Institut Eurécom
Department of Corporate Communications
2229, route des Crêtes
B.P. 193
06904 Sophia-Antipolis
FRANCE

Research Report RR-09-227

An Efficient Comparison Technique for Sanitized XML Trees

May 5th, 2009
Last update May 5th, 2009

Mohammad Ashiqur Rahaman, and Yves Roudier

Tel : (+33) 4 93 00 81 00
Fax : (+33) 4 93 00 82 00
Email : {mohammad.rahaman,yves.roudier}@eurecom.fr

[1]Institut Eurécom's research is partially supported by its industrial members: BMW Group Research & Technology - BMW Group Company, Bouygues Télécom, Cisco Systems, France Télécom, Hitachi Europe, SFR, Sharp, STMicroelectronics, Swisscom, Thales.

An Efficient Comparison Technique for Sanitized XML Trees

Mohammad Ashiqur Rahaman, and Yves Roudier

Abstract

When comparing different versions of large tree structured data the detection of changes and according generation of the *minimum cost edit script* is a CPU and disc I/O intensive task. State of the art requires the complete XML trees to be in memory and intermediate normalized trees to be computed before any comparison may start. Furthermore, the comparison of sanitized XML trees is not addressed in these techniques.

In this paper, we propose a comparison technique for sanitized XML documents which ultimately results into a *minimum cost edit script* that transforms an initial version of XML tree to a target tree. This method makes use of encrypted integer labels which encode the original XML structure and content. The content of the sanitized XML is readable only by a legitimate party. Based on this encoding, any third party can compare the tree nodes on the fly without relying on any intermediate normalized trees. Besides, it allows partial comparison as opposed to computing the full trees a-priori of starting any matching operation. To support our approach a modular algorithm describing the comparison technique is provided along with its complexity analysis.

Index Terms

Large XML, Sanitized XML, Partial comparison, Encryption

Contents

List of Figures

1 Introduction

Updating tree-structured data such as XML documents result in different versions of the original XML document. Detecting changes or similarity of tree structured data has many applications such as aggregation of similar XML databases, difference queries, versioning, merging of documents. Regarding sensitive data, both structure and content wise, comparison techniques must not allow the comparer, being a third party, to identify plain XML structure and content as these may contain confidential information such as about organizational strategy, marketing and financial data. Although comparison techniques for tree structured data and the generation of a *minimum cost edit script* using intermediate normalized trees have been extensively studied [1–3], they fall short with respect to enabling (1) partial comparison of large XML documents and (2) preserving confidentiality of the sensitive data. The main reason is that the full trees and intermediate normalized trees need to be in memory and thus more computation time and space are required. Typically comparison is assumed to be performed by a legitimate party. However, outsourcing any comparison activity requires a different approach.

We consider large XML documents where XML elements, its attributes and content are encrypted and thus not readable for any match maker or comparer who does not have the necessary decryption key. Imagine a company A uses various web services from another company B. A and B have a shared key and as such, B always provides encrypted WSDLs which only A can read. Now A appoints a third party C to compare different encrypted WSDLs provided by B over a period to determine the changes. C can compare such WSDLs and can detect the changes which A can decrypt and see if there are operations added or removed for example.

The differences between two versions of hierarchical structured data can be described in the form of *minimum cost edit script* which is defined as a sequence of edit operations such as node *update, insert, delete,* and *subtree move* performed sequentially over the transformed versions starting with the initial version. A well known approach to find such a script is to have initial matches of node pairs computed over the full trees of two versions for which comparison functions and approximations are applied [2]. However, this approach requires parsing of both the source and normalized trees multiple times (i.e. in pre-order, in-order, post-order) and thus the algorithms require multiple iterations.

