Fuzzy Private Matching

Łukasz Chmielewski[1], Jaap-Henk Hoepman[1,2]

[1] Security of Systems (SoS) group
Institute for Computing and Information Sciences
Radboud University Nijmegen
P.O. Box 9010, 6500 GL Nijmegen, the Netherlands
{lukaszc,jhh}@cs.ru.nl
[2] TNO Information and Communication Technology
P.O. Box 1416, 9701 BK Groningen, The Netherlands
jaap-henk.hoepman@tno.nl

Abstract. In the private matching problem, a client and a server each hold a set of n input elements. The client wants to privately compute the intersection of these two sets: he learns which elements he has in common with the server (and nothing more), while the server gains no information at all. In certain applications it would be useful to have a private matching protocol that reports a match even if two elements are only similar instead of equal. Such a private matching protocol is called *fuzzy*, and is useful, for instance, when elements may be inaccurate or corrupted by errors.
We consider the fuzzy private matching problem, in a semi-honest environment. Elements are similar if they match on t out of T attributes. First we show that the original solution proposed by Freedman *et al.* [8] is not private: the client can "steal" elements even if the sets have *no* similar elements in common. Subsequently we present two fuzzy private matching protocols. The first, simple, protocol has message complexity $O(n\binom{T}{t})$. The second, improved, protocol has message complexity $O(nT)$, but here the client incurs a $O(n^2\binom{T}{t})$ time complexity penalty. Additionally, we present a protocol based on the computation of the Hamming distance and on oblivious transfer.

1 Introduction

In the private matching problem [8], a client and a server each hold a set of elements as their input. The size of the set is n and the type of elements is publicly known. The client wants to privately compute the intersection of these two sets: the client learns the elements it has in common with the server (and nothing more), while the server obtains no information at all.

In certain applications, the elements (think of them as words consisting of letters, or tuples of attributes) may not always be accurate or completely known. For example, due to errors, omissions, or inconsistent spelling, entries in a database may not be identical. In these cases, it would be useful to have a private matching algorithm that reports a match even if two entries are similar, but not necessarily equal. Such a private matching is called *fuzzy*, and was introduced by

Freedman *et al.* [8]. Elements are called similar (or matching) in this context if they match on t out of T letters at the right locations.

Fuzzy private matching (FPM) protocols could also be used to implement a more secure and private algorithm of biometric pattern matching. Instead of sending the complete template corresponding to say a scanned fingerprint, a fuzzy private matching protocol could be used to determine the similarity of the scanned fingerprint with the templates stored in the database, without revealing any information about this template in the case that no match is found.

Freedman *et al.* [8] introduce the fuzzy private matching problem and present a protocol for 2-out-of-3 fuzzy private matching. We show that, unfortunately, this protocol is incorrect (see Section 3): the client can "steal" elements even if the sets have *no* similar elements in common. Building and improving on their ideas, we present two protocols for t-out-of-T fuzzy private matching (henceforth simply called fuzzy private matching). The first, simple, protocol has message complexity $O(n\binom{T}{t})$. The second, improved, protocol is based on linear secret sharing and has message complexity $O(nT)$. Here the client incurs a $O(n^2\binom{T}{t})$ time complexity penalty, however. In these solutions data of the participants are encoded as secret shares. Linear secret sharing in this case can be considered as error-correcting code with erasure decoding. In e.g., [4, 13] a relation is defined between error-correcting codes and many versions of secret sharing. In this paper we start to explore the use of coding theory to construct efficient solutions for the fuzzy matching problem.

Indyk and Woodruff [12] present another approach for solving fuzzy private matching, using the computation of the Hamming distance together with generic techniques like secure 2-party computations and oblivious transfer. To compare their results to ours, we use their notation to express the bit complexity of the protocols: for functions f and g, they define $f = \tilde{O}(g)$ if $f(n,k) = O\left(g(n,k)\ \log^{O(1)}(n) \cdot \mathrm{poly}(k)\ \right)$, where k is the security parameter.

Based on the protocol from [12] we design protocols based on the computation of the Hamming distance that do not use secure 2-party computation: one protocol is efficient for small domains of letters and the second protocol uses oblivious transfer. The major drawback of the first protocol is a strong dependence on the size of the domain of letters. The main weakness of the second protocol is its high complexity – in the protocol there are $n^2 \cdot T$ oblivious transfer calls, where one oblivious transfer costs $\tilde{O}(1)$ bit communication. However, the protocols are interesting because they use new techniques, especially in the subroutine `obtain-letters` (that shows a scheme for obtaining an encryption of a single bit using only one oblivious transfer).

Solutions based on a secure function evaluation (using generic secure 2-party computation) work with bit complexity $\tilde{O}(n^2T)$, while the solution of Indyk and Woodruff [12] works in $\tilde{O}(nT^2 + n^2)$. The Freedman *et al.* [8] protocol (though incorrect), as well as our first corrected version work in bit complexity $\tilde{O}(n\binom{T}{t})$. In comparison, our most efficient protocol works in bit complexity $\tilde{O}(nT)$ (however, with the aforementioned increased time complexity of the client). Moreover, our protocols (including the one based on protocols from [8]) do not use generic

secure 2-party computation constructions or oblivious transfer protocols. As a consequence, they are more efficient than would appear from the $\tilde{O}$ notation (see above).

Related work can be traced back to private equality testing [2, 6, 8, 15] in the 2-party case, where each party has a single element and wants to know if they are equal (without publishing these elements). Private set intersection [8, 15, 14] (possibly among more than two parties) is also related. In this problem the output of *all* the participants should be the intersection of all the input sets, but nothing more: a participant should gain no knowledge about elements from other participant's sets that are not in the intersection.

Similarly related are the so called secret handshaking protocols [11, 1, 3]. They consider membership of a secret group, and allow members of such groups to reliably identify fellow group members without giving away their group membership to non-members and eavesdroppers. We note that the (subtle) difference between secret handshaking and set-intersection protocols lies in the fact that a set-intersection protocol needs to be secure for arbitrary element domains (small ones in particular), whereas group membership for handshaking protocols can be encoded using specially constructed secret values taken from a large domain.

Privacy issues have also been considered for the approximation of a function f among vectors owned by several parties. The function f may be Euclidean distance ([5], [7], [12]), set difference ([8]), Hamming distance ([5], [12]), or scalar product (reviewed in [9]).

