Secure and Efficient Outsourcing of Sequence Comparisons[*]

Marina Blanton[1], Mikhail J. Atallah[2], Keith B. Frikken[3], and Qutaibah Malluhi[4]

[1] Department of Computer Science and Engineering, University of Notre Dame
[2] Department of Computer Science, Purdue University
[3] Computer Science and Software Engineering, Miami University
[4] Computer Science and Engineering Department, Qatar University

Abstract. In this work we treat the problem of secure outsourcing of sequence comparisons by a client to remote servers. The sequence comparison problem, given two strings λ and μ of respective lengths n and m, consists of finding a minimum-cost sequence of insertions, deletions, and substitutions (also called an *edit script*) that transform λ into μ. In our framework a client owns strings λ and μ and outsources the computation to two remote servers without revealing to them information about either the input strings or the output sequence. Our solution is non-interactive for the client (who only sends information about the inputs and receives the output) and the client's work is linear in its input/output. The servers' performance is $O(\sigma mn)$ computation (which is optimal) and communication, where σ is the alphabet size, and the solution is designed to work when the servers have only $O(\sigma(m+n))$ memory. By utilizing garbled circuit evaluation techniques in a novel way, we completely avoid the use of public-key cryptography, which makes our solution efficient in practice.

1 Introduction

Design and development of secure outsourcing techniques of various functionalities to untrusted servers are getting growing attention in the research community. The rapid growth in availability of cloud services, makes such services attractive for clients with limited computing or storage resources who are unwilling or unable to procure and maintain their own computing infrastructure. Security and privacy considerations, however, stand on the way of harnessing the benefits of cloud computing to the fullest extent and prevent clients from placing their private or sensitive data on the cloud. This is the problem that secure outsourcing techniques aim to address.

This work develops efficient techniques for secure outsourcing of a specific type of computation, namely sequence comparisons. Secure computation and outsourcing of sequence comparisons, in particular for genomic sequences, has been a subject of prior research. The results, e.g., include [1–3, 18, 19, 12, 9, 6, 11, 5], which securely implement computation of the edit distance, finite automata evaluation, the Smith-Waterman

[*] Portions of this work were supported by NSF Grants CNS-0915436, CNS-0913875, CNS-0915843, and CCF-0939370; by an NPRP grant from the Qatar National Research Fund; by Grant FA9550-09-1-0223 from AFOSR; and by sponsors of the CERIAS center.

and other algorithms. Because individual DNA and protein sequences commonly used in such comparisons are highly sensitive and vulnerable to re-identification even when anonymized, the need for techniques that allow such sequences to be privately processed has been recognized and is reflected by the list of available publications above. Furthermore, given the large lengths of such sequences, it is not surprising that there is an increasing need for such computation to be outsourced by resource limited clients. These outsourcing techniques should enable the desired computation without revealing any information about the sequences to the parties carrying out the computation.

Techniques for securely computing the edit distance based on dynamic programming have been studied in [1, 12, 11]. The work [2, 3] is the only one we are aware of that treats the problem of secure outsourcing of the edit distance and [3] is the only work that treats the computation of the edit script (defined as a minimum-cost sequence of insertions, deletions, and substitutions that transform one input string λ into the other input string μ). An edit script contains important information about the types of differences that cannot always be deduced from the edit distance alone. For that reason, we revisit the problem of secure outsourcing of the edit distance and the corresponding edit script computation and improve the performance of known results.

It is well known that computing the edit distance (or the edit script) of two strings λ and μ of size n and m, respectively, requires $O(mn)$ work. Because n and m are often large in genomic computations, the need to reduce the memory footprint of secure sequence comparisons was recognized in prior literature. In particular, the edit distance can be computed one row or one column of the $m \times n$ matrix at a time, which uses only $O(m+n)$ memory. This is the approach taken in [3] based on homomorphic encryption, and the publications that use garbled circuit evaluation [12, 11] also partition the circuit into sub-circuits, so that the memory requirement of $O(m + n)$ can be achieved.

Unfortunately, the above partitioning approach does not work when the computation consists of producing an edit script (rather than just the edit distance) while keeping the memory requirement at $O(m + n)$. Furthermore, the only known result for securely computing an edit script with the linear memory requirement for the servers carrying out the computation requires them to perform $O(mn \min(m, n))$ work with the same amount of communication [3]. In this work, we substantially improve the performance of the existing secure edit script outsourcing techniques to require the servers to perform only $O(mn)$ work with the same $O(m + n)$ memory requirement for the servers. This also implies that when the servers have $O(m + n)$ memory, the round complexity of the solution improves from $O((\min(m, n)^2)$ in [3] to $O(\min(m, n))$ in this work (we note that the number of rounds in this work is primarily bounded by the ratio of the overall amount of communication and the amount of available memory, while it is fixed at $O((\min(m, n)^2)$ in [3]).

To emphasize that the memory requirements of $O(mn)$, or more generally $O(\sigma mn)$, where σ is the alphabet size, are unacceptable even when the computation is outsourced to resourceful servers, consider, for example, a server with 32GB of RAM. Even when σ is small (e.g., $\sigma = 4$), an efficient implementation based on garbled circuits or homomorphic encryption will allow the servers to process only strings a couple of thousands characters long. As clients would be more inclined to outsource tasks of large rather than small size, this imposes strict limits on the practicality of such an approach.

Besides the obvious complexity improvements, our solution has additional advantages. Similar to [3], our solution assumes that a client outsources its computation to two non-colluding computation servers, but unlike [3], no homomorphic encryption is used. In fact, our solution completely avoids the use of public-key cryptography by utilizing garbled circuit techniques in a novel way. To the best of our knowledge, this is the first time secure two-party computation or outsourcing techniques are realized without reliance on any public-key operations (e.g., the solutions in [12, 11] have to invoke Oblivious Transfer (OT) protocols). This gives us the fastest general secure outsourcing techniques, which are of independent interest.

Our solution is non-interactive for the client, who only sends information about its inputs to the servers and receives the outcome of the computation from which it reconstructs the output. The client's communication and computation is therefore $O(m+n)$.

Lastly, our solution works for any alphabet Σ of size σ, from which strings λ and μ are drawn. Because σ may not be treated as constant, we explicitly include it in our analysis. In particular, the servers' space requirements are $O(\sigma(m+n))$, their computation and communication are $O(\sigma mn)$, and the client's work and communication are $O(\sigma(m+n))$ (note that prior results also have the same factor σ in their complexities).

As noted above, our improvements make the same assumption of non-colluding servers as the prior work that improve upon. A natural question that one might ask is how viable such an assumption is. The practical viability of using non-colluding servers has been well demonstrated, for instance, by the Sharemind system [8] and the company that develops it, where three non-colluding servers are used (we only use two). One possible instantiation of our solution would be to use two servers, each from a different service provider. Collusion of both servers would require corruption of both service providers, which is unlikely in practice.

Organization. We first state the problem and provide background review in section 2. Section 3 provides an overview of the techniques that allow us to achieve the claimed result. Section 4 describes an oblivious algorithm for the edit script problem which is suitable for secure computation with $O(\sigma(n+m))$ memory requirements using only $O(\sigma mn)$ overall work. Our preliminary protocol for secure outsourcing is given in section 5. While that solution already provides significant complexity improvements over prior work and does not use public-key operations, it requires the client to participate in $O(\log(\min(m,n)))$ rounds of the protocol. This disadvantage is mitigated in section 6, where we describe our final result. Lastly, section 7 concludes this work.

2 Preliminaries

Problem statement. In this work we treat the problem of secure outsourcing of the edit distance and the corresponding edit script computation by a client C for any strings $\lambda = \lambda_1 \ldots \lambda_n$ and $\mu = \mu_1 \ldots \mu_m$ over alphabet $\Sigma = \{1, \ldots, \sigma\}$ to two computational servers S_1 and S_2. In its general form considered here, the sequence comparisons problem requires quadratic work [21].