An XML document typically consists of multiple level of tree nodes where nodes that have the same depth are positioned in the same level. In this paper, we propose a comparison technique of two tree structured data. We use a hybrid technique of tree and event based parsing, i.e. DOM [4] and SAX [5] respectively to parse the tree versions, detailed in our previous work [self reference]. It is possible to represent each level of XML nodes by DOM for a SAX event, i.e. *startElement*, and thus allowing processing such as compare or edit. This XML parsing technique assigns encrypted integer labels to nodes, i.e. XML element tags, attributes, text content of each level in breadth-first order to encode the original structural information, i.e. location, depth, number of nodes and is thus unreadable to a third party. However, the encrypted labels on nodes act as unique identifiers and allow a third party to compare nodes of different XML tree levels without knowing the node names. Note that, if there would be no confidentiality requirement then a plaintext breadth-first order label also achieves the uniqueness of the node.

The proposed comparison technique in this paper has five key characteristics:

1. *Large XML.* We focus on large hierarchical structured information such as enterprise XML documents. One such large WSDL of a SAP purchase order can be found in [6]. This schema contains 442 element definitions, of which 36 may occur unboundedly. Existing literature tackles the tree comparison problem using at least a complete representation of the trees in memory [2], then transforming those into possibly multiple normalized forms such as, *binary branch* [7], which make those techniques infeasible for large XML documents.

2. *Partial Comparison.* The comparison technique should enable partial comparison so that two document portions, e.g. one level of tree nodes, from two versions can be compared as opposed to comparing two complete trees. For example, a WSDL document containing different service operations that are only changed by adding or removing service operations resulting into some insertions or deletions of few elements in the same level in the new version.

3. *Confidential Information.* In order to support the outsourcing of comparison tasks to specialized third parities we require techniques for not only plaintext XML tree but also sanitized XML both structure and content wise. Here the challenge is to let the encrypted XML nodes to be compared without exposing sensitive structural information, e.g. number of nodes, size of the document, and content information such as plaintext values, e.g. element name, attribute name value pairs, text content. We assume a symmetric and deterministic encryption is used to encrypt the XML nodes.

4. *Edit Operations.* The edit operations are performed on an encrypted tree node of a level as opposed to its plaintext value. To enable partial comparison, the edit operations in this paper are defined with respect to sibling relationships of XML nodes in a level rather than their parent-child hierarchy. We allow four edit operations, i.e. *update, insert, delete, move* where each operation is performed on a single sibling node. The sibling node can be a leaf or internal as opposed to existing work which only allows for instance to delete a leaf node in a specific order, e.g. post-order.

5. *Minimum Cost Edit Script.* The *minimum cost edit script* is generated in a single pass algorithm on the document versions. The algorithm starts with an empty match and edit script and as it proceeds it finds the appropriate matches of nodes and the *minimum cost edit script*.

In the rest of the paper we focus on comparing encrypted tree structured data while the technique equally applies to a plaintext tree structured data. The paper is structured as follows. Section 2 reviews related work and positions our work in that

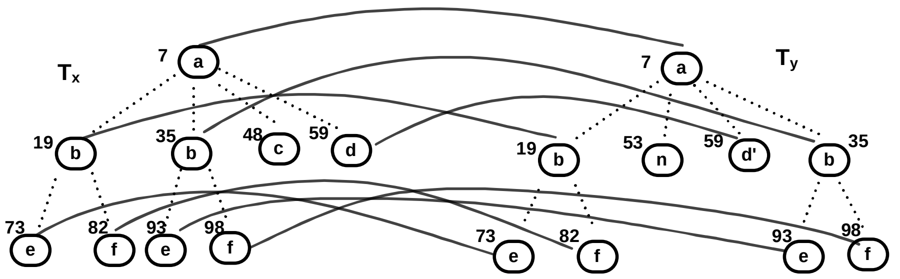


Figure 1: The running example of comparing two trees, T_x (initial) and T_y (edited). Solid lines represent appropriate matches.

context. A high level solution overview of the comparison technique is provided in Section 4 which is elaborated incrementally in later sections. Section 5 briefly introduces the XML parsing model that ensures encoding of the XML tree nodes and which are then utilized further in *minimum cost edit script* algorithm. This section also formalizes the edit operations and their cost model. The algorithm of determining a *minimum cost edit script* is described in Section 6. A complexity analysis of the algorithm is provided in Section 7.

