CERIAS Tech Report 2005-38

SECURE AND PRIVATE SEQUENCE COMPARISONS

by Mikhail J. Atallah, Florian Kerschbaum, and Wenliang Du

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Secure and Private Sequence Comparisons [*]

Mikhail J. Atallah
Department of Computer Sciences and CERIAS
Purdue University
West Lafayette, IN 47907
mja@cs.purdue.edu

Florian Kerschbaum
Department of Computer Sciences and CERIAS
Purdue University
West Lafayette, IN 47907
fk@cerias.purdue.edu

Wenliang Du
Department of Electrical Engineering and Computer Science
Syracuse University
Syracuse, NY 13204
wedu@ecs.syr.edu

ABSTRACT

We give an efficient protocol for sequence comparisons of the edit-distance kind, such that *neither party reveals anything about their private sequence to the other party* (other than what can be inferred from the edit distance between their two sequences – which is unavoidable because computing that distance is the purpose of the protocol). The amount of communication done by our protocol is proportional to the time complexity of the best-known algorithm for performing the sequence comparison.

The problem of determining the similarity between two sequences arises in a large number of applications, in particular in bioinformatics. In these application areas, the edit distance is one of the most widely used notions of sequence similarity: It is the least-cost set of insertions, deletions, and substitutions required to transform one string into the other. The generalizations of edit distance that are solved by the same kind of dynamic programming recurrence relation as the one for edit distance, cover an even wider domain of applications.

Categories and Subject Descriptors

F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems;
D.4.6 [**Operating Systems**]: Security and Protection

General Terms

Algorithms, Security

Keywords

Dynamic programming, edit distance, longest common subsequence, privacy, secure multi-party computation, string matching

1. INTRODUCTION

We begin with a broad motivation for the framework considered in this paper. Suppose one bio-engineering company (call it "Alice") has a requirement for a certain biosequence λ and wants to check whether another company (call it "Bob") already has a sequence *similar* (not necessarily exactly equal) to λ (in which case Alice may want to negotiate its purchase from Bob). But Alice does not want Bob to know what Alice is looking for, and similarly Bob does not want its proprietary catalog of sequences unnecessarily revealed. It could be valuable for the companies to have the ability to perform such comparisons, without either of them revealing anything about their own sequence (other than the cooperatively computed distance between their respective sequences). If one of the entities is an individual, then there are privacy reasons for not wanting to send one's DNA to a corporation – the danger is that one could send it for one purpose X, and later on it also gets used for another (unauthorized) purpose Y; it is better if there is a protocol for X that gets the job done yet inherently safeguards the privacy of the participants' data. The protocols we give in this paper achieve such a result for a certain kind of sequence comparison (the edit distance), and can also be modified so that only one of the two parties learns the answer. We believe this kind of framework deserves more investigation in the bioinformatics community, and our paper is a first step in that direction.

One of the fundamental methods for molecular sequence comparison and alignment is the Needleman-Wunsch algorithm [12], which is used in software for detecting similarities between two DNA sequences. The underlying sequence comparison and alignment problem is also known as the *string edit* problem in the literature. The dynamic programming recurrence relation that subtends the solution to this problem, also serves to solve many other important related problems (either as special cases, or as generalizations that have the same dynamic programming kind of solution). These include the longest common subsequence problem, and the problem of approximate matching between a pattern sequence and text sequence (there is a huge literature of published work for the notion of approximate pattern matching

[*] Portions of this work were supported by Grants EIA-9903545, IIS-0219560, IIS-0312357, and IIS-0242421 from the National Science Foundation, Contract N00014-02-1-0364 from the Office of Naval Research, by sponsors of the Center for Education and Research in Information Assurance and Security, and by Purdue Discovery Park's e-enterprise Center.

WPES'03, October 30, 2003, Washington, DC, USA.

and its connection to the sequence alignment problem). Any solution to the general sequence comparison problem could also be used to solve these related problems. For example, our protocol can remotely compute the answer to the `Unix` command

`diff` *file1 file2* | `wc -l`

where *file1* is with Alice and *file2* is with Bob, without Alice learning anything about *file2* and without Bob learning anything about *file1*.