In our outsourcing context, it is required that C performs only work linear in the size of its inputs, with the super-linear work done by the remote servers. Furthermore, the security requirement is such that neither S_1 nor S_2 learns anything about the client's inputs or output other than the lengths of the input stringsand the alphabet size (i.e., the servers do not learn anything other than the problem size).

More formally, we assume that S_1 and S_2 do not collude and if they are semi-honest, they follow the computation as prescribed but might attempt to learn additional information from the messages that they observe. Security in this case is guaranteed if both S_1's and S_2's views can be simulated by a simulator with no access to either C's inputs or output other than the parameters n, m, and σ and such simulation is indistinguishable from the real protocol execution. This is a standard definition that can be found, e.g., in [10]. While our solution can be made secure against malicious parties, the semi-honest model can be generally acceptable for the following reasons:

1. With computation outsourcing, the current practice is that a cloud service provider is bound with a client by a contractual agreement, violation of which exposes the provider to loss of reputation and client's business.
2. Cloud service providers themselves would like to minimize the amount of sensitive information to which they are exposed during outsourced computation or storage for legal liability issues.

Techniques secure against semi-honest participants are normally used as a foundation for designing efficient protocols secure against stronger adversaries. Generic techniques for modifying the garbled circuit techniques to enable security against covert or fully malicious participants are known (see, e.g., [4, 15, 16]). Furthermore, the standard garbled circuit techniques, as used in this work, already offer protection against one party, namely, malicious circuit evaluator. The specifics of our setting, however, enable us to design an effective mechanism for detecting and eliminating malicious behavior at low cost. Because the techniques we use are resilient to misbehavior of one of the parties, we can run the solution twice, with the roles of S_1 and S_2 swapped on the second run. When the client obtains two results that disagree, it will know that one of the servers did not comply with its prescribed behavior. As in our protocols neither server learns any outputs, creation and evaluation of an incorrect circuit does not pose security risks to the client. This means that the cost of the solution in the malicious model is twice the cost of the solution in the semi-honest model.

Review of edit distance via dynamic programming. We briefly review the standard dynamic programming algorithm for the edit distance, using the same notation and terminology as in [3]. Let $M(i,j)$, for $0 \le i \le m$ and $0 \le j \le n$, be the minimum cost of transforming the prefix of λ of length j into the prefix of μ of length i, i.e., the cost of transforming $\lambda_1 \dots \lambda_j$ into $\mu_1 \dots \mu_i$. Then $M(0,0) = 0$, $M(0,j) = \sum_{k=1}^{j} D(\lambda_k)$ for $1 \le j \le n$ and $M(i,0) = \sum_{k=1}^{i} I(\mu_k)$ for $1 \le i \le m$. Furthermore, for all $1 \le i \le m$ and $1 \le j \le n$ we have that

$$M(i,j) = \min \begin{cases} M(i-1,j-1) + S(\lambda_j,\mu_i) \\ M(i-1,j) + I(\mu_i) \\ M(i,j-1) + D(\lambda_j) \end{cases}$$

where $S(\lambda_j,\mu_i)$ denotes the cost of substituting character λ_j with character μ_i, $D(\lambda_j)$ denotes the cost of deleting λ_j, and $I(\mu_i)$ denotes the cost of inserting μ_i. 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 three $M(i-1,j-1)$, $M(i-1,j)$, and $M(i,j-1)$ are involved in the computation of the final value of $M(i,j)$.

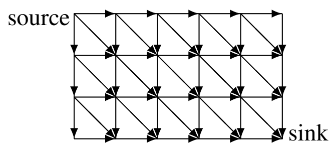

Fig. 1. Example of a 3×5 grid DAG.

Our solution works when $S : \Sigma \times \Sigma \to \mathbb{N}$, $I : \Sigma \to \mathbb{N}$, and $D : \Sigma \to \mathbb{N}$ are arbitrary functions that are implemented using table lookups as well as when they are special cases of such functions.

Grid graph view of the problem. The interdependencies among the entries of the M-matrix induce an $(m+1) \times (n+1)$ *grid* directed acyclic graph (DAG) associated with the string editing problem. It is easy to see (and well-known) that the string editing problem can be viewed as a shortest-path problem on a grid DAG, which is *implicitly* described by the two input strings and the cost tables (otherwise there is no hope of achieving the linear-space performance we seek). We say that an $\ell_1 \times \ell_2$ grid DAG is a directed acyclic graph whose vertices are the $\ell_1 \ell_2$ points of an $\ell_1 \times \ell_2$ grid, and such that the only edges from grid point (i, j) are to grid points $(i, j+1)$, $(i+1, j)$, and $(i+1, j+1)$. Figure 1 shows an example of a grid DAG and our convention of drawing the points such that point (i, j) is at the ith row from the top and the jth column from the left. Note that the top-left point is $(0, 0)$ and has no edge entering it (i.e., is a *source*), and that the bottom-right point is (m, n) and has no edge leaving it (i.e., is a *sink*).

An $(m+1) \times (n+1)$ grid DAG G is associated with the string editing problem in the natural way: The vertices of G are in one-to-one correspondence with the entries of the M-matrix, and the *cost* of an edge from vertex (k, ℓ) to (i, j) is equal to $D(\lambda_j)$ if $k = i$ and $\ell = j - 1$, to $I(\mu_i)$ if $k = i - 1$ and $\ell = j$, and to $S(\lambda_j, \mu_i)$ if $k = i - 1$ and $\ell = j - 1$. We restrict our attention to edit paths which do no obviously inefficient moves (such as inserting then deleting the same symbol) and thus only consider edit scripts that apply at most one edit operation to a given symbol. Such edit scripts that transform λ into μ or vice versa are in one-to-one correspondence to the weighted paths of G that originate at the source (which corresponds to $M(0, 0)$) and end at the sink (which corresponds to $M(m, n)$).

Garbled circuit evaluation. Our solution uses techniques based on Yao's two-party garbled circuit evaluation originated in [22]. Garbled circuit evaluation allows two parties to securely evaluate any function represented as a Boolean circuit. The basic idea is that, given a circuit composed of gates, one party P_1 creates a garbled circuit by assigning to each wire i two randomly chosen labels or keys $\ell_0^{(i)}$ and $\ell_1^{(i)}$, where $\ell_b^{(i)}$ encodes bit b. P_1 also encodes gate information in a way that given keys corresponding to the input wires (encoding specific inputs), the key corresponding to the output of the gate on those inputs can be recovered. This is often achieved by representing each gate as a table of encrypted values, where, e.g., for a binary gate g with input wires i, j and output wire k, the table consists of four values of the form $\mathsf{Enc}_{\ell_{b_i}^{(i)}, \ell_{b_j}^{(j)}}(\ell_{g(b_i, b_j)}^{(k)})$.

The second party, P_2, evaluates the circuit using keys corresponding to inputs of both P_1 and P_2 (without learning anything in the process). That is, P_2 directly obtains keys corresponding to P_1's input bits from P_1 and engages in the OT protocol to obtain keys corresponding to P_2's input bits. Garbled circuit evaluation consists of processing the gates in topological order, during which one entry of each gate's table is decrypted allowing P_2 to learn the output wire's key. Security relies on the fact that P_2 does not have a correspondence between the labels it decrypts and the bits that they represent. At the end, the result of the computation can be recovered by linking the output labels to the bits which they encode (e.g., by having P_1 send all output wire labels and their meaning to P_2). Recent literature [14, 17, 13] provides optimizations that significantly reduce computation and communication overhead associated with garbled circuits.

