $NAG(S_B, e_{max})$ and thus used as an replacement. $NAG(S_A, e_{max})$ can also be replaced for the same reason. This has the effect, that the chosen t_{max} does not depend on the input sequence S_A.

Due to the probabilistic nature of the Bloom filter, elements are mapped to the same positions with a probability p as described in Section III-A. As the Bloom filter cardinality $|B_A|$ is therefore on average smaller than the number of variable grams for S_A by a factor p, the upper bound is corrected to an approximated upper bound.

$$t_{max} \quad = \quad 2 \cdot NAG(S_{rCRS}, e_{max})) \cdot (1-p)$$

The protocol for calculating the return values for Alice by Bob is as follows:

- Calculate encrypted inverse thresholds $\forall t_i \in [0, t_{max}]$: $E(-t_i) = E(t_i)^{-1}$
- Calculate encrypted threshold differences for all inverse thresholds
  $E(D_i) = E(d - t_i) = E(d) \cdot E(-t_i)$
- Multiply all differences with random values. $E(rD_i) = E(D_i)^r$ for fresh r drawn uniformly from the plaintext space of the used cryptographic system.

After the first two steps Bob has $t_{max}+1$ values, expressing the differences between the incremented thresholds and the actual distance. If the calculated distance d is within the defined threshold range $[0, t_{max}]$, then there is one single element, which is the encryption of zero due to equal threshold and distance values.

Performing the last step randomizes all values through multiplication with a random number, except the one encrypting a zero. All these $t_{max} + 1$ encrypted values are then shuffled randomly and sent to Alice, who decrypts and checks them against zero. In case a zero is found, she learns that the Bloom filter intersection cardinality was within the specified threshold and thus the compared strings have an edit distance equal or less than the specified maximum edit distance e_{max} used to calculate t_{max} in Section III-B.

IV. Security Analysis

Our protocol is secure under the semi-honest, also called honest-but-curious model and under the assumption the integrated crypto system builds upon. In our case this is based on the higher residuosity problem used in the Naccache-Stern cryptosystem. Several other additive homomorphic cryptosystems like Paillier [21] can easily be used instead of the currently employed system, bringing possibly another assumption like one based on the decisional composite residuosity problem as basis.

The encryption of the used cryptosystem must however be probabilistic, such that similar plaintexts are mapped to different ciphertexts at random. This is true for our employed Naccache-Stern system and the above mentioned Paillier cryptosystem. This property is also called semantic security and corresponds to indistinguishability under chosen plaintext attack (IND-CPA).

In the first part of our protocol, Alice translates her input string into variable length grams, generates a Bloom filter representation and encrypts it using a public key cryptosystem. As she is not using any information from Bob, she cannot gain any insight into Bob's input.

The second part involves Bob working on the encrypted Bloom filter from Alice and her encrypted Bloom filter cardinality. As all values are encrypted using an asymmetric, probabilistic cryptosystem, for which only Alice has the private decryption key, Bob cannot decide if an encrypted value represents a zero, a one or any other value, which directly follows the security analysis of the underlying hardness assumption. The number of elements received does not depend on Alice input, as only public information is used to infer the Bloom filter length, as introduced in Section III-D. Further Bob sums up elements from Alice's encrypted Bloom filter, based on his Bloom filter. The result is then subtracted several times from different threshold values and multiplied with random numbers, chosen uniformly from within the domain of plaintexts of the underlying cryptosystem. All results are shuffled at random and transmitted back to Alice. Bob gained no information in this phase about Alice's input.

As a last step Alice decrypts all results received from Bob and checks if they contain a zero. If a zero is found, she learns that the Hamming distance between the Bloom filters was below a predefined threshold t_{max}. There can be at most one zero. If no zero was found, the threshold was lower than the Hamming distance. From the decrypted non-zero results, she cannot learn anything, as these numbers are uniformly distributed due to the multiplication with uniform random numbers drawn from the plaintext domain modulo the the plaintext domain modulus. The index of the zero element, if there was one, gives no information to Alice, as the return values were randomly shuffled by Bob. The number of returned elements also holds no further information, as there are always $t_{max} + 1$ results.