We now precisely state the problem. We first describe the *weighted edit distance* problem, in which the cost of an insertion or deletion or substitution is a symbol-dependent weight that can be different from 1, and the edit distance is then the *least-cost* set of insertions, deletions, and substitutions required to transform one string into the other. More formally, if we let λ be a string of length n, $\lambda = \lambda_1 \ldots \lambda_n$ and μ be a string of length m, $\mu = \mu_1 \ldots \mu_m$, both over some alphabet Σ. There are three types of allowed *edit operations* to be done on λ: insertion of a symbol, deletion of a symbol, and substitution of one symbol by another. Each operation has a cost associated with it, namely $I(a)$ denotes the cost of inserting the symbol a, $D(a)$ denotes the cost of deleting a, and $S(a,b)$ denotes the cost of substituting a with b. Each sequence of operations that transforms λ into μ has a *cost* associated with it (= the sum of the costs of the operations in it), and the least-cost such sequence is the *edit-distance.* Our solution allows arbitrary $I(a)$, $D(b)$, and $S(a,b)$ values, and we give more practical solutions for two special cases: (i) $S(a,b) = |a-b|$, and (ii) unit insertion/deletion cost and $S(a,b) = 0$ if $a = b$ and $S(a,b) = +\infty$ if $a \neq b$ (in effect forbidding substitutions).

The *longest common subsequence* problem is a special case of edit distance, where insertions and deletions have unit-cost, $I(a) = D(a) = 1$, and substitutions are only used if a and b are equal, $S(a,b) = 0$ if $a = b$, else $S(a,b) = +\infty$. The length l of the longest common subsequence is then easily seen to be related to the edit distance e in the following way:

$$l = \frac{n+m-e}{2}.$$

2. RELATED WORK

As mentioned above, the edit distance problem arises in a large number of application areas. For this reason, it has been studied rather extensively in the past, and forms the object of several papers (e.g. [9, 10, 12, 14, 15, 17, 20], to list a few). The problem is solved by an algorithm that runs in $\Theta(mn)$ time, through dynamic programming (cf. for example, [20]). Such a performance represents a lower bound when the queries on symbols of the string are restricted to tests of equality [1, 21]. Many important problems are special cases of the edit distance problem, including the above-mentioned longest common subsequence problem and the problem of *approximate matching* between a pattern string and text string (see [8, 18, 19] for the notion of approximate pattern matching and its connection to the edit distance problem). Needless to say that our solution to the edit distance problem implies similar bounds for all of these special cases.

The problem of computing the edit distance without Alice revealing her string λ to Bob and without Bob revealing his string μ to Alice, is a special case of secure multi-party computation. Elegant general constructions have been developed to solve any secure two-party (or even multi-party) function computation [6, 22]. Goldreich recommends that these general solutions not be used in practice, but that more efficient problem-specific solutions should be developed [6]. Many such solutions to specific problems have been developed, but not for the edit distance problem. The previous work most related to this paper is [4], where protocols for other distance metrics were given, including Hamming distance, the L_1 and L_2 distance metrics. All these metrics considered in [4] were between strings that have *the same length* as each other – it is indeed a limitation of the techniques in [4] that they do not extend to the present situation where the strings are of different length and therefore insertions and deletions have to be allowed. This makes the problem substantially different, as the edit distance algorithm is described by a dynamic program that computes it, rather than as a simple one-line mathematical expression.

3. BUILDING BLOCKS

Giving the full-fledged protocol would make it too long and rather hard to comprehend. This section aims at making the later presentation of the protocol much crisper by presenting some of the ideas and building blocks for it ahead of time, right after a brief review of the standard dynamic programming solution to string edit.

3.1 Edit Distance via Dynamic Programming

As is evident from [15], the dynamic programming idea for computing edit distance was independently discovered by over a dozen groups of researchers – each group working in one particular application area (thus whereas computer scientists call it the Wagner-Fischer technique, biologists call it Needleman-Wunsch).