Prior results. Using the fact that computing a row of the matrix depends only on entries from its current and previous rows, computing the edit distance (not path) is done with S_1 and S_2 in [2, 3] using $O(\sigma(m+n))$ space and $O(\sigma mn)$ time in $O(\min(m, n))$ rounds. Similarly, securely computing the edit distance in the two-party setting using garbled circuit evaluation is done in [12, 11] by partitioning the overall computation into multiple sub-circuits or rounds to achieve the same result. Computing the path itself took in [3] an extra factor of $\min(m, n)$ work and rounds. One of our main goals is therefore removing that extra factor for the path (as opposed to the distance) computation. Our solution is also more flexible in terms of its round complexity even for the distance computation. In addition to asymptotic complexity savings, the fact that our solution does not use expensive public-key operation makes it significantly more efficient (even for the distance computation) than [2, 3] which made an extensive use of homomorphic encryption. Furthermore, our technique for removing the need for public-key operations is of independent interest for secure computation outsourcing.

Lastly, while [11] implements the idea of partitioning a circuit into sub-circuits, which is used in this work as well, and provides circuit optimizations for computing the edit distance, that work is complementary to ours. Because a distance protocol is used as a subroutine in our solution, these circuit optimization techniques can be used with our result to create a fast circuit for computing the elements of the M matrix.

3 Overview of the Solution

Before describing our solution in detail, we provide an intuition behind it. First, notice that if the amount of available memory is $O(mn)$, it is easy to compute the edit script. That is, first compute all elements of the matrix M. Then, starting from $M(m, n)$, follow the link to either $M(i-1, j)$, $M(i-1, j-1)$, or $M(i, j-1)$ that produced the current value of $M(i, j)$ (breaking ties arbitrarily), until the process terminates at $M(0, 0)$. The produced path corresponds to the desired edit script that the client would like to learn. This approach, however, does not work if the amount of available memory is $o(mn)$ because the value stored at any given $M(i, j)$ might be necessary for reconstructing the path. Because our goal is to use only $O(\sigma(m+n))$ memory, the servers will not be able to maintain all necessary information and a different approach is needed.

To address this problem without increasing the cost of the overall computation beyond $O(\sigma mn)$, we can use a recursive solution, which works as follows: in the first round, instead of computing all elements of M as described earlier, we compute the elements in the "top half" of the matrix as before and also compute the elements of the "bottom half" of the matrix in the reverse direction starting from $M(m,n)$ (see section 4 for detail). Then for each element $M(m/2,j)$ of the middle row we add the distances computed from the top and from the bottom and determine the position of the element with the minimum sum. In section 4 we denote this element by $M(m/2,\theta(m/2))$. Because we know that the computed element has to lie on a path that results in the minimum edit distance, to determine other parts of this path, we can safely disregard all cells from the top half that lie to the right of $M(m/2,\theta(m/2))$ and all cells from the bottom half that lie to the left of $M(m/2,\theta(m/2))$. We then recursively apply this algorithm to the remaining portions of the matrix (which together contain only a half of the elements of M) which allows us to reconstruct all points of the path. While this approach doubles the amount of computation (i.e., the work is $\leq 2mn$ compared to the original mn), it is suitable for our situation when the amount of available space is only linear in m and n.

Now notice that this solution works in a traditional setting, but in our case revealing the position of the minimum element $M(m/2,\theta(m/2))$ (which is necessary for determining what parts of the matrix should be discarded for the next round), leaks important information about the edit path to the computational servers and violates security requirements. Our solution is to recurse on sub-problems of slightly larger size without revealing information about the value of $\theta(m/2)$. In particular, we form two sub-problems of size 1/2 and 1/4 of the original, where the 1/2 sub-problem consists of the top (resp. bottom) half and the 1/4 sub-problem consists of the right bottom (resp. top left) quadrant when $\theta(m/2) \geq n/2$ (resp. $\theta(m/2) < n/2$). This ensures that the asymptotic complexity of the solution does not change (the work is $\leq 4mn$), while hiding information about the path. This process, however, requires care because the strings that form the sub-problems of fixed size must be padded based on the value of $\theta(m/2)$. That is, we need to ensure that the way the strings are padded (as a prefix or suffix of an existing string) should not affect the overall result. We achieve this by extending the alphabet with a new character with carefully chosen insertion, substitution, and deletion costs so as to take a certain path within the matrix and not alter the edit distance.

The last remaining bit that we want the computational servers to prevent from learning is whether the subtask of size 1/2 corresponds to the top or bottom portion of the problem (which, once again, leaks information about the edit path). This is achieved by always having the sub-tasks of different sizes in a fixed order and obliviously assigning the correct portion of the grid to a sub-task. This allows us to obtain a solution that can be safely outsourced to the computational servers and meets their space requirements.

Having arrived at the oblivious algorithm for computing an edit script with $O(\sigma(m+n))$ memory and $O(mn)$ overall work, we now need to see how this computation can be securely outsourced. Recall that our solution relies on garbled circuit evaluation which we use in a new way. The first idea that we employ is for the client to produce garbled circuit's random labels corresponding to the wires of its inputs only (two labels per input bit). The client sends the labels for all wires to S_1, who forms the rest of the circuit for the computation. The client also sends to S_2 one label per wire that corresponds to its input value. Once the circuit is formed, S_2 will be able to evaluate it using the labels. In this case, no OT protocols (or any other public-key operations) are necessary.

Note that this approach is general and by itself would be sufficient to result in a secure outsourcing solution for most types of functions with no public-key cryptography involved at any point in the protocol. For our problem, however, it does not lead to a non-interactive (for the client) solution because after the first round of the computation, the servers will need to contact the client again to obtain the labels for the next round of the computation (since they are not allowed to know what input values or labels are to be reused in the consecutive round). Because the depth of the recursion in our algorithm is $O(\log(\min(m,n)))$, the client has to participate in $O(\log(\min(m,n)))$ interactions with the servers. This forms our preliminary solution in section 5.

To eliminate the client's involvement, we employ the second idea, which consists of the servers using the output wire labels from the current round of the computation as the input wire labels for the sub-problems in the next round. This solution requires a great care to ensure that all input labels for the next round are formed correctly and computed obliviously (inside a garbled circuit) and is described in section 6. We thus obtain our target result in which the solution is non-interactive for the client, the client's work is $O(\sigma(n+m))$, the servers' work is $O(\sigma mn)$, the entire computation can be carried out within $O(\sigma(m+n))$ space, and no public-key operations are used at any point.

4 Enabling the Computation to be Performed Obliviously

As a first step toward building our result, we design an algorithm that allows the computation to be performed in $O(\sigma mn)$ time using $O(\sigma(m+n))$ space. To be suitable for secure outsourcing, the algorithm must be oblivious or data-independent (i.e., it always performs the same sequence of steps regardless of the inputs). This will ensure that no information about the inputs is leaked based on the algorithm itself. We therefore first describe a procedure for such computation and later refine and instantiate it with secure building blocks to obtain the overall solution with the desired performance.

To build our solution, we first need to extend the distance-computation to the computation of an optimal *edit path* (i.e., a minimum-cost sequence of operations on λ that turn it into μ). We adapt the approach of [3] that combines the distance computation algorithm with a backward version of it which we review next.

The backward version of the distance computation. The algorithm mentioned in section 2 is a distance rather than path algorithm. It computes the length of a shortest path from vertex $(0,0)$ to any vertex (i,j) in the grid graph G. We call this the *forward* algorithm and denote its M matrix as M_F where F is a mnemonic for "forward." Let G^R denote the *reverse* of G, i.e., the graph obtained from G by reversing the direction of every edge. Clearly, in G^R vertex (m,n) is the source and vertex $(0,0)$ is the sink, and every v-to-w shortest path in G^R corresponds to a similar shortest path in G *but in the backwards direction* (i.e., w-to-v). We thus use M_B to denote the matrix that is to G^R what matrix M_F was to graph G (B is a mnemonic for "backward"). Then $M_B(i,j)$ denotes the length of a shortest path in G^R from the source of G^R (vertex (m,n)) to vertex (i,j), which is equal to the length of a shortest path in G from (i,j) to (m,n). The edit distance we seek is therefore $M_B(0,0)$ (which is the same as $M_F(m,n)$). Defined in terms of the two input strings, $M_B(i,j)$ is the edit distance from the suffix of λ of length $n-j$, to the suffix of μ of length $m-i$. Therefore computing M_B in an analogous manner to the computation of M_F involves filling in its entries by *decreasing* (rather than increasing) row and column order. An algorithm for M_B follows from any algorithm for M_F, which we thus assume and use in the subsequent description.