2 Related Work

Tree comparison techniques are in general based on ideas from the string matching literature [8–11]. A comprehensive survey of edit script computation, known as tree edit distance, can be found in [12]. Due to inherent differences of tree structured data from flat data the computation technique for the former varies from different dimensions, namely atomic vs bulk edit operation, order of operation, ordered vs unordered trees, key vs keyless data and usage of intermediate trees.

In [13–15], authors support insertion and deletion anywhere in the tree whereas in [13] insertion is supported only before deletion. In [16] insertion and deletion of single nodes at the leaf level and updating of nodes anywhere in the tree are allowed. In [2] a subtree movement (bulk operation) for the ordered trees is introduced. For unordered tree comparison the authors in [1] introduce comparison techniques including copy operation for unordered trees. In our paper, four atomic edit operations (i.e. update, insert, delete, and move) are defined with respect to a single node; to be more specific a sibling node of a level of an XML document. Edit operations can be performed independently of each other except the update operation which needs to be performed before any of the other operations for a level (details in Sections 5.2 and 6).

The matching algorithms to find initial matches of node pairs for ordered trees are presented in [15, 17]. The algorithm of [15] runs in $O(n^2 log^2 n)$ which is further improved by the authors in [2] as their matching algorithm runs in $O(ne+e^2)$; where n and e are the number of leaf nodes and the 'weighted edit distance' respectively. The *minimum cost edit script* algorithm of [2] runs in $O(ND)$ time; where N is the total number of nodes of the two source trees and D is the number of misaligned nodes. The fundamental difference of our proposed algorithm (described in Section 6) from the work of [2, 15] is that we do not consider any initial set of ”matches of node pairs” between the XML nodes of the source trees which would then need to be parsed fully (against our motivation of partial comparison). The ”matches of node pairs” are computed as a side effect of the *minimum cost edit script* computation. Our proposed algorithm (i.e. MCES) enabling partial comparison proposed in this paper runs in $O(N)$ time; where N is the maximum number of nodes of the two levels of the source trees.

The authors in [7] proposed the tree edit distance between two trees should be computed based on the so called 'string edit distance' and in [7] the authors suggest 'binary branch distance' for the same computation. Both of these techniques require intermediate normalized trees of the source trees to be computed as for the former two sequences of nodes by pre-order and post-order traversal and for the later two binary tree representation of the source trees are required. The algorithm proposed by us does not require any intermediate form of a tree except a FIFO queue storing one level of tree nodes.

The comparison technique of [2] for the ordered trees assumes that tree node contains keyless data for which comparison is considered to be harder for the same problem of ordered trees. In addition, several domain characteristics are considered (e.g. semantic tagging of nodes in the source trees, nearly no duplicate tree nodes) to find an efficient solution. We consider the XML data to be keyless, but our encrypted breadth first order labeling over XML nodes during parsing time assigns unique identifiers for the parsed nodes in a level (details in Section 5.1). Furthermore, our only assumption is that the number of similar nodes between two levels of source trees is greater than that of dissimilar nodes (described in Section 6.2).

3 Tree vs Event Based Parsing

Tree structured data such as XML documents consume memory space not only for the XML nodes but also for their hierarchy and sibling relationship. For example, an empty element `<e/>` (4 bytes for the source file) could easily take 200 bytes of tree storage in Java [18].