Let $M(i,j)$, $(0 \leq i \leq n,\ 0 \leq j \leq m)$ be the minimum cost of transforming the prefix of λ of length i into the prefix of μ of length j, i.e., of transforming $\lambda_1 \ldots \lambda_i$ into $\mu_1 \ldots \mu_j$. Then

$$\begin{array}{ll} M(0,0) = 0 & \\ M(0,j) = \sum_{k=1}^{j} I(\mu_j) & \text{for } 1 \leq j \leq m \\ M(i,0) = \sum_{k=1}^{i} D(\lambda_i) & \text{for } 1 \leq i \leq n \end{array}$$

and for positive i and j, $1 \leq i \leq n$ and $1 \leq j \leq m$ we have

$$M(i,j) = \min \left(\begin{array}{l} M(i-1,j-1) + S(\lambda_i, \mu_j) \\ M(i-1,j) + D(\lambda_i) \\ M(i,j-1) + I(\mu_j) \end{array} \right).$$

Hence $M(i,j)$ can be evaluated row-by-row or column-by-column in $\Theta(mn)$ time [20]. Observe that, of all entries of the M-matrix, only the three entries $M(i-1,j-1)$, $M(i-1,j)$ and $M(i,j-1)$ are involved in the computation of the final value of $M(i,j)$.

Not only does the above dynamic program for computing M depend on both λ and μ, but even if M could be computed without exchanging λ and μ, the problem remains that M itself is too revealing: It reveals not only the overall edit distance, but also the edit distance from every prefix of λ to every prefix of μ. It is a requirement of our problem that the only information revealed to Alice and Bob by the protocol is the overall edit distance. The matrix M should therefore be known neither to Alice nor to Bob. Only at then end is $M(n,m)$ revealed to Alice and Bob (or to only one of them, if that is the goal).

3.2 How M is Stored

Our edit distance protocol computes the same matrix as the dynamic programming algorithm, in the same order (e.g., row by row). A crucial difference is that the matrix M is additively shared between Alice and Bob: Alice and Bob each hold a matrix M_A and M_B, respectively, the sum of which is the matrix M, i.e., $M = M_A + M_B$. The protocol will keep this as an invariant through all its steps. This does not solve all privacy problems of the computation, but it will be one of the guiding principles of our protocol. The problem is more difficult than that, because the result of each comparison, as well as the indices of the minimum elements, have to be shared (in the sense that neither party individually knows them).

Notation: Throughout the paper, items subscripted with A are known to Alice but not to Bob, those subscripted with B are known to Bob but not to Alice.

3.3 Homomorphic Encryption

In a homomorphic encryption scheme, the following holds: $E(a) * E(b) = E(a + b)$. Several such encryption systems have been proposed that operate over a "modulo-group" [2, 7, 11, 13]. We assume that the group size is large enough for $a + b$ and $a - b$ to avoid "wrap-around"; in our case this is easy to achieve because the numbers involved are in a small range, hence (if computations are modulo n) we can shift the whole range of interest up by $n/2$ and no wraparound will occur (we henceforth assume this is done and avoid repeating it explicitly in what follows, to avoid unnecessarily cluttering the exposition). The encryption scheme should also be public-key and semantically secure, i.e., $E(a)$ gives no information about a. Several steps of our protocol use homomorphic encryption.

3.4 Minimum Finding Protocol for Split Data

Assume that Alice and Bob share a vector $\vec{c}$ additively, i.e., $\vec{c} = \vec{a} + \vec{b}$ such that Alice has $\vec{a} = (a_1, \ldots, a_l)$ and Bob has $\vec{b} = (b_1, \ldots, b_l)$. They want to compute the minimum element of the vector $\vec{c} = \vec{a} + \vec{b}$. As a cryptographic tool they can use a protocol for *Yao's millionaire problem* [22], which privately compares two values α (held by Alice) and β (held by Bob), such that Alice and Bob only learn if $\alpha \geq \beta$ is true. Various protocols for this have been proposed (e.g., see [3, 5]). However, Alice and Bob want to compare elements of the additively shared vector $\vec{c}$. Assume they want to determine if $c_i \geq c_j$ is true, they can compare $a_i - a_j$ (held only by Alice) and $b_j - b_i$ (held only by Bob). This follows from expansion:

$$c_i \geq c_j \Longleftrightarrow a_i - a_j \geq -(b_i - b_j)$$

Using the above mechanism for performing a pairwise comparison of a c_i to a c_j, a naive minimum finding protocol could mimic any standard minimum finding algorithm, but that approach suffers from the same problem as the dynamic programming algorithm: Alice and Bob would learn (at least) the index of the minimum element by observing the results of the comparisons. One way to avoid this problem is to do the following before performing the above-mentioned naive minimum finding protocol: To first blind Alice and Bob (before they engage in such a protocol) by effecting a random permutation of each of their two vectors as a random vector is added to Alice's vector and subtracted from Bob's vector. This permutation and random vector are known neither to Alice nor to Bob; observe that, while the resulting two vectors are such that the minimum entry in their sum is the same as for the sum of the original vectors, it now no longer matters if Alice and Bob know where the minimum occurs because they have no way of relating that position to the index that corresponds to it in their original vectors (and therefore using the naive minimum finding protocol becomes acceptable).