Note that $M_F(i, j) + M_B(i, j)$ is the length of a shortest source-to-sink path *constrained to go through vertex* (i, j) and hence might not be the shortest possible source-to-sink path. However, if the shortest source-to-sink path goes though vertex (i, j), then $M_F(i, j) + M_B(i, j)$ is equal to the length of the shortest path. We use M_C to denote $M_F + M_B$ (where C stands for "constrained").

Oblivious edit path computation. We can now describe our oblivious edit path algorithm with the desired bounds. Similar to the structure of computation in [3], we find for each row i of M_C the column $\theta(i)$ of the minimum entry of that row, with ties broken in favor of the rightmost such entry. Note that $M_C(i, \theta(i))$ is the edit distance $M_F(m, n)$. Computing the θ function provides an implicit description of the edit path because:

- If $\theta(i + 1) = \theta(i) = j$, then the edit path "leaves" row i through the vertical edge from vertex (i, j) to vertex $(i+1, j)$. The cost of that edge is that of inserting μ_{i+1}.
- If $\theta(i + 1) = \theta(i) + \delta$, where $\delta > 0$, then the client can "fill in" the portion of the edit path from vertex $(i, \theta(i))$ to $(i + 1, \theta(i) + \delta)$ in $O(\delta)$ time (such a "thin" problem on a $2 \times \delta$ subgrid is trivially solvable in $O(\delta)$ time). The cumulative cost of all such "thin problem solutions" is $O(n)$ because the sum of all such δ's is $\leq n$.

Without loss of generality, let $m \leq n$. For reasons that will become apparent, similar to [3] we introduce a new symbol ϵ that does not occur in Σ and denote $\Sigma_\epsilon = \Sigma \cup \epsilon$. We assign to ϵ an insertion cost of 0, a deletion cost of ∞, and an ∞ cost for any substitution in which it is involved. In practice, ∞ can be set to be $(m + n)$ times the largest cost appearing in the cost tables for Σ (whether it is insertion, deletion, or substitution).

Because given $\theta(0), \ldots, \theta(m)$, C can compute the edit path in linear additional time, we give an algorithm for computing the θ function. It proceeds in $\log m$ rounds, the kth of which consists of 2^{k-1} grid graphs (each described implicitly by two substrings of μ and λ) of respective dimensions $(m/2^{k-1}) \times n_1$, ..., $(m/2^{k-1}) \times n_{2^{k-1}}$, where $\sum_{t=1}^{2^{k-1}} n_t = (3/4)^{k-1} n$ as explained below. The first round proceeds as follows:

1. Run the forward edit distance algorithm to compute row $m/2$ of M_F.
2. Run the backward edit distance algorithm to compute row $m/2$ of M_B.
3. Compute $\theta(m/2)$ as the minimum of $M_C(m/2, j)$ across all $0 \leq j \leq n$.

The two subproblems of round 2 could, *if one were not concerned about information leakage*, be defined by the following two sub-grids: (i) the $(m/2) \times \theta(m/2)$ one that lies to the left and above vertex $(m/2, \theta(m/2))$ and is described implicitly by the strings $\mu_1, \ldots, \mu_{m/2}$ and $\lambda_1, \ldots, \lambda_{\theta(m/2)}$; and (ii) the $m/2 \times (n - \theta(m/2))$ one that lies to the right and below vertex $(m/2, \theta(m/2))$ and is described implicitly by the strings $\mu_{(m/2)+1}, \ldots, \mu_m$ and $\lambda_{\theta(m/2)+1}, \ldots, \lambda_n$. The area of those two subgrids is half the original, but their size would leak the value $\theta(m/2)$ during outsourced computation. We fix this by using, for round 2, subgrids whose size does not depend on $\theta(m/2)$ and yet their combined area is $3/4$ of the original, as described below. In what follows, we assume without loss of generality that $\theta(m/2) \geq n/2$. While in this description it appears that the fact that $\theta(m/2) \geq n/2$ is leaked, in our actual protocol described later this information is not revealed and the execution is fully oblivious.

- The first subgrid is of size $(m/2) \times n$ and is defined by the strings $\mu_1, \ldots, \mu_{m/2}$ and $\lambda_1, \ldots, \lambda_{\theta(m/2)}, \epsilon, \ldots, \epsilon$. The appending of the $n - \theta(m/2)$ symbols of type ϵ

(a) M_F

0	1	2	3	4
1	0	1	2	3
2	1	2	3	2
3	2	1	2	3
4	3	2	1	2

(b) M_B

2	3	4	5	4
3	2	3	4	3
2	1	2	3	2
3	2	1	2	1
4	3	2	1	0

(c) M_C

2	4	6	8	8
4	2	4	6	6
4	2	4	6	4
6	4	2	4	4
8	6	4	2	2

(d) θ

$\theta(0) = 0$
$\theta(1) = 1$
$\theta(2) = 1$
$\theta(3) = 2$
$\theta(4) = 4$

Table 1. Matrices for edit distance between strings AACG and AGAC.

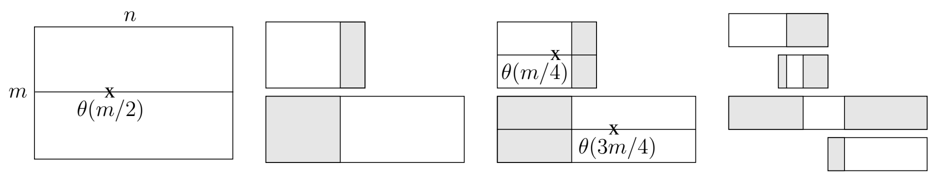

Fig. 2. Illustration of θ function computation.

at the end of the second string hides $\theta(m/2)$ without changing the answer because the edit path for that subproblem has to use the $n - \theta(m/2)$ horizontal edges of 0 cost that link vertex $(m/2, \theta(m/2))$ to the vertex $(m/2, n)$.

- The second subgrid is of size $(m/2) \times (n/2)$ and is defined by the strings $\mu_{(m/2)+1}, \dots, \mu_m$ and $\epsilon, \dots, \epsilon, \lambda_{\theta(m/2)+1}, \dots, \lambda_n$. The pre-pending of the $(n/2) - \theta(m/2)$ ϵ symbols at the beginning of the second string hides $\theta(m/2)$ without changing the answer because the edit path for that subproblem has to use the $(n/2) - \theta(m/2)$ horizontal edges of 0 cost that link vertex $(m/2, n/2)$ to the vertex $(m/2, \theta(m/2))$.

A pair of 3rd-round sub-problems is derived from each 2nd-round subgrid in the same way as above, thus the third round consists of 4 subgrids whose total (combined) number of columns is $9n/16$ (namely, $n/4$, $n/8$, $n/8$, and $n/16$) and the total number of rows is m ($m/4$ rows for each).

Because the total (combined) problem size decreases by a factor of $3/4$ from one round to the next, the overall work of the above algorithm is as claimed: $O(\sigma mn)$. More precisely, the recurrence is $T(m, n) = T(\frac{m}{2}, n) + T(\frac{m}{2}, \frac{n}{2}) + \alpha\sigma mn$, and by easy induction it can be shown that $T(m, n) \leq 4\alpha\sigma mn$. Space is linear because each invocation of the edit-distance protocol uses linear space.

To clarify many of the above notions, we give a small example using strings AACG and AGAC with insertion and deletion costs of 1, and substitution cost of 0 for equal characters and 2 for non-equal characters. The 5×5 DAG for this example is like the one in Figure 1, and Table 4 provides matrices M_F, M_B, M_C and the values for θ. Notice that $M_B(0, 0) = M_F(4, 4) = 2$, which is the edit distance between these two strings. Further, note that the shortest path goes through $M(i, \theta(i))$ for any row i.