The only information Alice learns about the input of Bob is, if the threshold was above the Bloom filter Hamming distance or not.

V. Evaluation

For the experiments a Linux Laptop with an Intel Core2 Duo T9600 running at 2.8 GHz was used. The code is written in Java, using the Bouncy Castle library[1]. The first tests evaluate the relation between the Levenshtein distance and the measure introduced in section III-B. Further the runtime performance of the algorithm is evaluated for string lengths, which were also used for comparing other privacy-preserving string matching protocols. All code implementing the techniques in this paper and producing the test results can be found under http://dud.inf.tu-dresden.de/~beck/bloomEncryption.tar.bz2.

A. Distance Measure

As the similarity metric is based on the Levenshtein distance as described in [17], we measure the relation between the edit distance and the Bloom filter Hamming distance as our proposed metric. The parameters $q_{min} = 2$ and $q_{max} = 40$ are used as [17] states that the variable length gram algorithm can start with a low q_{min} and a large q_{max} to find appropriate values for these parameters after pruning the built Trie.

To run the tests the Bloom filters are set up to use a single hash function, in our case "SHA1" modulo the size of the filter. The used strings contain roughly $n = 16,569$ characters, which means that about the same number of variable grams have to be inserted into each Bloom filter.

The probability that a false-positive test occurs after the n elements are added to the filter is set to $p = 0.1$, which in turn generates a Bloom filter of size 157261 bits. We used 10000 runs and for each 100 applied a fixed number of edit operations. The original string and the altered string are then compared using our distance measure. The resulting value is the difference between the union cardinality and the intersection cardinality of both Bloom filters B_1 and B_2. This represents the total number of unique elements for both parties, or the Hamming distance between both Bloom filters. Following Lemma 1 in [17] this directly correlates with the Levenshtein distance between the strings.

Figure 1 shows a Boxplot for every Levenshtein distance and the according 100 runs tested with our approximate distance measure. As can be seen from the figure, our distance value approximates small Levenshtein distances very good, with a narrow range of possible values and a small variance. The Pearson correlation between the Levenshtein distances and the approximated distances is $c_p = 0.997$ for up to 100 edit operations.

B. Protocol Execution Time

To evaluate the performance of our protocol, we ran 100 runs for each test. The parameters were set to $q_{min} = 2$, $q_{max} = 40$, $p = 0.1$ and an edit distances of up to 10 operations.

The client runtime depends linearly on the length of the input sequence, where the most time is spent on decrypting

[1] http://www.bouncycastle.org/

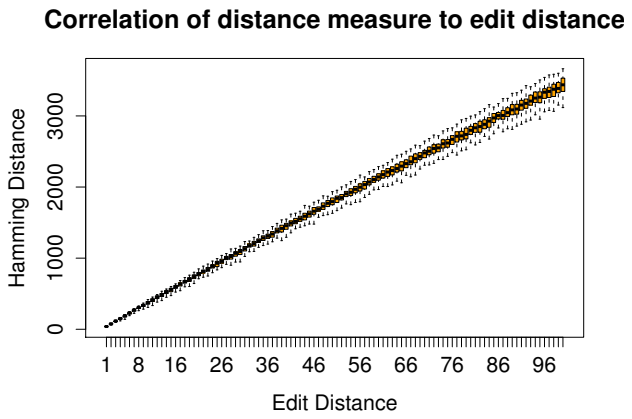

Fig. 1. Approximation of the Levenshtein distance by our distance measure

the results from the server and encrypting the Bloom filter prior to transmission. We can see a pretty high variance on client runtimes, growing linearly with longer sequences, due to the unknown number of results which are needed to be decrypted until a zero is found. If the distance between both compared sequences is not within the predefined range given by the threshold, the client always needs to decrypt all results, as no zero will be found within the returned values.