Our paper is structured as follows. We formally define the fuzzy private matching problem in Section 2, and introduce our system model, some additional notation, and primitives there as well. Then in Section 3 we present the solution from [8] for 2-out-of-3 fuzzy private matching and show where it breaks down. Section 4 contains our first protocol for t-out-of-T fuzzy private matching that uses techniques similar to the ones used in [8]. Then we present our second protocol based on linear secret sharing in Section 5. Finally, Section 6 presents two protocols based on the computation of a Hamming distance. All our protocols assume a semi-honest environment (see Section 2.4).

2 Preliminaries

In this section, we introduce the fuzzy matching problem as well as the mathematical and cryptographic tools that we use to construct our protocols.

2.1 Fuzzy Private Matching Problem Definition

Let a client and a server each own a set of words. A fuzzy private matching scheme is a 2-party protocol between a client and a server, that allows the client to compute the fuzzy set intersection of these sets (without leaking any information to the server).

To be precise, let all the words $X = x^1 \dots x^T$ in these sets consist of T letters x^i from a domain D. We define an auxiliary relation $X \approx_t Y$ among these words

as follows: we say that two words $X = x^1 \dots x^T$ and $Y = y^1 \dots y^T$ match on t letters if and only if $t \leq |\{k : x^k = y^k \cap (1 \leq k \leq T)\}|$. The input and the output of the protocol are defined as follows. The client input is the set $X = \{X_1, \dots X_{n_C}\}$ of n_C words of length T, while the server's input is defined in a similar way: $Y = \{Y_1, \dots Y_{n_S}\}$ of n_S words of length T. Both the client and the server have also in their inputs n_C, n_S, T and t. The output of the client is a set $\{Y_i \in Y | \exists X_i \in X : X_i \approx_t Y_j\}$. This set consist of all the elements from Y that match with any element from the set X. The server's output is empty (the server does not learn anything). Usually we assume that $n_C = n_S = n$. In any case, the sizes of the sets are fixed and a priori known to the other party (so the protocol does not have to prevent the other party to learn the size of the set).

2.2 Additively Homomorphic Cryptosystem

In all our protocols we use a semantically secure, additively homomorphic public-key cryptosystem, e.g., Paillier's cryptosystem [16]. Let $\{\cdot\}_K$ denote the encryption function with the public key K. The homomorphic cryptosystem supports the following two operations, which can be performed without the knowledge of the private key.

1. Given the encryptions $\{a\}_K$ and $\{b\}_K$, of a and b, one can efficiently compute the encryption of $a + b$, denoted $\{a + b\}_K := \{a\}_K +_h \{b\}_K$
2. Given a constant c and the encryption $\{a\}_K$, of a, one can efficiently compute the encryption of $c \cdot a$, denoted $\{a \cdot c\}_K := \{a\}_K \cdot_h c$

These properties hold for suitable operations $+_h$ and $\cdot_h$ defined over the range of the encryption function. In Paillier's system, operation $+_h$ is a multiplication and $\cdot_h$ is an exponentiation.

Remark The domain R of the plaintext of the homomorphic cryptosystem in all of our protocols (unless specified differently) is defined as follows: R should be larger than D^T and a uniformly random element from R should be in D^T with negligible probability. This property can be satisfied by representing an element $a \in D^T$ by $r_a = 0^k || a$ in R. The domain R should be a ring (e.g., $\mathbb{Z}_M$).

Operations on encrypted polynomials We represent any polynomial p of degree n (on some ring) as the ordered list of its coefficients: $[\alpha_0, \alpha_1, \dots \alpha_n]$. We denote the encryption of a polynomial p by $\{p\}_K$ and define it to be the list of encryptions of its coefficients: $[\{\alpha_0\}_K, \{\alpha_1\}_K, \dots \{\alpha_n\}_K]$.

Many operations can be performed on such encrypted polynomials (assuming that the encryption has an additively homomorphic property), like: addition of two encrypted polynomials or multiplication of an encrypted and a plain polynomial. s We use the following property: given an encryption of a polynomial $\{p\}_K$ and some x one can efficiently compute a value $\{p(x)\}_K$. This follows from the

properties of the homomorphic encryption scheme:

$$\{p(x)\}_K = \left\{ \sum_{i=0}^{n} \alpha_i \cdot x^i \right\}_K = \sum_{i=0}^{n}{}_h \{\alpha_i \cdot x^i\}_K = \sum_{i=0}^{n}{}_h \{\alpha_i\}_K \cdot_h x^i$$

2.3 Linear Secret Sharing

Secret sharing refers to any method for distributing a secret among a group of n participants, each of which possesses a share of the secret. The secret can only be reconstructed when at least t shares are combined together. Combining less than t individual shares should give no information whatsoever about the secret. We denote a secret share by $\overline{s}_i$ (for $i \in \{1, \ldots n\}$) and the corresponding secret as $\overline{s}$.

A Linear Secret Sharing (LSS) scheme is a secret sharing scheme with additional properties. In this paper we use the following property: given t shares $\overline{s}_i$ (of secret $\overline{s}$), and t shares $\overline{r}_i$ (of secret $\overline{r}$) on the same indices, using $\overline{s}_i + \overline{r}_i$ one can reconstruct the sum of the secrets $\overline{s} + \overline{r}$. One such LSS scheme is Shamir's original secret sharing scheme [17]. There is strong correspondence between error-correcting codes and secret sharing (e.g., [13]). In our protocols we encode words using linear secret sharing, which in this context can be seen as an error-correcting code with erasure decoding.

2.4 Adversary Models

In this section we describe the adversary model that we use. We prove correctness of our protocols only against a semi-honest adversary. Here we provide the intuition and the informal notion of this model, the reader is referred to [10] for full definitions. To simplify matters we only consider the case of only two participants, the client and the server.

In the model with a semi-honest adversary, both parties are assumed to act accordingly to the protocol (but they are allowed to use all information that they collect in an unexpected way to obtain extra information). The security definition is straightforward in our particular case, as only one party (the client) learns the output. Following [8] we divide the requirements into:

- The client's security – **indistinguishably**: Given that the server gets no output from the protocol, the definition of the client's privacy requires simply that the server cannot distinguish between cases in which the client has different inputs.
- The server's security – **comparison to the ideal model**: The definition ensures that the client does not get more or different information than the output of the function. This is formalized by considering an ideal implementation where a trusted third party TTP gets the inputs of the two parties and outputs the defined function. We require that in the real implementation of the protocol (one without TTP) the client does not learn different information than in the ideal implementation.