Also, Figure 2 demonstrates our algorithm for edit path computation, where at each iteration a given (sub-)grid is partitioned into two subgrids of $1/2$ and $1/4$ of its original size and the remaining $1/4$ is removed. In the figure, shaded areas correspond to string padding with character ϵ. In the figure, because $\theta(m/2) < n/2$, the top subgrid has size 1/4 and the bottom subgrid has size $1/2$. In the second round, $\theta(m/4) > n/4$

and therefore the top subgrid is further partitioned into subgrids of size $1/8$ and $1/16$, respectively; also, $\theta(3m/4) > n/2$ and therefore the bottom grid is partitioned into subgrids of size $1/4$ and $1/8$, respectively.

5 Preliminary Protocol for Secure Edit Path Outsourcing

The above algorithm can be executed in the secure outsourcing setting using the round complexity of $O(\log m)$ – or, more generally, $O(\log(\min(m, n)))$ – if the servers can afford $O(\sigma mn)$ space. If, however, the servers have only linear space $O(\sigma(m + n))$, their round complexity increases to $O(\min(m, n))$ because the computation uses the total of $O(\sigma mn)$ space. This does not affect the client's round complexity, which in our preliminary solution described next is $O(\log(\min(m, n)))$. We subsequently improve it in section 6 to make it non-interactive for the client at no extra (for the client) cost.

Having described the structure of the computation, we now proceed with the description of the secure outsourcing protocol for the edit path. Recall that the client's work should be $O(\sigma(m + n))$, while the servers perform $O(\sigma mn)$ work. The protocol consists of executing the same procedure for each sub-problem in each round (starting with the problem of size $m \times n$ in round 1), at the end of which the client learns the value of the θ function at a single point. That is, for a subgrid defined by strings $\hat{\mu}_{k+1}, \ldots, \hat{\mu}_{k+a}$ and $\hat{\lambda}_{\ell+1}, \ldots, \hat{\lambda}_{\ell+b}$, the client learns $\theta(k + a/2)$ and the servers learn nothing. Here $\hat{\mu}_i$ and $\hat{\lambda}_j$ are from Σ_ϵ since after the first round each subgrid is formed by prepending or appending a number of ϵ characters to portions of the original strings.

In this protocol the client performs $O(\sigma(a+b))$ work for a subgrid of size $a \times b$, and the servers perform $O(\sigma ab)$ work. The client's work is thus characterized by recurrence $T(m, n) = T(\frac{m}{2}, n) + T(\frac{m}{2}, \frac{n}{2}) + \beta\sigma(m + n)$ and can be shown to be $\leq 4\beta\sigma(m + n)$ using the total of $O(\log m)$ rounds. In what follows, we describe a protocol for a subgrid defined by strings $\hat{\mu}_{k+1}, \ldots, \hat{\mu}_{k+a}$ and $\hat{\lambda}_{\ell+1}, \ldots, \hat{\lambda}_{\ell+b}$, in which the client learns the θ value and prepares two subgrids for the next round.

For the sake of the current description, suppose that S_1 has access to $\hat{\mu}_{k+1}, \ldots, \hat{\mu}_{k+a}$, but wants to keep the string private from S_2, and S_2 has access to $\hat{\lambda}_{\ell+1}, \ldots, \hat{\lambda}_{\ell+b}$, but likewise wants to keep its string private from S_1. S_1 and S_2 can engage in secure two-party computation, where S_1 inputs each $\hat{\mu}_i$ and the corresponding $I(\hat{\mu}_i)$, and S_2 inputs each $\hat{\lambda}_j$, the corresponding $D(\hat{\lambda}_j)$, and a vector $S(\hat{\lambda}_j, \cdot)$ that defines the cost of substituting $\hat{\lambda}_j$ with every character in Σ_ϵ. Then to be able to proceed with each step of the dynamic programming problem, they compute each $M(i, j)$ as specified, where the computation proceeds in an oblivious way as follows:

```
1. for $t = 1$ to $\sigma + 1$ do
2.     $c = (\hat{\mu}_i \overset{?}{=} t)$;
3.     $s_t = c \cdot S(\hat{\lambda}_j, t)$;
4. $s = \sum_{t=1}^{\sigma+1} s_t$;
5. $M(i, j) = \min(M(i-1, j-1) + s, M(i-1, j) + I(\hat{\mu}_i), M(i, j-1) + D(\hat{\lambda}_j))$;
```

Here $(x \overset{?}{=} y)$ denotes an equality test that outputs a bit which is set to 1 iff $x = y$. The procedure obliviously chooses the correct substitution cost from the vector $S(\hat{\lambda}_j, \cdot)$ and uses it to compute $M(i, j)$. The cost of computing each $M(i, j)$ is thus linear in σ.

To take this to the outsourcing context in which neither S_1 nor S_2 have access to the input strings or the output, we will now have the client C provide all of the inputs that S_1 and S_2 will use without learning any information about them (other than the lengths m, n, and σ). In particular, one server, say S_1, will be responsible for garbled circuit construction for a subgrid problem using oblivious execution described above, while the second server, S_2, will evaluate it on the client's inputs without knowing the meaning of the random labels that it handles. In a traditional implementation, we would have S_1 build a garbled circuit and send it to S_2, after which the client and S_1 engage in OT so that the client learns the (random) labels of the input wires corresponding to all of its inputs (namely, $\hat{\mu}_{k+1}, \ldots, \hat{\mu}_{k+a}$, $\hat{\lambda}_{\ell+1}, \ldots, \hat{\lambda}_{\ell+b}$, $I(\hat{\mu}_i)$ for each $k < i \le k+a$, and $D(\hat{\lambda}_j)$ and $S(\hat{\lambda}_j, \cdot)$ for each $\ell < j \le \ell + b$). The client then would send the labels it received from S_1 to S_2, S_2 would evaluate the garbled circuit on the client-supplied input wire labels and send the labels corresponding to the output wires to C. S_1 then would send to C the meaning of all output wire labels and C learns the result. We, however, propose a more efficient solution in which the need for computationally-intensive OTs is entirely eliminated. In detail, we have the client generate all input wire labels that it consequently sends to S_1. S_1 uses these labels to produce a garbled circuit that it sends to S_2. S_1 also sends all output wires and their meaning to C. C then sends the labels corresponding to its private input to S_2, who evaluates the circuit as before and sends the labels corresponding to the output to C.

Input: C has private strings $\hat{\mu}_{k+1}, \ldots, \hat{\mu}_{k+a}$ and $\hat{\lambda}_{\ell+1}, \ldots, \hat{\lambda}_{\ell+b}$ and the corresponding insertion, deletion, and substitution costs, namely, $I(\hat{\mu}_i)$ for $k < i \le k+a$ and $D(\hat{\lambda}_j)$ and $S(\hat{\lambda}_j, \cdot)$ for $\ell < j \le \ell + b$. S_1 and S_2 contribute no input.

Output: C obtains $\theta(k+a/2)$ and new pairs of strings $\hat{\mu}'_{k'+1}, \ldots, \hat{\mu}'_{k'+a/2}$, $\hat{\lambda}'_{\ell'+1}, \ldots, \hat{\lambda}'_{\ell'+b}$ and $\hat{\mu}''_{k''+1}, \ldots, \hat{\mu}''_{k''+a/2}$, $\hat{\lambda}''_{\ell''+1}, \ldots, \hat{\lambda}''_{\ell''+b/2}$ that define subgrids for the next round. S_1 and S_2 learn nothing.