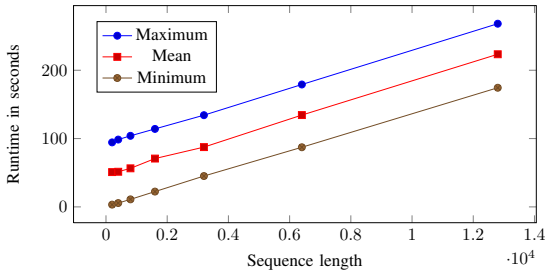

Fig. 2. Client runtimes

Server runtime depends linearly on the threshold value, whereas runtimes for different sequence lengths are only increasing slightly. The measured values for a constant threshold derived from a maximum edit distance of 10 and a variable sequence length range between 8.54 seconds for sequences of length 200 and 9.01 seconds for full mitochondrial DNA sequences.

The amount of data that needs to be transferred between Alice and Bob is shown in table I and grows linearly with the length of the Bloom filter for the traffic from Alice to Bob and linearly with the size of the threshold range for the traffic from Bob to Alice. For this test the threshold t_{max} is set to the maximum Hamming distance defined in Section III-D for a maximum edit distance of 10.

Comparing these results to the ones given by Jha et al. [15] and Huang et al. [12] in the evaluations of their state of the art protocols, we achieve an increased performance starting with the smallest sequence lengths of 200 characters. Due to

Sequence length	Client to server	Server to client
200	296 KB	123 KB
400	590 KB	123 KB
800	1169 KB	123 KB
1600	2337 KB	123 KB
3200	4663 KB	123 KB
6400	9323 KB	123 KB
12800	18636 KB	123 KB

TABLE I
BANDWIDTH USED FOR TRANSMISSION

the lower linear complexity of our protocol, comparisons of full mitochondrial DNA sequences can be performed more efficiently. The referenced protocols have computational complexity of $O(n \log n)$, $O(n^2)$ and $O(n * m)$ for input string lengths n and m.

VI. Conclusion

We presented a novel, non-interactive approach for a privacy-preserving approximate string matching protocol, that achieves superior performance for real-world sized genomes. An attacker will not even learn the exact distances or approximations, but only whether two compared strings are within a predefined distance range.

Due to the computation having linear complexity in the used sequence length and the communication having linear complexity in the range of allowed distances, respectively in the Bloom filter length, this protocol is very practical and was tested for full mitochondrial sequences with 16500 characters in length and a maximum edit distance of 10, which took about 286 seconds on the mentioned hardware to complete.

Further enhancements could go into using our protocol for database searches.

References

[1] S. Anderson, A. T. Bankier, B. G. Barrell, M. H. L. de Bruijn, A. R. Coulson, J. Drouin, I. C. Eperon, D. P. Nierlich, B. A. Roe, F. Sanger, P. H. Schreier, A. J. H. Smith, R. Staden, and I. G. Young. Sequence and organization of the human mitochondrial genome. *Nature*, 290(5806):457–465, April 1981. ISSN 0028-0836.

[2] Tobias Bachteler and Rainer Schnell. An empirical comparison of approaches to approximate string matching in private record linkage. *Proceedings of Statistics Canada*, 2010.

[3] Pierre Baldi, Roberta Baronio, Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Countering GATTACA: efficient and secure testing of fully-sequenced human genomes. In *Proceedings of the 18th ACM conference on Computer and communications security*, CCS '11, pages 691–702, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0948-6.

[4] Steven M. Bellovin, Steven M Bellovin, and William R Cheswick. Privacy-Enhanced Searches Using Encrypted Bloom Filters, 2004.

[5] Emiliano De Cristofaro, Paolo Gasti, and Gene Tsudik. Fast and Private Computation of Cardinality of Set Intersection and Union. *Cryptology ePrint Archive, Report 2011/141*, pages 1–19, 2011.

[6] Jane Gitschier. Inferential genotyping of Y chromosomes in Latter-Day Saints founders and comparison to Utah samples in the HapMap project. *American journal of human genetics*, 84(2):251–8, February 2009. ISSN 1537-6605.