1. The client chooses a private key sk, a public key K and *parameters* for the additively homomorphic encryption scheme and sends K and the *parameters* to the server.
2. The client:
   (a) chooses, for every i (such that $1 \leq i \leq n_C$), a random value $r_i \in R$.
   (b) creates 3 polynomials: P_1, P_2, P_3 over R (where polynomial P_j is used to encode all letters on the jth position) defined by the set of equations $r_i = P_1(x_i^1) = P_2(x_i^2) = P_3(x_i^3)$, for $1 \leq i \leq n_C$.
   (c) uses interpolation to calculate coefficients of the polynomials (P_1, P_2, P_3) and sends their encryptions to the server. **Remark:** These polynomials have degree $n_C - 1$ (in [8] it is written that they have degree n_C).
3. For each Y_j (such that $1 \leq i \leq n_S$), the server responds to the client: $\{r \cdot (P_1(y_i^1) - P_2(y_i^2)) + Y_i\}_K$, $\{r' \cdot (P_2(y_i^2) - P_3(y_i^3)) + Y_i\}_K$, $\{r'' \cdot (P_1(y_i^1) - P_3(y_i^3)) + Y_i\}_K$, where r, r', r'' are fresh random values in R. This uses the properties of the homomorphic encryption scheme including the encrypted polynomials explained in Section 2.2.
4. If the client receives an encryption of an encoding of Y_i, which is similar to any word from his set X, then he adds it to the output set.

Fig. 1. original FPM protocol

Due to space constraints our proofs are informal, presenting only the main arguments for correctness and security.

3 The Original FPM Protocol

In this section we present the original fuzzy private matching protocol from Freedman *et al.* [8] (pages: 16–17). We show (following the original paper) the version for $T = 3$ and $t = 2$. Then we present example input data where this protocol fails. The protocol is presented in Figure 1.

The idea behind, and the problem of the protocol from Figure 1 Intuitively the protocol works because if $X_i \approx_2 Y_j$ then, say, $x_i^2 = y_j^2$ and $x_i^3 = y_j^3$. Hence $P_2(x_i^2) = P_2(y_j^2) = r_i$ and $P_3(x_i^3) = P_3(y_j^3) = r_i$ so $P_2(y_i^2) - P_3(y_j^3) = 0$. Then the result $\{r' \cdot (P_2(y_j^2) - P_3(y_j^3)) + Y_j\}_K$ sent back by the server simplifies to $\{Y_j\}_K$ (the random value r' is canceled by the encryption of 0) which the client can decrypt. If X_i and Y_j do not match, the random values r, r' and r'' do not get canceled and effectively blind the value of Y_j in the encryption, hiding it to the client.

There is however a problem with this approach. Consider the following proper input data. The input of the client is $\{[1,2,3]\ ,\ [1,4,5]\}$, while the input of the server is $\{[5,4,3]\}$. Then in step 2c of the protocol, the polynomials are defined (by the client) in the following way:

$\mathbf{P}_1$:	$\mathbf{P}_2$:	$\mathbf{P}_3$:
$P_1(1) = r_1 \cap P_1(1) = r_2$	$P_2(2) = r_1 \cap P_2(4) = r_2$	$P_3(3) = r_1 \cap P_3(5) = r_2$

But now we see that, unless $r_1 = r_2$ (which is unlikely when they are both chosen at random), P_1 remains undefined! Freedman *et al.* do not consider this possibility. However, if we try to remedy this problem by setting $r_1 = r_2$ we run into another one. Among other things, the server computes $\{r' \cdot (P_2(y_i^2) - P_3(y_i^3)) + Y_i\}_K$, which, in this particular case equals $\{r' \cdot (P_2(4) - P_3(3)) + [5,4,3]\}_K$. This equals $\{r' \cdot (r_2 - r_1) + [5,4,3]\}_K$, which by equality of r_1 and r_2 reduces to $\{[5,4,3]\}_K$. In other words, the client learns $[5,4,3]$ even if this value does not match any of the elements held by the client. This violates the requirements of the fuzzy private matching problem: if a semi-honest client happens to own a set of tuples with this property it learns a tuple of the server.

4 Polynomial Based Protocol for the FPM problem

In this section we present our protocol solving the private fuzzy matching problem inspired by the protocol from [8] (presented in Section 3). Our protocol works for any T and t. The protocol is presented in Figure 2. In the protocol we use the following definition of σ. Let σ be a combination of t different indices $\sigma_1, \sigma_2, \ldots, \sigma_t$ from the range $\{1, \ldots, T\}$ (there are $\binom{T}{t}$ of those). For a word $X \in D^T$, define $\sigma(X) = x^{\sigma_1} || \cdots || x^{\sigma_t}$ (i.e., the concatenation of the letters in X found at the indices in the combination).

1. The client chooses a private key sk, a public key K and *parameters* for the additively homomorphic encryption scheme and sends K and the *parameters* to the server.
2. For every combination σ of t out of T indices the client:
   (a) constructs a polynomial:
   $P_\sigma(x) = (x - \sigma(X_1)) \cdot (x - \sigma(X_2)) \cdots (x - \sigma(X_{n_C}))$ of degree n_C
   (b) sends $\{P_\sigma\}_K$ (the encrypted polynomial, see Section 2.2) to the server.
3. For every $Y_i \in Y$, $1 \le i \le n_S$, and every received polynomial $\{P_\sigma\}_K$ (corresponding to the combination σ) the server performs:
   (a) evaluate polynomial $\{P_\sigma\}_K$ at the point $\sigma(Y_i)$ to compute $\{w_i^\sigma\}_K = \{r * P_\sigma(\sigma(Y_i)) + Y_i\}_K$, where $r \in R$ is always a fresh random value.
   (b) send $\{w_i^\sigma\}_K$ to the client.
4. The client decrypts all received messages and if a received message $w_i^\sigma \in D^T$ matches with any word from X then he adds w_i^σ to the output set.

Fig. 2. Corrected Protocol solving FPM problem.