The question that remains is how to achieve this "permuting and additive blinding". This is described next.

Blind-and-Permute Protocol

It is enough to describe how the "permuting and additive blinding" is done when Bob is to know both the permutation and random vector used, because by doing it another time with the roles of Alice and Bob reversed (that is, Alice is to know, but not Bob) we achieve the desired result (of neither of them knowing the overall permutation and random vector that were used). This is because the overall permutation would then be the composition of two permutations each of which is known to only one of the two parties, and the random vector is the sum of two vectors each of which is known to only one of the two parties. The following describes how the "permuting and additive blinding" is done when Bob knows both the permutation and random vector used, but Alice doesn't.

Alice has a vector $\vec{a} = (a_1, \ldots, a_l)$ and Bob wants to permute the elements of Alice's vector after adding to it a random vector (of course he also similarly permutes his own vector after subtracting from it the same random vector, thus maintaining the value of the minimum entry in the sum of their two vectors). Observe that, without the addition of the random vector, Alice could infer the permutation. The protocol is:

1. Alice generates a public and private key pair for a homomorphic semantically-secure public key system and sends the public key to Bob. In what follows $E(\cdot)$ denotes encryption with Alice's public key, and $D(\cdot)$ decryption with Alice's private key.

2. Alice encrypts each entry $(a_1, \ldots, a_l)$ using her public key and sends $\vec{a'} = (E(a_1), \ldots, E(a_l))$ to Bob.

3. Bob generates a random vector $\vec{r} = (r_1, \ldots, r_l)$ (the r_i's are random and both positive and negative). He then computes $\theta_i = a'_i \cdot E(r_i) = E(a_i + r_i)$, for $i = 1, \ldots, l$.

4. Bob permutes, according to a (randomly chosen) permutation π_B, the order of the entries of the $\vec{\theta}$ vector he computed in the previous step. Let $\vec{a''}$ denote the permuted version of the vector $\vec{\theta}$. He sends $\vec{a''}$ to Alice who decrypts its entries and obtains the set of l values of the form $a_i + r_i$ in a permuted order according to π_B (she knows neither π_B nor $\vec{r}$).

5. Bob computes $\vec{b'} = \vec{b} - \vec{r}$ and then obtains $\vec{b''}$ by permuting the entries of $\vec{b'}$ according to π_B.

As mentioned earlier, Alice and Bob must then repeat the above starting with $\vec{a''}$ and $\vec{b''}$ but with the roles of Alice and Bob interchanged. After this is done, they end up with vectors whose sum has the same minimum entry as in the sum of their original vectors, but now it is safe for them to run a naive minimum finding protocol to compute an α_A and an α_B whose sum is $\min_{1\leq i\leq l} c_i$. Although they could at that point exchange α_A and α_B and both would learn the minimum value of the c_i's, in all uses of this minimum-finding scheme by our edit distance protocol no such exchange will take place (i.e., we will deliberately choose to keep the minimum value additively split between Alice and Bob).

3.5 Minimum Finding Protocol for Non-Split Data

In this case Alice has a vector $\vec{a} = (a_1, \dots, a_l)$ and Bob has a vector $\vec{b} = (b_1, \dots, b_{l'})$, and the goal is to compute $\gamma = \min(\min_{1\leq i\leq l} a_i, \min_{1\leq i\leq l'} b_i)$ but without either side actually knowing this γ: Rather, Alice ends up with a γ_A and Bob ends up with a γ_B such that $\gamma = \gamma_A + \gamma_B$. This is achieved as follows:

1. Alice creates $\vec{a'}$ of length 2: $\vec{a'} = (\min_{1\leq i\leq l} a_i, 0)$. Similarly, Bob creates the vector $\vec{b'}$ of length 2: $\vec{b'} = (0, \min_{1\leq i\leq l} b_i)$
2. They run the minimum finding protocol for split data (as described in the previous section). Alice ends up with a scalar γ_A and Bob with a scalar γ_B whose sum is $\min(\min_{1\leq i\leq l} a_i, \min_{1\leq i\leq l} b_i)$.