Protocol 1:

1. C generates $a(s_\Sigma + s_C) + b(s_\Sigma + s_C + s_C|\Sigma_\epsilon|)$ pairs of random labels $(\ell_0^{(t)}, \ell_1^{(t)})$, where $s_\Sigma = \lceil \log(|\Sigma_\epsilon|) \rceil = \lceil \log(\sigma + 1) \rceil$ is the size of binary representation of an alphabet character, s_C is the size of binary representation of costs[5] in tables $I(\cdot)$, $D(\cdot)$, and $S(\cdot, \cdot)$, and $t \in [1, s_\Sigma(a+b) + s_C(a + 2b + \sigma b)]$.
2. C sends all $(\ell_0^{(t)}, \ell_1^{(t)})$ to S_1 who uses them as the input wire labels in constructing a garbled circuit.
3. C sends a single label $\ell_{b_t}^{(t)}$ for each t to S_2, where b_t is 0 or 1 depending on the corresponding bit of C's input.
4. S_1 sends the garbled circuit to S_2 and all output wire labels to C, which we denote by $(\hat{\ell}_0^{(t)}, \hat{\ell}_1^{(t)})$ for $t = 1, \ldots, s_b$, where $s_b = \lceil \log b \rceil$ is the size of the binary representation of the output $\theta(k + a/2)$ which takes on b possible values.
5. S_2 evaluates the garbled circuit using the input labels received from C and sends labels $\hat{\ell}_{b_t}^{(t)}$ that correspond to the computed output for $t \in [1, s_b]$ to C.

[5] For simplicity of representation we use fixed length s_C for costs in all tables, but this does not need to be the case. Also, because ϵ character is not present in the original strings, the values of s_Σ and s_C can be adjusted accordingly in the first round.

6. C recovers the meaning of the output (i.e., the bit b_t) for each $\hat{\ell}_{b_t}^{(t)}$ using the labels $(\hat{\ell}_0^{(t)}, \hat{\ell}_1^{(t)})$ it previously received from S_1. Let b' denote the output $\theta(k + a/2)$.
7. C forms two new sub-problems based on the value of b'. If $b' \geq b/2$, C sets:
   - $\hat{\mu}'_{k'+1}, \ldots, \hat{\mu}'_{k'+a/2} = \hat{\mu}_{k+1}, \ldots, \hat{\mu}_{k+a/2}$,
   - $\hat{\lambda}'_{\ell'+1}, \ldots, \hat{\lambda}'_{\ell'+b} = \hat{\lambda}_{\ell+1}, \ldots, \hat{\lambda}_{\ell+b'}, \epsilon, \ldots \epsilon$,
   - $\hat{\mu}''_{k''+1}, \ldots, \hat{\mu}''_{k''+a/2} = \hat{\mu}_{k+(a/2)+1}, \ldots, \hat{\mu}_{k+a}$,
   - $\hat{\lambda}''_{\ell''+1}, \ldots, \hat{\lambda}''_{\ell''+b/2} = \epsilon, \ldots, \epsilon, \hat{\lambda}_{\ell+b'+1}, \ldots, \hat{\lambda}_{\ell+b}$.

   Otherwise, C sets:
   - $\hat{\mu}'_{k'+1}, \ldots, \hat{\mu}'_{k'+a/2} = \hat{\mu}_{k+(a/2)+1}, \ldots, \hat{\mu}_{k+a}$,
   - $\hat{\lambda}'_{\ell'+1}, \ldots, \hat{\lambda}'_{\ell'+b} = \epsilon, \ldots, \epsilon, \hat{\lambda}_{\ell+b'+1}, \ldots, \hat{\lambda}_{\ell+b}$,
   - $\hat{\mu}''_{k''+1}, \ldots, \hat{\mu}''_{k''+a/2} = \hat{\mu}_{k+1}, \ldots, \hat{\mu}_{k+a/2}$,
   - $\hat{\lambda}''_{\ell''+1}, \ldots, \hat{\lambda}''_{\ell''+b/2} = \hat{\lambda}_{\ell+1}, \ldots, \hat{\lambda}_{\ell+b'}, \epsilon, \ldots, \epsilon$.

C and the servers can now engage in the next round of computation using two newly determined subgrids. Note that this solution works even when the cost tables that define insertion, deletion, and substitution are private and known only to the client. This concludes our description of secure edit path outsourcing.

6 Reducing Client's Involvement

While the solution above already significantly outperforms prior work, in this section we further improve it by making the protocol non-interactive for the client. Now the client initially sends data to S_1 and S_2 and at the end of the computation receives the result from S_1 and S_2 and recovers the edit path.

Our idea in eliminating the client's interaction such that no oblivious transfer for garbled circuit evaluation has to be introduced consists of using output wires of a garbled circuit as input wires for the garbled circuits used in the next round. To be able to do so, the server needs to obliviously compute the input strings for the next round of computation, the wires of which will then be reused in subsequent garbled circuits. Let S_1 and S_2 compute $\theta(m/2)$ in the first round of the computation, where C provides inputs $\mu_1, \ldots, \mu_m$, $\lambda_1, \ldots, \lambda_n$, $I(\mu_i)$ for $i = 1, \ldots, m$, and $D(\lambda_j)$ and $S(\lambda_j, \cdot)$ for $j = 1, \ldots, n$ in the manner described above. After determining the value of $\theta(m/2)$, S_1 and S_2 can proceed with obliviously computing strings $\mu'_1, \ldots, \mu'_{m/2}$, $\lambda'_1, \ldots, \lambda'_n$ and $\mu''_1, \ldots, \mu''_{m/2}$, $\lambda''_1, \ldots, \lambda''_{n/2}$ (with the corresponding insertion, deletion, and substitution costs) which will become inputs for the next round as follows:

```
1. $c = (\theta(m/2) \overset{?}{<} n/2)$;
2. for $i = 1$ to $m/2$ do
3.     $\mu'_i = (1-c)\mu_i + c\mu_{(m/2)+i}$;
4.     $\mu''_i = c\mu_i + (1-c)\mu_{(m/2)+i}$;
5. for $j = 1$ to $n$ do
6.     $c_j = (\theta(m/2) \overset{?}{\leq} j)$;        // can always set $c_n = 1$
7.     $\lambda'_j = (1 - c \oplus c_j)\lambda_j + (c \oplus c_j)\epsilon$;
8. for $j = 1$ to $n/2$ do
9.     $\lambda''_j = c(c_j\epsilon + (1-c_j)\lambda_j) + (1-c)(c_{(n/2)+j}\lambda_{(n/2)+j} + (1 - c_{(n/2)+j})\epsilon)$;
```

The computation of the μ'_i's and μ''_i's above is rather straightforward. To compute λ'_j's (for the larger 1/2 part), when c is set, the larger area corresponds to the bottom rows and the beginning needs to be populated with ϵ characters. So we keep λ_j if c_j is set and replace it with ϵ otherwise. When c is not set, the larger area comes from the top rows and erasing happens at the end. In this case, we keep λ_j if c_j is not set and replace it with ϵ otherwise. The expression for λ'_j above corresponds to this logic in a more compact form. To compute λ''_j's (for the 1/4 part), when c is set, the top left quadrant is used and padding happens at the end. Thus, if c_j is set, use ϵ, and use λ_j otherwise. When c is not set, we are using the bottom right quadrant with padding in the beginning. So if $c_{(n/2)+j}$ is set, use $\lambda_{(n/2)+j}$ and use ϵ otherwise.

Referring back to the example in Figure 2, the value of c determines whether the size of the top or bottom subgrid should be reduced and the values of c_j determine what portions of the strings should be replaced with ϵ. As part of the computation, the servers always process the 1/2-sized and 1/4-sized grids in the same way, regardless of from what portion of the original grid they come. This means that a subgrid processing purely depends on its size, while the origin of a subgrid of any given size is protected (i.e., unlike this computation, the positioning of subgrids in Figure 2 is not oblivious).