For any enterprise XML processing in an enterprise application (e.g. ERP, SCM, CRM), the related document part must be in memory for useful processing. There are two general ways to parse XML documents: Tree and event based parsing. The Document object model (DOM) [4], Java Optimized DOM (JDOM) [19], simple API for XML (SAX) [5] and streaming API for XML (StAX) [20] are the defacto standard APIs for tree based and event based processing. DOM and JDOM require the whole document to be in memory whereas required memory for related document part might be less. For example, an empty element `<e/>` (4 bytes for the source file) could easily take 200 bytes of tree storage for these 4 bytes of source with empty information in Java [18]. JDOM optimizes the representation of XML nodes in memory by avoiding unrelated nodes but yet it needs to parse the whole document before the application can do any useful processing. SAX and StAX require only the current document node[1] in memory and an associated event is raised which any application specific event listener can then process.

Using StAX, applications can control the required events whenever they need to process rather than processing after receipt of an associated event from SAX even if the processing is not required. With respect to memory space and processing time, event based parsing (e.g. SAX, StAX) outperforms DOM [5]. However, if document updates (adding elements, attributes, changing them) are required frequently DOM is better than event based parsing [21] as tree based API (e.g. DOM) preserves the hierarchical structure of the XML documents.

As enterprise documents are large and sometimes update intensive we face contradicting requirements of consuming less memory and low processing time but maintaining hierarchy of document nodes to do the updates. To address these contradicting requirements we propose a hybrid approach. In particular, purely view based application scenarios where updates are not required should follow the event based technique. For update intensive applications, event based parsing can be used to get an event for each node for which tree based parsing (e.g. DOM) will then be used to get the children of it. While the former technique allows straightforward pre-order labeling, the latter allows labeling the nodes of each level in breadth first fashion on the fly and to free the extra memory required by DOM before moving to the next level of nodes.

4 Solution Overview

We consider XML documents (possibly with mixed content) of ordered nodes where nodes can be elements, attributes, and text. Attributes of an element can be represented as the first set of children before its sub-elements and text.

As a running example, consider two trees, T_x and T_y of Fig 1, each having two levels of nodes where T_x is the initial and T_y is the edited tree, any comparer (e.g. third party) wants to find the differences between them. Accordingly, the task is to find an appropriate transformation as described by the *minimum cost edit script* from T_x to T_y. In other words, to determine the correspondence (i.e. appropriate matching) among nodes of these two trees as shown by the solid lines in the figure. Appropriate matching also identifies an insertion or deletion of nodes as shown in the figure for the nodes n and c respectively (without solid lines). In our context, the nodes to be matched are encrypted values of XML nodes, i.e. elements, text, attributes, as opposed to plaintext values that may be updated in different versions. Whenever, we refer to the node name we refer to its encrypted value unless stated otherwise.

[1]XML elements, attributes, comments, space.

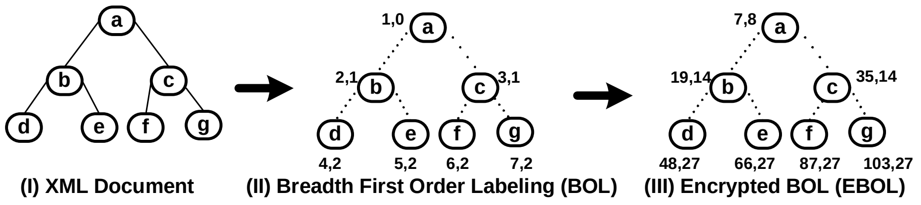


Figure 2: Solid and dotted lines represent explicit memory footprint and no memory footprint respectively. (II,III) are implicit hierarchy representations of (I).

Recall *Partial Comparison* of Section 1 which implies comparing two trees, level by level from root to leaves as opposed to bottom-up approaches (i.e. leaf to root). A special XML parsing technique (described in Section 5) allows us to determine each node's identity by location and depth wise, keeping (both the location and depth) its confidentiality. It also allows us to edit XML nodes independently of their hierarchy relationship. For example, we are able to delete any node (leaf or internal) irrespective of its children nodes. Hence, our strategy is to utilize this unique identity and thus to use a node's sibling relationship as opposed to using parent-child hierarchy in the matching process. Given two trees T_x and T_y, a node x in the level l of T_x will be compared first with the same level l nodes of T_y. If any node of T_x in level l remains unmatched the other levels of T_y can be considered to find the appropriate matching. Surely this strategy finds the appropriate matches (if exists) in the same level without parsing the complete tree.