Correctness and security of the protocol from Figure 2 In the protocol, the client produces $\binom{T}{t}$ polynomials P_σ of degree n_C. Every polynomial represents one of the combinations σ of t letters from T letters. It is easy to see that if $X \approx_t Y$ then $\sigma(X) = \sigma(Y)$ for some combination σ. The roots of each polynomial are concatenated letters (of every word in the client set) corresponding to each combination. Hence, if there is an element $Y_j \in Y$ that matches with any $X_i \in X$, then in step 3a of the protocol the value of the evaluated polynomial at a

"matching point" σ is 0 and then the encryption of Y_j is sent to the client. Afterwards the client can recognize this value. Otherwise (if Y_j does not match with any element from X) all the values sent to the client contain a random blinding element r (and therefore their decryptions are in X with negligible probability).

The client's input data is secure because all the data received by the server are encrypted (using a semantically secure cryptosystem). Hence the server cannot distinguish between different client's inputs. The privacy of the server is protected because the client learns about those elements from Y that are also in X. If an element $y_i \in Y$ does not belong to X then a random value is sent by the server (see the correctness proof above).

Complexity The messages being sent in this protocol are encryptions of plaintext from a domain D^T enlarged by k bits (where k is the security parameter). In step 2 the client sends $\binom{T}{t}$ polynomials of degree n_C. Then in step 3 the server responds with n_S values for every polynomial. Hence in total $O((n_S + n_C) \cdot \binom{T}{t})$ messages are sent.

Optimization for a large domain of messages For large D and T the domain of plaintext can be really large and therefore the messages being sent in the protocol can significantly slow down the performance. However there is a way to make this domain smaller by slightly modifying the protocol. For every Y_i the server should prepare a unique secret key sk_i and public key K_i. Then for every Y_i the server sends $E_{K_i}(0^k||Y_i)$ to the client. After that, the protocol remains unchanged, except that in step 3a the server calculates and sends $\{w_i^\sigma\}_K = \{r * P_\sigma(\sigma(Y_i)) + (0^k||sk_i)\}_K$. Later in step 3a the client can distinguish a valid secret key from the random value (by the prefix 0^k) and check to which encryption $E_{K_i}(0^k||Y_i)$ it fits. After the client finds such an encryption he can add it to his output set. In this modified protocol $O((n_S + n_C) \cdot \binom{T}{t})$ messages from a domain of size $O(k)$ and $O(n_s)$ messages from a domain of size $O(\log(|D|^T) + k)$ are sent.

Remarks All the optimizations described in [8] used for the private matching problem can be easily used in this protocol. It is also easy to modify our protocol to be resistant to a malicious adversary (using the protocol resistant to a malicious adversary from [8]). Every polynomial should be protected separately from a malicious adversary (this is a similar situation to $\binom{T}{t}$ instances of private matching problem against a malicious adversary).

5 Secret Sharing Based Protocol for the FPM problem

In this section we present two of our protocols solving the FPM problem. Both of them use the linear secret sharing technique (described in Section 2.3) and work in the model with a semi-honest adversary. First we describe the simple

(but slow) protocol and later the faster, improved one. We present the simple version mainly to facilitate the understanding of the improved protocol.

5.1 A Simple Version of the Protocol

The simple protocol is presented in Figure 3. In this protocol the client first sends encryptions of all his words (every letter is encrypted separately) to the server. Then the participants, for every pair of words from X and Y, run the subroutine `find-matching`(i,j). The aim of a single call of this procedure is to provide Y_j to the client if and only if $X_i \approx_t Y_j$. This is achieved by using a t–out–of–T secret sharing scheme. The client receives a correct share from the server if the corresponding letters x_i^w and y_j^w are equal (otherwise he receives a random value). Hence he can recover Y_i if he receives at least t correct shares (and this happens if and only if at least t letters from X_i are equal to Y_j).

Due to space constraints we skip the proofs of correctness and security of the protocol from Figure 3 (they can be found in the appendix).

1. The client generates sk, K and *parameters* for the additively homomorphic cryptosystem and sends K and the *parameters* to the server.
2. For each $X_i \in X$
   (a) The client encrypts each letter x_i^w of X_i and sends $\{x_i^w\}_K$ to the server.
   (b) For each $Y_j \in Y$, run the protocol `find-matching`(i,j).

`find-matching`(i,j):

1. The server prepares t–out–of–T secret shares $[\overline{s}_1, \overline{s}_2, \ldots \overline{s}_T]$ with secret $0^k||Y_j$, where k is the security parameter.
2. For every letter y_j^w in Y_j, the server computes:
   $v_w = ((\{x_i^w\}_K -_h \{y_j^w\}_K) \cdot_h r) + \{\overline{s}_w\}_K$ which equals $\{((x_i^w - y_j^w) \cdot r + \overline{s}_w)\}_K$, where r is always a fresh, random value from the domain of plaintext.
3. The server sends $[v_1, v_2, \ldots v_T]$ to the client.
4. The client decrypts the values and checks whether it is possible to reconstruct the secret $0^k||z$ from them. In order to do that, he needs to try all possible combinations of t among the T decrypted (potential) shares. If it is possible and z matches X_i then he adds z to his output set.

Fig. 3. Simple protocol solving FPM problem

Complexity The messages being sent in this protocol are encryptions of plaintext from the domain D^T enlarged by k bits (where k is a security parameter). The optimization from Section 4 can be applied to this protocol in a straightforward way.

The main impact on the bit complexity of the protocol is the fact that the subroutine `find-matching` is called $n_C \cdot n_S$ times. In this subroutine, the server sends in step 3 $O(T)$ ciphertexts. Hence, in total $O(n_C \cdot n_S \cdot T)$ messages are sent in this protocol.

The main part of the server time complexity is preparing $n_S \cdot n_C$ times the T secret shares. Producing T secret shares can be done efficiently and therefore

the time complexity of the server is reasonably low. The crucial part for the time complexity of the client is step 4 (it is performed once in every subroutine call). In this step the client verifies if he can reconstruct the secret Y_j. This verification costs $\binom{T}{t}$ reconstructions (and one reconstruction can be done efficiently). The number of reconstructions is in the order of $O(n_S \cdot n_C \cdot \binom{T}{t})$, which is the major drawback of this protocol.