[7] Michael T. Goodrich. The Mastermind Attack on Genomic Data. In *2009 30th IEEE Symposium on Security and Privacy*, pages 204–218. IEEE, May 2009. ISBN 978-0-7695-3633-0.

[8] Luis Gravano, Panagiotis G. Ipeirotis, Hosagrahar Visvesvaraya Jagadish, Nick Koudas, Shanmugauelayut Muthukrishnan, and Divesh Srivastava. Approximate String Joins in a Database (Almost) for Free. *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 491–500, September 2001.

[9] M. Gymrek, A. L. McGuire, D. Golan, E. Halperin, and Y. Erlich. Identifying Personal Genomes by Surname Inference. *Science*, 339(6117):321–324, January 2013. ISSN 0036-8075.

[10] Patrick A. V. Hall and Geoff R. Dowling. Approximate String Matching. *ACM Computing Surveys*, 12(4):381–402, December 1980. ISSN 03600300.

[11] Richard Wesley Hamming. Error-Detecting and Error-Correcting Codes. *Bell System Technical Journal*, 29: 147–160, 1950.

[12] Yan Huang, David Evans, and Jonathan Katz. Faster secure two-party computation using garbled circuits. *USENIX Security Symposium*, 2011.

[13] Yan Huang, David Evans, and Jonathan Katz. Private Set Intersection: Are Garbled Circuits Better than Custom Protocols? *NDSS*, 2012.

[14] Max Ingman and U Gyllensten. mtDB: Human Mitochondrial Genome Database, a resource for population genetics and medical sciences. *Nucleic Acids Research*, 34:749–751, 2006.

[15] Somesh Jha, Louis Kruger, and Vitaly Shmatikov. Towards Practical Privacy for Genomic Computation. In *2008 IEEE Symposium on Security and Privacy (sp 2008)*, pages 216–230. IEEE, May 2008. ISBN 978-0-7695-3168-7.

[16] Vladimir Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[17] Chen Li, Bin Wang, and Xiaochun Yang. VGRAM: improving performance of approximate queries on string collections using variable-length grams. In *Proceedings of the 33rd international conference on Very large data bases*, VLDB'07, pages 303–314, September 2007. ISBN 978-1-59593-649-3.

[18] Heng Li and Nils Homer. A survey of sequence alignment algorithms for next-generation sequencing. *Briefings in bioinformatics*, 11(5):473–83, September 2010. ISSN 1477-4054.

[19] Jeantine E Lunshof, Ruth Chadwick, Daniel B Vorhaus, and George M Church. From genetic privacy to open consent. *Nature reviews. Genetics*, 9(5):406–11, May 2008. ISSN 1471-0064.

[20] David Naccache and Jacques Stern. A new cryptosystem based on higher residues. *Proceedings of the 5th ACM conference on on computer and communication security*, pages 59–66, 1998.

[21] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. *Advances in Cryptography - Eurocrypt '99*, 1592:223–238, 1999.

[22] Odysseas Papapetrou, Wolf Siberski, and Wolfgang Nejdl. Cardinality estimation and dynamic length adaptation for Bloom filters. *Distributed and Parallel Databases*, 28 (2-3):119–156, September 2010. ISSN 0926-8782.

[23] Rainer Schnell, Tobias Bachteler, and Jörg Reiher. Privacy-preserving record linkage using Bloom filters. *BMC medical informatics and decision making*, 9(1):41, January 2009. ISSN 1472-6947.

[24] Temple F. Smith and Michael S. Waterman. Identification of common molecular subsequences. *Journal of molecular biology*, 147(1):195–7, March 1981. ISSN 0022-2836.

[25] Juan Ramón Troncoso-Pastoriza, Stefan Katzenbeisser, and Mehmet Celik. Privacy preserving error resilient dna searching through oblivious automata. In *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*, page 519, New York, New York, USA, October 2007. ACM Press. ISBN 9781595937032.