4. EDIT DISTANCE PROTOCOL

We now "put the pieces together" and give the overall protocol. Recall that Alice is supposed to hold matrix M_A and Bob to hold M_B, with $M = M_A + M_B$. They want to implicitly compute each element $M(i,j)$ as in the recursive edit distance formula, by suitably updating their own private M_A and M_B. Of course when they are done computing M, they could exchange $M_A(n,m)$ and $M_B(n,m)$ and obtain the edit distance $M(n,m) = M_A(n,m) + M_B(n,m)$. (If only Alice is supposed to know the answer then she simply refrains from sending $M_A(n,m)$ to Bob.)

We begin with the case of arbitrary $I(a)$'s and $D(a)$'s, but where $S(a,b) = |a-b|$. This then serves to solve the important practical case of unit insertion/deletion cost and forbidden substitutions (i.e., $S(a,b)$ is 0 if $a = b$ and $+\infty$ otherwise). Finally, the general case of arbitrary $I(a)$, $D(b)$, $S(a,b)$ is considered and a protocol is given for it that is asymptotically as good as the above two special cases, but that is less practical because of its use of oblivious transfer as a subroutine.

4.1 The Case $S(a,b) = |a-b|$

Initialization of M_A and M_B

M should be initialized the same way as in the standard algorithm, and we should maintain the property that $M = M_A + M_B$. Note also, that neither Alice nor Bob ever sends any element of M_A or M_B, respectively in clear-text to the other party after the initialization, until completion of the protocol and above-mentioned exchange of results. The following initializes the M_A and M_B matrices:

- Alice sets $M_A(0,j) = 0$ for $0 \leq j \leq m$, she sets $M_A(i,0) = \sum_{k=1}^{i} D(\lambda_k)$ for $1 \leq i \leq n$.
- Bob sets $M_B(i,0) = 0$ for $0 \leq i \leq n$, he sets $M_B(0,j) = \sum_{k=1}^{j} I(\mu_k)$.

Note that this does implicitly initialize $M(i,j)$ in the correct way, because it results in

- $M_A(0,0) + M_B(0,0) = 0$,
- $M_A(0,j) + M_B(0,j) = \sum_{k=1}^{j} I(\mu_j)$ for $1 \leq j \leq m$,
- $M_A(i,0) + M_B(i,0) = \sum_{k=1}^{i} D(\lambda_i)$ for $1 \leq i \leq n$.

Note: There is no need to initialize $M_A(i,j)$ and $M_B(i,j)$ for positive i and j, because their final value is computed before they are ever used in a computation.

Mimicking a step of the dynamic program

The following protocol describes how an $M(i,j)$ computation is done by Alice and Bob, i.e., how they modify their respective $M_A(i,j)$ and $M_B(i,j)$, thus implicitly computing the final $M(i,j)$ without either of them knowing which update was performed.

1. Alice and Bob use the minimum finding protocol for non-split data to compute γ'_A and γ'_B such that $\gamma'_A + \gamma'_B = \min(\lambda_i, \mu_j)$. Then they use a version of that protocol for computing the max rather than the min, to obtain γ''_A and γ''_B such that $\gamma''_A + \gamma''_B = \max(\lambda_i, \mu_j)$. Alice then obtains $\gamma_A = \gamma''_A - \gamma'_A$, and Bob obtains $\gamma_B = \gamma''_B - \gamma'_B$. The crucial observation is that $\gamma_A + \gamma_B = \max(\lambda_i, \mu_j) - \min(\lambda_i, \mu_j) = |\lambda_i - \mu_j| = S(\lambda_i, \mu_j)$.
2. Alice then forms $v_A = M_A(i-1, j-1) + \gamma_A$ and Bob forms $v_B = M_B(i-1, j-1) + \gamma_B$. Observe that $v_A + v_B = M(i-1, j-1) + S(\lambda_i, \mu_j)$, which is one of the three quantities involved in the update step for $M(i,j)$ in the dynamic program.
3. Alice forms $u_A = M_A(i-1, j) + D(\lambda_i)$, Bob forms $u_B = M_B(i-1, j)$. Observe that $u_A + u_B = M(i-1, j) + D(\lambda_i)$, which is another one of the three quantities involved in the update step for $M(i,j)$ in the dynamic program.
4. Alice forms $w_A = M_A(i, j-1)$, Bob forms $w_B = M_B(i, j-1) + I(\mu_j)$. Observe that $w_A + w_B = M(i, j-1) + I(\mu_j)$, which is the last of the three quantities involved in the update step for $M(i,j)$ in the dynamic program.
5. Alice and Bob use the minimum finding protocol for split data on their respective vectors (v_A, u_A, w_A) and (v_B, u_B, w_B). As a result, Alice gets an x_A and Bob gets an x_B whose sum $x_A + x_B$ is
$$x_A + x_B = \min(v_A + v_B, u_A + u_B, w_A + w_B) =$$
$$= \min \begin{pmatrix} M(i-1, j-1) + S(\lambda_i, \mu_j) \\ M(i-1, j) + D(\lambda_i) \\ M(i, j-1) + I(\mu_j) \end{pmatrix}.$$
6. Alice replaces $M_A(i,j)$ in her M_A by x_A and Bob replaces $M_B(i,j)$ by x_B.