5.2 An Improved Protocol

The improved protocol is presented in Figure 4. It solves the fuzzy matching problem with a few generalizations. Firstly, X and Y could be multisets (instead of sets) and secondly, some additional information that does not change the matching's properties could be attached to any word Y_i from Y, i.e., if the word Y_i matches with some word from X then the additional information is also added to the client's output set (we denote this additional information as $\hat{Y}_i$).

This protocol consists of a polynomial and a ticket phase. In the polynomial phase the server prepares n groups of secret shares. From each group of shares it is possible to reconstruct the corresponding secret $0^k||Y_i||\hat{Y}_i$. The t–out–of–T secret sharing scheme might be used in this situation. However, shares are also used for creating encrypted polynomials: for $w \in \{1,\dots T\}$, if $y_i^w = y_m^w$ then $\overline{s}_{i,w} = \overline{s}_{m,w}$. In this case it may be impossible to create groups of shares that have different secrets, e.g., two matching, but different words from Y, would have the same secret, because more than t secrets would be the same. We solve that problem by adding some additional shares that are sent in plaintext. They are chosen only to enable setting different secrets.

After the polynomial phase the client has n lists of groups of encryptions of potential secret shares and n list of groups of unencrypted shares. From all these shares, if they are unencrypted, the client is able to recover his output set. However, if he receives all of the shares in plaintext he can abuse the protocol by gaining illicit information (he would be able to do so by connecting shares from different groups of potential shares). To prevent this there is a ticket phase, which aims at protecting the server's privacy. At the beginning of this phase the client has n encrypted groups of potential secret shares that he wants to be decrypted by the server. For each group he sends his encrypted potential shares blinded by random values. To make a client's group independent (from other client's groups) the server generates a new group of secret shares (called a ticket) for a secret 0. Subsequently, he decrypts the blinded shares and adds the corresponding ticket's shares. Later he sends the results to the client together with those ticket's shares that correspond to shares sent in plaintext in the polynomial phase. Notice that this modification of the client's potential secret shares does not affect the potential secret (because the ticket's shares' secret is 0 and because of the linear property of the secret sharing scheme). The client can unblind the received values and try to recover the secret from the polynomial phase. This phase makes lists of groups of independent secret shares and therefore the client cannot mix shares from different groups to abuse the protocol.

Polynomial Phase:

1. **The server** prepares sk, K and *parameters* for the additively homomorphic cryptosystem and sends K and the *parameters* to the client.
2. For all $Y_i \in Y$, the server prepares $[T+1]$–out–of–$[2 \cdot T - t + 1]$ secret shares $[\overline{s}_{i,1}, \overline{s}_{i,2}, \ldots \overline{s}_{i,2 \cdot T - t + 1}]$ with the secret $0^k || Y_i || \hat{Y}_i$, where k is the security parameter. If $y_i^w = y_m^w$ then $\overline{s}_{i,w} = \overline{s}_{m,w}$.
   The server sends $[\overline{s}_{i,T+1}, \ldots \overline{s}_{i,2 \cdot T - t + 1}]$ to the client.
3. The server prepares T polynomials (for $w = 1$ to T) of degree n :
   (a) $((P_w(y_1^w) = \overline{s}_{1,w}) \cap (P_w(y_2^w) = \overline{s}_{2,w}) \cap \ldots (P_w(y_n^w) = \overline{s}_{n,w}))$
   (b) The server encrypts each polynomial $\{P_w\}_K$ and sends it to the client.
4. The client evaluates T polynomials (for $w = 1$ to T) on each letter of each word (for $i = 1$ to n): $\{v_i^w\}_K = \{P_w(x_i^w)\}_K$. If $x_i^w = y_m^w$ then $v_i^w = \overline{s}_{m,w}$.
5. The client blinds the results v_i^w with a random values r_i^w and sends them to the server: $\{v_i^w + r_i^w\}_K$.

Ticket Phase:

6. For $i = 1$ to n, the server prepares $[T+1]$–out–of–$[2 \cdot T - t + 1]$ secret shares $[\overline{\tau}_{i,1}, \overline{\tau}_{i,2}, \ldots \overline{\tau}_{i,2 \cdot T - t + 1}]$ with secret 0. Later he sends $[\overline{\tau}_{i,T+1}, \ldots \overline{\tau}_{i,2 \cdot T - t + 1}]$ to the client.
7. For $i = 1$ to n and for $w = 1$ to T, the server decrypts the received messages $D_{sk}(\{v_i^w + r_i^w\}_K)$ and sends $(v_i^w + r_i^w + \tau_i^w)$ to the client.
8. The client unblinds them (by subtracting r_i^w) obtaining q_i^w.
   If $x_i^w = y_m^w$ then $q_i^w = \overline{s}_{m,w} + \overline{\tau}_{i,w}$.
9. For $i = 1$ to n and $j = 1$ to n, the client checks if it is possible to reconstruct the secret $0^k || z || \hat{z}$ from: $[q_i^1, q_i^2, \ldots q_i^T, \overline{s}_{j,T+1} + \overline{\tau}_{i,T+1}, \overline{s}_{j,T+2} + \overline{\tau}_{i,T+2}, \ldots \overline{s}_{j,2 \cdot T - t + 1} + \overline{\tau}_{i,2 \cdot T - t + 1}]$.
   In order to do that, the client needs to try all possible combinations of t shares among the T decrypted q shares (the rest of the shares is the same during reconstructions). If it is possible and z is similar to any $X_i \in X$ then he adds $z || \hat{z}$ to his output set.

Fig. 4. Improved protocol solving FPM problem

Correctness and security of the protocol from Figure 4 The first important issue appears in step 2 of the polynomial phase. Here the server prepares n groups of $[T+1]$–out–of–$[2 \cdot T - t + 1]$ shares $[\overline{s}_{i,1}, \overline{s}_{i,2}, \ldots \overline{s}_{i,2 \cdot T - t + 1}]$. From the ith group he can recover Y_i. During the creation of these shares the server uses the rule:

$$\text{for } w \in \{1, \ldots T\}\text{: if } y_i^w = y_m^w \text{ then } \overline{s}_{i,w} = \overline{s}_{m,w}. \tag{1}$$

This rule is necessary because the first T shares from each group are later encoded as polynomials.