One level of a tree T_x is said to be isomorphic [2] to a level of another tree T_y if they are identical except for node names. We need to transform T_x to a tree T'_x which is level wise isomorphic to T_y using a sequence of edit operations. As T_x is transformed to T'_x an appropriate matching between the nodes of T'_x and T_y that incurs the cheapest cost must be determined. So our goal is to generate the *minimum cost edit script* by finding these appropriate matches. The *minimum cost edit script* algorithm is described in details in Section 6.

5 XML Parsing and Edition Model

As mentioned in Section 1 we want to enable partial comparison (level wise) of large encrypted XML documents. To achieve this, the first step is to parse the XML documents level by level. In the following, the XML parsing model is described first and followed by an edition model that formalizes the edit operations and their cost model.

5.1 Parsing Model

We parse two XML document versions to be compared in breadth first order [22]. While interested readers find the details of this parsing technique in [self reference], we briefly describe it in the following. For each level of a tree version,

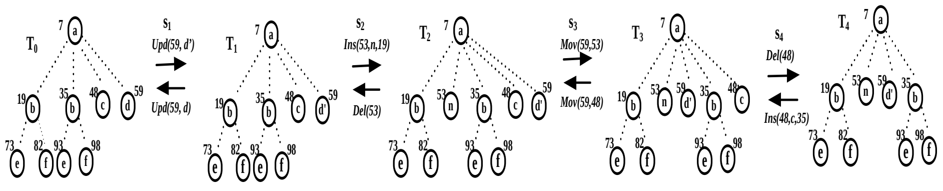


Figure 3: Basic edit operations on encrypted tree structured data.

take the sibling nodes (having the same parent) in a FIFO queue and associate an integer pair to those as these nodes are fetched from the queue in FIFO order. Each such node, having associated integer pair, captures various structural properties without consuming any memory footprint for hierarchical relationships (i.e. parent-child, siblings, left/right child) of the parsed XML node.

Breadth First Order Labels (BOL): A *BOL* is a pair of integers associated to an XML node as it is parsed in breadth first order . The first integer in the pair is the order associated with a node whose left siblings and ancestors have already been parsed and thus have associated BOLs. The second integer is the depth of the node in the document which is increased by one as new depth level is reached. The BOL starts with (1,0) as illustrated in Fig. 2 (the example given is a binary tree, but BOLs can be defined on any type of tree)

Let a be the parent of two nodes $b, c \in d_i$. We denote its BOL as B_a. Let f_{order} and f_{level} be two functions operating on a BOL respectively returning the BOL order (first attribute of the BOL pair) and BOL depth (second attribute). Let us assume that b is the last child of a parsed and that c is to be parsed next. c will be associated a BOL with $f_{order}(B_c) = f_{order}(B_b) + 1$. $f_{level}(B_a)$ uniquely identifies the depth level of the node a in d. The order of the BOL exhibits the following structural properties:

1. $f_{order}(B_a)$ uniquely identifies node a in document d and the subtree d_a rooted at a.
2. Let $B^a_{Highest}$ be the largest BOL order of a parsed node in document portion d_a; then $B^a_{Highest} > f_{order}(B_z) > f_{order}(B_a)$, where $z \in d_a$.
3. $f_{order}(B_c) > f_{order}(B_b) > f_{order}(B_a)$.

The first property is used to identify and extract a specific XML node from the FIFO queue. The second property being the depth of a node along with the first property allows us to determine all the nodes in a level. The final property shows that the BOL assigns integer values by maintaining a strict inequality among sibling nodes which are stored in the FIFO queue in that order.

A BOL is by definition plaintext and thus may reveal important structure specific information such as number of nodes and thus the size of the document and even hierarchy relationship among the nodes to an adversary. Encryption over such BOL numbers protects this undesired information disclosure.