4.2 The Case of Unit Insertion and Deletion Costs and Forbidden Substitutions

Forbidden substitutions means that $S(a,b)$ is $+\infty$ unless $a = b$ (in which case it is zero because it is a "do nothing" operation). Of course a substitution is useless if its cost is 2 or more (because one might as well achieve the same effect with a deletion followed by an insertion). Let the alphabet be $\Sigma = \{1, \ldots, \sigma\}$. This could be a known fixed set of symbols (e.g., in biology $\Sigma = \{A, C, T, G\}$), or the domain of a hash function that maps the potentially infinite alphabet into a finite domain.

The protocol is then:

1. For $i = \sigma, \ldots, 1$ in turn, each side replaces, in their input string, every occurrence of symbol i by the symbol $2i$. Effectively the alphabet becomes $\{2, 4, 6, \ldots, 2\sigma\}$.

2. They run the protocol given in the previous section for the case of $S(a, b) = |a - b|$, using a unit cost for every insertion and every deletion.

The reason it works is that, after the change of alphabet, $S(a, b)$ is zero if $a = b$ and 2 or more if $a \neq b$, i.e., it is as if $S(a, b) = +\infty$ if $a \neq b$ (recall that a substitution is useless if its cost is 2 or more, because one can achieve the same effect with a deletion followed by an insertion).

4.3 The General Case: Arbitrary $I(a)$, $D(b)$, $S(a, b)$

Whereas the above protocols for the two special cases do not use oblivious transfer (which we review below when we do use it), this subsection for the most general case does use it, and the resulting protocol is therefore less practical than for the two special cases (although it has same the asymptotic communication and computation complexity).

The necessary modification is in the first step of the protocol, in Sub-section 4.1, titled "mimicking a step of the dynamic program". Specifically, we give below a new version of Step 1, whose aim is still to produce a v_A with Alice and a v_B with Bob such that $v_A + v_B = M(i-1, j-1) + S(\lambda_i, \mu_j)$. We assume in what follows that the alphabet is $\Sigma = \{1, \ldots, \sigma\}$, i.e., all symbols λ_i and μ_j come from Σ.

1. Alice selects a random r_A and sets $v_A = M_A(i-1, j-1) - r_A$. The task is now for Bob to get a v_B that equals $M_B(i-1, j-1) + S(\lambda_i, \mu_j) + r_A$. This is done as follows:

   (a) Alice produces the σ items $\alpha_1, \ldots, \alpha_\sigma$ where $\alpha_t = S(\lambda_i, t) + r_A$.

   (b) Bob uses a 1-out-of-σ oblivious transfer protocol to obtain α_{μ_j} from Alice without revealing to Alice which of her α_t's he received (see [16] for detailed oblivious transfer protocols).

   (c) Bob sets $v_B = M_B(i-1, j-1) + \alpha_{\mu_j}$, which equals $M_B(i-1, j-1) + S(\lambda_i, \mu_j) + r_A$.

Note that, at the end of Step 1, $v_A + v_B$ equals $M_A(i-1, j-1) - r_A + M_B(i-1, j-1) + S(\lambda_i, \mu_j) + r_A = M(i-1, j-1) + S(\lambda_i, \mu_j)$ as required.