This secret sharing is used here in the same role as the t–out–of–T one. However if the t–out–of–T scheme is used, then it is impossible to choose the proper value of secrets (e.g., two matching, but different, words from Y, would have the same secret because of Rule 1). Secret shares $[\overline{s}_{i,T+1}, \ldots \overline{s}_{i,[2 \cdot T - t + 1]}]$ are chosen arbitrarily only to enable proper values of the secrets. To choose arbitrary secrets even for equal words (Y could be a multiset) $(T - t + 1)$ new shares (the ones that are sent in plaintext) is exactly enough. The role of shares $[\overline{s}_{i,1}, \ldots \overline{s}_{i,T}]$ is like in classical secret sharing. Because the last $T - t + 1$ shares are known, the first T shares work like a t–out–of–T secret sharing scheme.

Subsequently, in step 3, the server creates T polynomials of degree n in such a way that evaluating a polynomial on a corresponding letter from some word from Y results in a corresponding secret share. Later he sends the encrypted polynomials to the client. The client evaluates the polynomials on his words and achieves $\{v_i^w\}_K$ (where the following property holds: if $x_i^w = y_m^w$ then $v_i^w = \overline{s}_{m,w}$). After the ticket phase, the client receives T values $q_i^w = v_i^w + \overline{\tau}_{i,w}$, where $[\overline{\tau}_{i,1}, \overline{\tau}_{i,2}, \dots \overline{\tau}_{i,T}]$ are tickets – secret shares with the secret 0. Hence the client receives the group: $[v_i^1 + \overline{\tau}_{i,1}, v_i^2 + \overline{\tau}_{i,2}, \dots v_i^T + \overline{\tau}_{i,T}]$, where if $x_i^w = y_m^w$ (for some $Y_m \in Y$) then $v_i^w = \overline{s}_{m,w}$. Therefore, by the linear property of LSS, if v_i^w is a correct secret share, then $q_i^w = v_i^w + \overline{\tau}_{i,w}$ is also a correct secret share. The client is trying to recover a secret for every received group of potential shares. However, for a proper reconstruction, he also needs shares that have been sent to him in plaintext by the server. These shares are always correct, but he needs to combine shares from the polynomial and ticket phases. Moreover, he does not know which shares from the polynomial phase correspond to the shares from the ticket phase. As a result, the client has to check all of the combinations (n^2). If the client combines non-fitting shares then he cannot recover the proper secret.

Hence, for $i, j \in \{1, \dots n\}$, the client checks if he can reconstruct the secret from the following shares:

$$[q_i^1,\ q_i^2,\ \dots\ q_i^T,\ \ \overline{s}_{j,T+1} + \overline{\tau}_{i,T+1},\ \overline{s}_{j,T+2} + \overline{\tau}_{i,T+2},\ \dots\ \overline{s}_{j,2\cdot T-t+1} + \overline{\tau}_{i,2\cdot T-t+1}]$$

If enough corresponding secret shares are in the group q_i, then the secret that could be recovered from them is $0^k||Y_m||\hat{Y_m}$ (because the secret of τ shares is 0). Hence, in step 9 the client recovers all of the secrets that he has corresponding shares of.

The privacy of the client's input data is secure because all of the data received by the server (in step 5 of the polynomial phase) is of the form: $v_i^w + r_i^w$, where r_i^w is a random value from the domain of the plaintext. Hence the server cannot distinguish between different client inputs.

The privacy of the server is protected because the client receives correct secret shares of some $Y_j \in Y$ if and only if there is an element $X_i \in X$ such that $X_i \approx_t Y_j$. In the polynomial phase, the client receives encrypted polynomials and n groups with $T - t + 1$ shares ($[\overline{s}_{i,T+1}, \dots \overline{s}_{i,[2\cdot T-t+1]}]$) of $[T + 1]$–out–of–$[2 \cdot T - t + 1]$ secret sharing scheme. Hence there is no leakage of information in the polynomial phase. The client receives information in plaintext in steps 6 and 7 of the ticket phase. In this situation, the client has at least $T + 1$ correct secret shares during step 7 and he can reconstruct the secret $0^k||Y_m||\hat{Y_m}$.

If there is no such element in X to which Y_j is similar, then the client receives no more than t shares in every group q_i of potential shares: $q_i^w = \overline{\tau}_{i,w} + \overline{s}_{j,w}$ (where i is an index of the received group of potential shares). The client cannot reconstruct Y_j for any group separately (by the secret sharing assumption), because he has less than $T + 1$ correct secret shares. Of all the shares, $(T - t + 1)$ come from values that are sent in plaintext. For every group of shares, τ values are different and therefore make every received group of shares independent.

The probability that a random value from R is a correct share is negligible (with respect to the security parameter k). Therefore, the probability that the client can recover illicit information is negligible.

Complexity Ciphertexts being sent in this protocol are encryptions of plaintext from the domain that contains D^T enlarged by k bits (where k is a security parameter). There are also messages being sent that are unencrypted, but less than ciphertexts. The optimization from Section 4 can be applied to this protocol in a straightforward way.

In step 3 the server sends encryptions of T polynomials of degree n and n lists of $T-t+1$ unencrypted shares. This totals to $O(n \cdot T)$ messages. For every received polynomial, the client computes n values and sends them encrypted to the server (again $O(n \cdot T)$ messages). In the ticket phase, the server responds to every received message by sending one unencrypted value. Moreover he again sends n lists of $T-t+1$ tickets. Hence in the entire protocol, $O(n \cdot T)$ messages are sent.

The main part of the server time complexity is preparing $2 \cdot n$ times $[T+1]$–out–of–$[2 \cdot T - t + 1]$ secret shares. Since producing $(2 \cdot T - t + 1)$ secret shares can be done efficiently, the time complexity of the server is reasonable. The crucial part for the time complexity of the client is step 9 (which is performed n^2 times). In this step the client checks whether he can reconstruct the secret Y_j. This verification costs $\binom{T}{t}$ reconstructions (and one reconstruction can be done efficiently). The total number of reconstructions is in the order of $O(n^2 \cdot \binom{T}{t})$, which is the major drawback of this protocol.

6 Hamming Distance Based Protocol for the FPM Problem

In this section we present two protocols solving the FPM problem based on computing the encrypted Hamming distance: one that is simple and efficient for small domains and another that uses oblivious transfer. The difference between them is only the implementation of the subroutine `equality-matrix` (the frame of the protocol is the same for both of them). Firstly we describe the simple protocol and later the one using oblivious transfer.