Encrypted BOL (EBOL): Let B_a be the BOL of an XML node a. Let f_e be an order preserving encryption function [23]. The EBOL of a, denoted as E_a is a pair of integers defined as : $(f_e(f_{order}(B_a)), f_e(f_{level}(B_a)))$. While $f_e(f_{order}(B_a))$ is performed for each node a, $f_e(f_{level}(B_a))$ is performed if a is the first node in a level.

An EBOL-based parsed XML node in a level has a unique identifier and its n children where each child node x_i has $0..i-1$ left sibling and $i+1..n$ right sibling nodes respectively. Intuitively, it avoids explicit hierarchy representation and as such all the figures show the dotted lines among parent and its children. For each node, x, we assume a dummy first child node exists, (not shown in the figures) denoted as x_ϵ which is used in different edit operations (details follow).

Example: Fig 2 (III) is the EBOL representation of (II). The EBOL of the node c is pair of integers 35 and 14. The first and second integers are order preserving encrypted values computed over the unique integer, 3 of the BOL associated to c and the c's depth level, 1. □

The EBOL preserves exactly the same properties of BOL (see Fig 2). The EBOL value hides the actual node number and its depth level as opposed to the BOL numbers and thus prevents disclosure of structural information infereable with respect to a node. Most importantly, it implicitly preserves a node's hierarchy information that allows to design reach edit operations based on solely node's sibling relationship (details in next section). For example, in Fig 4, we are able to delete an internal node b without deleting its children. When an event of *startElement* of the node b is sent, b's child nodes, i.e. d, e, including the dummy child node b_ϵ are queued in the FIFO $\xrightarrow{II}$. The memory required for parent-child relationship of b and its children; and sibling relationship of its children can be freed as the sibling nodes, i.e. children, are stored in sibling order in the queue and thus sibling relationship is preserved implicitly. It is also possible to move an internal node as its child nodes are put into the queue in similar fashion.

5.2 Edit Operations

In this section we formalize the edit operations on an XML structure. We refer to the node with the identifier x, its encrypted name value with the val_x and to an EBOL-based parsed XML document with the tree, T_i for conciseness. T_i refers to the tree on which an edit operation is performed and T_{i+1} refers to the resulting tree. Formally, the edit operations are as follows:

- **Update:** The *update* of the value of a node x in T_i, denoted as *Upd(x,val)*. This operation leaves T_{i+1} as the same T_i except that in T_{i+1}, the value of x is val. This is depicted in T_0 and T_1 of Fig 3 for *Upd(59,d')*.
- **Insert:** The *insertion* of a new node x with a value v into T_i, denoted as *Ins(x,v,k)*. A node x with value v is inserted after the node k as its immediate

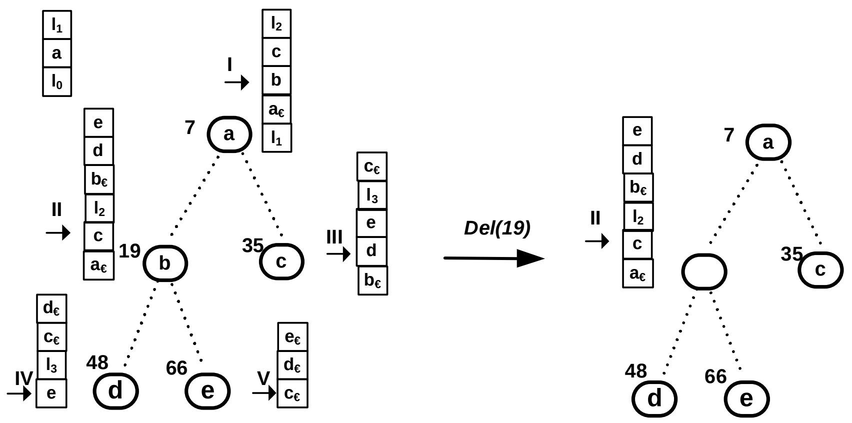


Figure 4: Deleting an internal node, b, (*Del(18)*) using EBOL-based XML parsing. The FIFO queue stores the nodes of the current level and keeps track of the current parsed node shown as $\xrightarrow{[I..V]}$. The nodes including the dummy nodes in one level are delimited by two l_i entries.