5. CONCLUDING REMARKS

We gave an efficient protocol for sequence comparisons of the string-edit kind, such that neither party reveals anything about their private sequence to the other party. This is but a first step in an area of activity that is compelling in its potential usefulness; in addition to the already-mentioned scenarios of preserving a corporation's private bio-sequences from competitors, and preserving the privacy of individuals, many other questions come up such as "can Alice perform data mining on Bob's biological database without revealing Alice's queries to Bob and without revealing Bob's database to Alice (other than what Alice can infer from the query responses)" ? Which other bioinformatics problems have efficient protocols like the one presented in this paper ?

6. REFERENCES

[1] A.V. Aho, D.S. Hirschberg and J.D. Ullman. Bounds on the Complexity of the Longest Common Subsequence Problem, *Journal of the ACM* 23, 1, pp.1–12 (1976).

[2] J. Benaloh. Dense Probabilistic Encryption, *Proceedings of the Workshop on Selected Areas of Cryptography*, pp.120-128 (1994).

[3] C. Cachin. Efficient Private Bidding and Auctions with an Oblivious Third Party. *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pp.120-127 (1999).

[4] W. Du and M.J. Atallah. Protocols for Secure Remote Database Access with Approximate Matching. *Proceedings of the 1st ACM Workshop on Security and Privacy in E-Commerce* (2000).

[5] M. Fischlin. A Cost-Effective Pay-Per-Multiplication Comparison Method for Millionaires. RSA Security 2001 Cryptographer's Track. *Lecture Notes in Computer Science* 2020, pp.457-471 (2001).

[6] O. Goldreich. Secure Multi-party Computation (working draft). Available at *http://www.wisdom.weizmann.ac.il/∼oded/pp.html* (2001).

[7] S. Goldwasser and S. Micali. Probabilistic Encryption. *Journal of Computer and System Sciences* 28, 2, pp.270-299 (1984).

[8] G. Landau and U. Vishkin. Introducing Efficient Parallelism into Approximate String Matching and a new Serial Algorithm, *Proceedings of the 18-th ACM STOC,* pp.220–230 (1986).

[9] H. M. Martinez (ed.) Mathematical and Computational Problems in the Analysis of Molecular Sequences, *Bulletin of Mathematical Biology* (Special Issue Honoring M. O. Dayhoff), 46, 4 (1984).

[10] W. J. Masek and M. S. Paterson. A Faster Algorithm Computing String Edit Distances, *Journal of Computer and System Science* 20, pp.18–31 (1980).

[11] D. Naccache and J. Stern. A New Cryptosystem based on Higher Residues. *Proceedings of the ACM Conference on Computer and Communications Security* 5, pp.59-66 (1998).

[12] S. B. Needleman and C.D. Wunsch. A General Method Applicable to the Search for Similarities in the Amino-acid Sequence of Two Proteins, *Journal of Molecular Biology* 48, pp.443–453 (1973).

[13] T. Okamoto and S. Uchiyama. A New Public-Key Cryptosystem as Secure as Factoring. Eurocrypt'98 *Lecture Notes in Computer Science* 1403, pp.308-318 (1998).

[14] D. Sankoff. Matching Sequences Under Deletion-insertion Constraints, *Proceedings of the National Academy of Sciences of the U.S.A.* 69, pp.4–6 (1972).

[15] D. Sankoff and J. B. Kruskal (eds.). Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison, *Addison-Wesley*, Reading, PA (1983).

[16] Bruce Schneier. Applied Cryptography : Protocols, Algorithms, and Source Code in C (Second Edition), *John Wiley & Sons*, New York, NY (1996).

[17] P.H. Sellers. An Algorithm for the Distance between two Finite Sequences, *J. of Combinatorial Theory* 16, pp.253–258 (1974).

[18] P.H. Sellers. The Theory and Computation of Evolutionary Distance: Pattern Recognition, *Journal of Algorithms* 1, pp.359–373 (1980).

[19] E. Ukkonen. Finding Approximate Patterns in Strings, *Journal of Algorithms* 6, pp.132–137 (1985).

[20] R. A. Wagner and M. J. Fischer. The String to String Correction Problem, *Journal of the ACM* 21,1, pp.168–173 (1974).

[21] C.K. Wong and A.K. Chandra. Bounds for the String Editing Problem, *Journal of the ACM* 23, 1, pp.13–16 (1976).

[22] A. Yao. Protocols for Secure Computations. *Proceedings of the Annual IEEE Symposium on Foundations of Computer Science* 23, pp.160-164 (1982).