A technique to compute the encrypted Hamming distance to solve the FPM problem has been introduced in [12]. However, the protocol in that paper uses generic 2-party computations together with oblivious transfer, making their approach less practical.

Our protocol (see Figure 5) works as follows. The server first obtains, using the subroutine `equality-matrix`, a 3-dimensional matrix $f(w,i,j)$ containing the encrypted equality test for the w-th letter in words X_i and Y_j (where $\{0\}_K$ denotes equality and $\{1\}_K$ denotes inequality). The server sums the entries in this matrix to compute the encrypted Hamming distance $d_i^j = \Delta(X_i, Y_j)$ between the words X_i and Y_j. Subsequently, the server sends Y_j blinded by a random value r multiplied by $d-\ell$, for all $0 \le \ell \le T-t$. If $0 \le d \le T-t$, then for some ℓ the

1. The client prepares sk, K and the *parameters* for the additively homomorphic cryptosystem and sends K and the *parameters* to the server.
2. Run subroutine `equality-matrix`. After this subroutine the server has obtained the following matrix: $f(w,i,j) = \begin{cases} \{0\}_K, \text{ for } x_i^w = y_j^w \\ \{1\}_K, \text{ for } x_i^w \neq y_j^w \end{cases}$, where $w \in \{1,\dots T\}$ and $i,j \in \{1,\dots n\}$
3. For each $X_i \in X$ and $Y_j \in Y$:
   (a) the server computes $\{\Delta(X_i,Y_j)\}_K = \{\sum_{w=1}^{T} f(i,j,w)\}_K$ and, for $\ell = 0$ to $T-t$, sends $\{(\Delta(X_i,Y_j)-\ell)\cdot r + (0^k||Y_j))\}_K$ to the client. Here r is always a fresh, random value.
   (b) The client decrypts all $T-t$ messages and if any plaintext is in D^T and matches any word from X, then the client adds this plaintext to the output set.

Fig. 5. Hamming distance based protocol for FPM problem

value Y_j is not blinded at all. This allows the client to recover Y_j. Otherwise Y_j is blinded by some random value for every ℓ, and the client learns nothing.

Correctness and Security of the protocol from Figure 5 Assuming that in the subroutine `equality-matrix` the matrix f has been securely obtained, protocol 5 calculates a correct output. This can be concluded from the following facts: if $X_i \approx_t Y_j$ then (in step 3a) $\Delta(X_i,Y_j) \in \{0 \dots T-t\}$, and therefore $\{0^k||Y_j\}_K$ is sent to the client. Privacy of the server is protected because in step 3a if $X_i \not\approx_t Y_j$ then $\Delta(X_i,Y_j) \notin \{0,\dots T-t\}$ and therefore all values received by the client look random to him. Correctness and security proofs of this protocol resemble the proofs of the protocol presented in Figure 3 and are omitted here.

6.1 Implementing Subroutine `equality-matrix`

The first method to implement the subroutine `equality-matrix` is as follows. The client sends the letters of all his words to the server as encrypted vectors d_i^w: $\{0,\dots|D|-1\}$ (where $i \in \{1,\dots n_C\}$ and $w \in \{1,\dots T\}$) such that $d_i^w(v) = \{1\}_K$ if $v = x_i^w$, and $d_i^w(v) = \{0\}_K$ otherwise. Subsequently the server defines the matrix as $f(w,i,j) = d_i^w(y_j^w)$. The main drawback of this method is that its bit complexity includes a factor $O(|D|\cdot n\cdot T + n^2\cdot(T-t))$. However, the protocol is simple, and for small domains D it is efficient. For constant size D and $T \approx t$ the bit complexity of the protocol reduces to $\tilde{O}(n^2 + n\cdot T)$ (which is significantly better than the bit complexity of the protocol from [12] in this situation).

The second implementation of the subroutine is shown in Figure 6. This implementation uses 1-out-of-q oblivious transfer. An oblivious transfer is a 2-party protocol, where a client has a vector of q elements, and the server chooses any one of them in such a way that the server does not learn more than one, and the client remains oblivious to the value the server chooses. Such an oblivious transfer protocol is described in [15]. The fastest implementation of oblivious transfer works in time $\tilde{O}(1)$.

The second version of the subroutine `equality-matrix` uses such an oblivious transfer in the following way. Let d_i^w be the unary encoding of x_i^w as defined

1. The client generates vectors d_i^w: $[0, \ldots |D| - 1]$ (where $i \in \{1, \ldots n_C\}$ and $w \in \{1, \ldots T\}$) such that: $d_i^w(v) = 1$ if $v = x_i^w$, and $d_i^w(v) = 0$ otherwise.
2. The matrix f is defined in the following way (for all $i, j \in \{1, \ldots n\}$ and $w \in \{1, \ldots T\}$):
   (a) The client picks a random bit $b_{i,j}^w$.
   (b) The server and the client perform 1–out–of–$|D|$ oblivious transfer as follows. The client constructs $h_{i,j}^w$, which is a vector $[0, \ldots |D| - 1]$ as follows: $h_{i,j}^w = [d_i^w(0) \oplus b_{i,j}^w, d_i^w(1) \oplus b_{i,j}^w, \ldots d_i^w(|D| - 1) \oplus b_{i,j}^w]$.
   The server wants to obtain a value from the vector $h_{i,j}^w$ with an index y_j^w. For that they perform the oblivious transfer protocol (where the server has an index and the client an array). Subsequently, the server obtains the value $h = h_{i,j}^w(y_j^w)$.
   (c) The client sends $\{b_{i,j}^w\}_K$ to the server.
   (d) $f(w, i, j) = \begin{cases} \{b_{i,j}^w\}_K, & \text{for } h = 0 \\ \{1 - b_{i,j}^w\}_K, & \text{for } h = 1 \end{cases}$

Fig. 6. Subroutine `equality-matrix` based on oblivious transfer

above (in the description of first method of implementation). The client chooses a random bit $b_{i,j}^w$. Next he constructs a vector $h_{i,j}^w$ which contains all bits of d_i^w, each blinded by the random bit $b_{i,j}^w$. In other words $h_{i,j}^w[x] = d_i^w(x) \oplus b_{i,j}^w$. Using an oblivious transfer protocol, the server requests the y_j^w-th entry in this vector, and obtains $d_i^w(y_j^w) \oplus b_{i,j}^w$. By the obliviousness, the client does not learn y_j^w, and the server does not learn any other entry. Subsequently, the client sends the encryption $\{b_{i,j}^w\}_K$ to the server. Based on this the server constructs $f(w, i, j) = \{d_i^w(y_j^w)\}_K$ as explained in the protocol.