The above allows the servers to compute the strings themselves for the next round of the computation, but we also want to ensure that they are able to compute the rest of the input which consists of insertion, deletion, and substitution costs. Here we demonstrate oblivious computation of such values on the example of strings $\mu'_1, \ldots, \mu'_{m/2}$, $\lambda'_1, \ldots, \lambda'_n$. The costs for strings $\mu''_1, \ldots, \mu''_{m/2}$, $\lambda''_1, \ldots, \lambda''_{n/2}$ can be computed analogously. From the privacy point of view, we distinguish between two cases: (i) the insertion, deletion, and substitution cost tables are public (i.e., known to the servers) and (ii) the cost tables are private (i.e., known only to the client). Whether the cost tables are public or not will affect how a garbled circuit is constructed, but the computation built into the circuit must proceed obliviously regardless of that fact. In particular, when the cost tables are public, their values will be input into circuits as constants (in which case two inputs wires – one encoding a 0 and another encoding a 1 – can be used to encode all constants), while when they are private, the client will need to additionally produce input wires for all constant values that comprise the cost tables and communicate their values to S_1 and S_2 in the same manner as for all other private inputs. What follows describes oblivious computation of $I(\mu'_i)$, $D(\lambda'_j)$, and $S(\lambda'_j, \cdot)$ for the next round.

```
1.  for $i = 1$ to $m/2$ do
2.      $I(\mu'_i) = 0$;
3.      for $t = 1$ to $\sigma + 1$ do
4.          $c = (\mu'_i \overset{?}{=} t)$;
5.          $I(\mu'_i) = I(\mu'_i) + c \cdot I(t)$;
6.  for $j = 1$ to $n$ do
7.      $D(\lambda'_j) = 0$;
8.      $S(\lambda'_j, \cdot) = \langle 0, \ldots, 0 \rangle$;
9.      for $t = 1$ to $\sigma + 1$ do
10.         $c = (\lambda'_j \overset{?}{=} t)$;
11.         $D(\lambda'_j) = D(\lambda'_j) + c \cdot D(t)$;
12.         $S(\lambda'_j, \cdot) = S(\lambda'_j, \cdot) + c \cdot S(t, \cdot)$;
```

For compactness of presentation above, we define operations on vectors $S(\lambda'_i, \cdot)$ and $S(t, \cdot)$ in a single step, but it should be understood that all addition, multiplication, and assignment operations in this case are performed element-wise.

The above allows S_1 and S_2 to produce all inputs for the next round of the computation. Because the cost tables for insertion, deletion, and substitution are needed for each subgrid computation, when their values are public, S_1 will as before encode the constants into each circuit it forms. When, on the other hand, such values are private and should not be revealed to S_1 or S_2, S_1 will use the same wire labels for the constants as the ones provided by the client in the first round, and S_2 will also reuse the labels that it received from the client for these constants in the first round of the computation. We note that while in general reuse of garbled circuits or their parts is not safe from the privacy point of view, in this case the servers can use the same wires in multiple circuits because the labels (or inputs) on which S_2 evaluates the circuits are always the same. This means that the labels themselves do not change and do not allow S_2 to learn any information contained in the cost tables. All other labels in garbled circuits are chosen anew and therefore S_2 cannot deduce any information as a result of gate evaluation. This allows us to obtain the overall protocol as follows:

Input: C has private strings $\mu_1, \ldots, \mu_m$ and $\lambda_1, \ldots, \lambda_n$. The insertion, deletion, and substitution cost tables can be C's additional private input or known to all parties. S_1 and S_2 do not contribute any input.

Output: C obtains the edit path. S_1 and S_2 learn nothing.

Protocol 2:

1. C generates two random labels $(\ell_0^{(t)}, \ell_1^{(t)})$ for each bit of its input $\mu_1, \ldots, \mu_m$, $\lambda_1, \ldots, \lambda_n$, $I(\mu_i)$ for each $i \in [1, m]$, $D(\lambda_j)$ and $S(\lambda_j, \cdot)$ for each $j \in [1, n]$, $I(\cdot)$, $D(\cdot)$, and $S(\cdot, \cdot)$ resulting in $t \in [1, s_\Sigma(m+n) + s_C(m+n+3\sigma+\sigma^2))]$.
2. C sends all $(\ell_0^{(t)}, \ell_1^{(t)})$ to S_1, and it sends a single label $\ell_{b_t}^{(t)}$ for each t to S_2, where b_t is 0 or 1 depending on the corresponding bit of C's input.
3. S_1 uses the pairs of labels it received from C as the input wire labels in constructing a garbled circuit that produces $\theta(m/2)$, strings $\mu'_1, \ldots, \mu'_{m/2}$, $\lambda'_1, \ldots, \lambda'_n$ and the corresponding $I(\mu'_i)$, $D(\lambda'_j)$, and $S(\lambda'_j, \cdot)$, as well as strings $\mu''_1, \ldots, \mu''_{m/2}$, $\lambda''_1, \ldots, \lambda''_{n/2}$ and the corresponding $I(\mu''_i)$, $D(\lambda''_j)$, and $S(\lambda''_j, \cdot)$. Let the pairs of the output wire labels that correspond to $\theta(m/2)$ be denoted by $(\hat{\ell}_0^{(t)}, \hat{\ell}_1^{(t)})$, where $t \in [1, \lceil \log(n) \rceil]$, the labels corresponding to the output labels for the first sub-problem be denoted by $(\ell_0'^{(t)}, \ell_1'^{(t)})$, where $t \in [1, s_\Sigma(m/2+n) + s_C(m/2+n+\sigma n)]$, and the labels corresponding to the output labels for the second sub-problem be denoted by $(\ell_0''^{(t)}, \ell_1''^{(t)})$, where $t \in [1, s_\Sigma(m+n)/2 + s_C(m+n+\sigma n)/2]$.
4. S_1 sends its garbled circuit to S_2, which S_2 evaluates using the input labels received from C. S_1 stores for later reference pairs of labels $(\hat{\ell}_0^{(t)}, \hat{\ell}_1^{(t)})$ and S_2 stores the labels for the same wires $\hat{\ell}_{b_t}^{(t)}$ that it computed.
5. S_1 and S_2 now engage in the second round of the computation, where for the first circuit S_1 uses pairs $(\ell_0'^{(t)}, \ell_1'^{(t)})$ as the input wire labels as well as the pairs of the input wire labels from C that correspond to cost tables $I(\cdot)$, $D(\cdot)$, and $S(\cdot, \cdot)$. After the circuit is formed S_1 sends it to S_2 who uses the labels $\ell_{b_t}'^{(t)}$ it computed in the

first round as well as the labels for the cost tables supplied by C in the first round to evaluate this circuit.

6. S_1 forms and S_2 evaluates the second circuit of the second round and all circuits in consecutive rounds analogously. As before, for each circuit they store the labels of the output wires that correspond to evaluation of $\theta(\cdot)$ on a specific point (i.e., S_1 stores a pair for encoding each bit of the output and S_2 stores a label per output bit that it obtained as a result of circuit evaluation).
7. When S_1 and S_2 reach the bottom of recursion, S_1 sends pairs $(\hat{\ell}_0^{(t)}, \hat{\ell}_1^{(t)})$ and S_2 sends values $\hat{\ell}_{b_t}^{(t)}$ from each circuit to C. C uses the labels to reconstruct the values of the θ function on all evaluated points, from which it reconstructs the edit path as described in section 4.

We obtain the final result in which the servers' communication and computation is $O(\sigma mn)$ and the work is dominated by the same number of symmetric key or hash function operations for garbled circuit evaluation. The solution works when the servers have only $O(\sigma(m + n))$ available space. The client's communication and computation is $O(\sigma(m + n))$, where work is dominated by generation of the same number of random labels. The round complexity for the client is $O(1)$ and the round complexity for the servers can be expressed as a function of their space. When the servers' space is $O(\sigma mn)$, the round complexity is $O(\log(\min(m, n)))$. When the servers' space is lower, the round complexity increases as detailed later in this section. Security analysis is omitted due to space considerations and can be found in the full version.

Achieving linear space at the servers. As previously mentioned, our solution was designed to ensure that the servers will be able to carry out the computation using as little as $O(\sigma(m+n))$ space as m and n can be large. Because the circuit size starts from $O(\sigma mn)$ (before it exponentially reduces in each round), S_1 will generate and send to S_2 a part of the overall circuit before the next portion can be produced. Similarly, S_2 will receive and evaluate a part of a circuit at a time. Because the entries of the M-matrix can be computed row by row (or column by column), when the servers' space is constrained, the part of the circuit generated and evaluated in each round will follow the same structure of the computation (i.e., a circuit corresponding to the computation of one or more rows is produced and evaluated at a time). This causes the number of times S_1 and S_2 interact to increase from the minimum $O(\log \min(m, n))$. As the size of each circuit reduces in consecutive rounds, S_1 and S_2 will be able to process a larger portion of a circuit and then multiple circuits per interaction. Thus, the number of interactions for the servers is $O(\min(m, n))$ when they only have $O(\sigma(m + n))$ space available. In other words, for servers with memory constraints of $o(\sigma mn)$, there is a tradeoff between their space capacity and the number of interactions. This obviously does not affect the client who only sends and later receives its data.

Performance. To gain insights into performance of our solution, we compute the size of garbled circuits as a function of parameters m, n, and σ and approximate the protocol's runtime. For concreteness, we set the cost of insertion and deletion to be 1, and the cost of substitution with a different character to be 2 and with the same character to be 0.

In the circuit, we want to use the smallest possible number of bits to represent values and store intermediate results. This in particular means that the size of representation of input characters, substitution costs, and intermediate matrix values will differ. Also,

Value of $n = m$	Number of gates	Computation	Communication
250	50×10^6	8.3 min	1.4 GB
500	221×10^6	36.6 min	6.2 GB
1000	966×10^6	161 min	27.0 GB

Table 2. Servers' combined computation and communication.

with the free XOR gates technique of [14], we can implement equality testing of two ℓ-bit values using $\ell - 1$ non-free gates (i.e., XOR the inputs and compute OR of the resulting bits), multiplication of an ℓ-bit value by a bit using ℓ AND gates, addition of k ℓ-bit values from which at most one is non-zero (as on line 4 of matrix cell computation in section 5) using $k\ell$ OR gates, and regular addition and minimum as in [13]. All constants are encoded using the total of two input wires. For an $m \times n$ matrix with $\sigma = 4$, this gives us $< (n-1)(m-1)(7\log(n+m) + 18)$ non-XOR gates for the first round (without using ϵ) and $< (n-1)(m-1)(25\log(n+m) + 16)$ for all consecutive rounds. Thus, implementing the preliminary protocol in section 5 involves $< (n-1)(m-1)(82\log(n+m) + 64)$ non-XOR gates. $O(\log(n+m))$ bits are used to represent matrix elements $M(i,j)$. Removing client's involvement in the protocol introduces additional $\approx 84m + 3n(54\log(n+m) + \log n + 29)$ non-XOR gates. We note that the number of gates in our edit distance computation is larger than, e.g., in [11] for computing the Levenshtein's distance due to the generality of the edit distance problem we are solving. Some of the circuit optimizations from [11] can be applied to special cases of our problem to result in smaller circuits and faster performance.

Table 2 provides estimated number of gates and runtime of our solution assuming 100 non-free gates per msec (based on evaluation results in [11, 7]) on single-threaded commodity hardware (which can be reduced with more powerful or multi-core servers). Communication is computed using 25% savings per gate [17]. The client's work is only to generate $9n + m$ pairs of short random labels and communicate them to the servers ($180n + 20m$ bytes). This is computed assuming that the costs of insertion and deletion are known and fixed and the servers can add costs for ϵ to the circuits. We conclude that our techniques can be applied even to problems of large size, which was not feasible with other secure computation or outsourcing techniques.

7 Conclusions

This work treats the problem of secure outsourcing of sequence comparisons by a computationally limited client C to two servers S_1 and S_2. The client obtains the edit path of transforming string λ of length n into string μ of length m over an alphabet of size σ. Our solution uses new techniques to enable the servers to carry out the computation obliviously using $O(\sigma nm)$ computation and communication, while the solution is non-interactive for the client who only sends its data to and receives the result from the servers using the total of $O(\sigma(m+n))$ work. Our solution was designed to work with servers that have only $O(\sigma(m+n))$ space as m and n can be large in practice. By using garbled circuit evaluation techniques in a novel way – which may be of independent interest – we were able to completely avoid public-key cryptography. This makes our solution particularly practical, resulting in fast techniques for the edit distance and edit path computation in the privacy-preserving setting.

References

1. M. Atallah, F. Kerschbaum, and W. Du. Secure and private sequence comparisons. In *ACM Workshop on the Privacy in Electronic Society (WPES)*, 2003.
2. M. Atallah and J. Li. Secure outsourcing of sequence comparisons. In *Workshop on Privacy Enhancing Technologies (PET)*, pages 63–78, 2004.
3. M. Atallah and J. Li. Secure outsourcing of sequence comparisons. *International Journal of Information Security*, 4(4):277–287, 2005.
4. Y. Aumann and Y. Lindell. Security against covert adversaries: Efficient protocols for realistic adversaries. In *Theory of Cryptography Conference (TCC)*, pages 137–156, 2007.
5. P. Baldi, R. Baronio, E. De Cristofaro, P. Gasti, and G. Tsudik. Countering GATTACA: Efficient and secure testing of fully-sequenced human genomes. In *ACM Conference on Computer and Communications Security (CCS)*, pages 691–702, 2011.
6. M. Blanton and M. Aliasgari. Secure outsourcing of DNA searching via finite automata. In *DBSec*, pages 49–64, 2010.
7. M. Blanton and P. Gasti. Secure and efficient protocols for iris and fingerprint identification. In *ESORICS*, pages 190–209, 2011.
8. D. Bogdanov, S. Laur, and J. Willemson. Sharemind: A framework for fast privacy-preserving computations. In *ESORICS*, pages 192–206, 2008.
9. K. Frikken. Practical private DNA string searching and matching through efficient oblivious automata evaluation. In *DBSec*, pages 81–94, 2009.
10. O. Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2004.
11. Y. Huang, D. Evans, J. Katz, and L. Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*, 2011.
12. S. Jha, L. Kruger, and V. Shmatikov. Toward practical privacy for genomic computation. In *IEEE Symposium on Security and Privacy*, pages 216–230, 2008.
13. V. Kolesnikov, A.-R. Sadeghi, and T. Schneider. Improved garbled circuit building blocks and applications to auctions and computing minima. In *CANS*, pages 1–20, 2009.
14. V. Kolesnikov and T. Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP*, pages 486–498, 2008.
15. Y. Lindell and B. Pinkas. An efficient protocol for secure two-party computation in the presence of malicious adversaries. In *EUROCRYPT*, pages 52–78, 2007.
16. Y. Lindell and B. Pinkas. Secure two-party computation via cut-and-choose oblivious transfer. In *Theory of Cryptography Conference (TCC)*, pages 329–346, 2011.
17. B. Pinkas, T. Schneider, N. Smart, and S. Williams. Secure two-party computation is practical. In *Advances in Cryptology – ASIACRYPT*, pages 250–267, 2009.
18. D. Szajda, M. Pohl, J. Owen, and B. Lawson. Toward a practical data privacy scheme for a distributed implementation of the Smith-Waterman genome sequence comparison algorithm. In *Network and Distributed System Security Symposium (NDSS)*, 2006.
19. J. Troncoso-Pastoriza, S. Katzenbeisser, and M. Celik. Privacy preserving error resilient DNA searching through oblivious automata. In *CCS*, pages 519–528, 2007.
20. R. Wagner and M. Fischer. The string to string correction problem. *Journal of the ACM*, 21(1):168–173, 1974.
21. C. Wong and A. Chandra. Bounds for the string editing problem. *Journal of the ACM*, 23(1):13–16, 1976.
22. A. Yao. How to generate and exchange secrets. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 162–167, 1986.