Corollary These protocols are less efficient in bit complexity than the improved protocol (see Section 5.2, Figure 4). The first protocol is efficient for small domains, but significantly inefficient for large ones. In the second protocol there are $n^2 \cdot T$ oblivious transfer calls. Moreover, at this stage, we do not foresee a way to improve these protocols. However, the protocols are interesting because they do not use generic 2-party computations. Furthermore, the techniques being used contain novel elements especially in the subroutine `equality-matrix`, that presents a technique for obtaining the encryption of a single bit using only one oblivious transfer.

7 Summary and Future Work

In this paper we have shown a few protocols solving the FPM problem. The most efficient one works in a linear bit complexity with respect to the size of the input data. However we cannot call this protocol really efficient because of the slow time complexity of the client.

Currently, we are investigating how to speed up the time complexity of the client by using error correcting coding techniques. Instead of representing elements Y_i of the server's set by linear secret shares we are trying to represent

them as codewords of linear error-correcting codes (to achieve faster reconstruction of secrets). However, we then encounter problems violating the privacy of the server.

References

[1] Dirk Balfanz, Glenn Durfee, Narendar Shankar, Diana Smetters, Jessica Staddon, and Hao-Chi Wong. Secret handshakes from pairing-based key agreements. In *24th IEEE Symposium on Security and Privacy*, page 180, Oakland, CA, May 2003.

[2] Fabrice Boudot, Berry Schoenmakers, and Jacques Traoré. A fair and efficient solution to the socialist millionaires' problem. *Discrete Applied Mathematics*, 111(1–2):23–36, 2001.

[3] Claude Castelluccia, Stanislaw Jarecki, and Gene Tsudik. Secret handshakes from ca-oblivious encryption. In *In Advances in Cryptology - ASIACRYPT 2004: 10th International Conference on the Theory and Application of Cryptology and Information Security*, volume 3329, pages 293–307, December 2004.

[4] C. Ding, T. Laihonen, and A. Renvall. Linear multisecret-sharing schemes and error-correcting codes. *Journal of Universal Computer Science*, 3(9):1023–1036, 1997.

[5] Kevi Du and Mike Atallah. Protocols for secure remote database access with approximate matching. In *the First Workshop on Security and Privacy in E-Commerce, Nov. 2000.*, November 2000.

[6] Ronald Fagin, Moni Naor, and Peter Winkler. Comparing information without leaking it. *Communications of the ACM*, 39(5):77–85, 1996.

[7] Joan Feigenbaum, Yuval Ishai, Tal Malkin, Kobbi Nissim, Martin J. Strauss, and Rebecca N. Wright. Secure multiparty computation of approximations. *Lecture Notes in Computer Science*, 2076:927+, 2001.

[8] Michael Freedman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology — EUROCRYPT 2004.*, pages 1–19, 2004.

[9] Bart Goethals, Sven Laur, Helger Lipmaa, and Taneli Mielikainen. On private scalar product computation for privacy-preserving data mining. *Lecture Notes in Computer Science*, 3506:104–120, 2004.

[10] Oded Goldreich. *Secure multi-party computation.* Cambridge University Press, 2002.

[11] Jaap-Henk Hoepman. Private handshakes. In *4th Eur. Symp. on Security and Privacy in Ad hoc and Sensor Networks*, 2007.

[12] Piotr Indyk and David Woodruff. Polylogarithmic private approximations and efficient matching. In *The third Theory of Cryptography conference 2006*, volume 3876 of LNCS, pages 245–264, 2006.

[13] Aggelos Kiayias and Moti Yung. Polynomial reconstruction based cryptography. In *SAC '01: Revised Papers from the 8th Annual International Workshop on Selected Areas in Cryptography*, pages 129–133, London, UK, 2001. Springer-Verlag.

[14] Lea Kissner and Dawn Song. Privacy-preserving set operations. In *Advances in Cryptology — CRYPTO 2005.*, pages 68–80, 2005.

[15] Moni Naor and Benny Pinkas. Oblivious transfer and polynomial evaluation. In *Thirty-First Annual ACM Symposium on the Theory of Computing*, pages 245–254, May 1999.

[16] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology — EUROCRYPT 1999.*, pages 223–238, May 1999.

[17] Adi Shamir. How to share a secret. In *Communications of the ACM, vol. 22, n.11*, pages 612–613, November 1979.

A Correctness and security of the protocol from Figure 3

In this protocol the client encrypts all of his words and sends the results to the server. Then for every couple of words (X_i, Y_j), the participants run the subroutine `find-matching`. In the subroutine the server divides his words into T shares (with threshold t) and for every letter in Y_j calculates $v_w = \{((x_i^w - y_j^w) \cdot r + \overline{s}_w)\}_K$. If $x_i^w = y_j^w$ then the client receives the correct share, otherwise a random value. However, at this step the client cannot distinguish in which situation he is (he cannot distinguish a random value from the correct share). Then the client checks if he can reconstruct the secret using any combination of t out of the T elements $\{D_{sk}(v_w) | 1 \leq w \leq T\}$. He recognizes the secret by the 0^k prefix, and similarity with one of the words from his set. If he has less than t correct secret shares then he cannot recover the secret, and the retrieved data looks random to him (this follows from the security of the secret sharing scheme). Hence all required elements from Y appear in the client's output. The probability that some incorrect element is in the output set is negligible.

The client input data is secure because all of the data received by the server is encrypted (using the semantically secure cryptosystem). Hence the server cannot distinguish between different client inputs.

Privacy of the server is protected because the client receives correct secret shares of some $Y_j \in Y$ if and only if there is an element $X_i \in X$ such that $X_i \approx_t Y_j$. In this situation the client has at least t correct secret shares and he can reconstruct the secret $0^k || Y_j$. If there is no element in X to which Y_j is similar then the client receives n_S independent groups of shares, which has no group with at least t correct shares. Hence from any of these groups he cannot retrieve any secret. The probability that a random value from R is a correct share is negligible (with respect to security parameter k). Therefore the probability that the client can recover an illicit secret is negligible.