right sibling node in T_i. In particular, if $r_1, ..., r_m$ are the right sibling nodes of k in that order in T_i, then $x, r_1, ..., r_m$ are the right sibling nodes of k in T_{i+1}. In case of an insertion of a node as a first sibling node k is considered to be the dummy node as mentioned in Section 5.1. Insertion can be performed after any leaf or internal node. (T_1 and T_2 of Fig 3 for *Ins(53,n,19)*.

- **Delete:** The *deletion* of a node x from T_i, denoted as *Del(x)*. The resulting T_{i+1} is the same as T_i without the node x. In particular, if $l_1, ...l_n, x, r_1, ..., r_m$ is the sibling sequence in a level of T_i, then $l_1, ...l_n, r_1, ..., r_m$ is the sibling sequence in T_{i+1}. To delete a leaf sibling node is straightforward as depicted in T_3 and T_4 of Fig 3 for *Del(48)*. When deleting an internal sibling node, its children are stored in the FIFO queue as shown in Fig 4 so that these nodes can be fetched from the queue and thus be considered for the next level comparison.
- **Move:** The *move* of a node x after the node y, denoted as *Mov(x,y)* in T_i. T_{i+1} is the same as T_i, except x becomes the immediate right sibling of y. The children of the moved node are kept in the queue in similar fashion as the delete operation. (T_2 and T_3 of Fig 3 for *Move(59,53)*)

If the target level of the moved node is the same as its previous level then it is an intra level move (as in Fig 3). However, for any inter level move, for instance in Fig 5, the node n of T_1 is moved after the node e to the lower level (or higher level), the plain strategy mentioned in Section 4 may not find the appropriate match in the same level. In particular, for the first level comparison it will be identified as n is

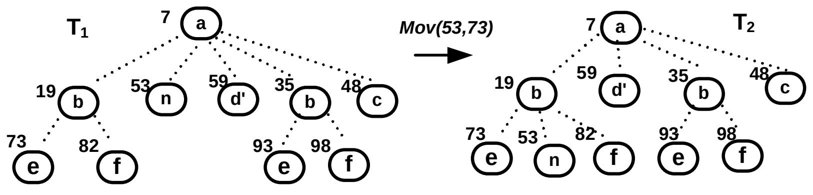


Figure 5: Inter level moving of node n in 1st level of T_1 to 2nd level of T_2.

deleted (or inserted) whereas it is moved to another level. Intuitively, when a node is moved upwards in a higher level it would be matched in for *insert* case as it is a new node for that level. In order to find an appropriate match, an edit cost model is required. In this regard, we also maintain an array of currently matched node pairs and the current edit script using which such an inter level move is detected (details in a later section).

5.3 Edit Script and Cost Model

The basic formalizations of *edit script* and *cost model* have been extensively studied in the literature [2,7,24,25] and as such we utilize the formalizations from there.

Consider, $T_0 \xrightarrow{s_1} T_1$ where T_1 is the result of applying the edit operation s_1 to T_0. An edit script, S, is a sequence of edit operations when applied to T_0 transforms it to T_i. Given a sequence $S = s_1 \ldots s_i$ of edit operations, we say $T_0 \xrightarrow{S} T_i$ if there exist $T_1, T_2, \ldots T_{i-1}$ such that $T_0 \xrightarrow{s_1} T_1 \xrightarrow{s_2} T_2 \ldots T_{i-1} \xrightarrow{s_i} T_i$. $S = \{Upd(59, d'), Ins(53, n, 19), Mov(59, 53), Del(48)\}$ is an edit script that transforms T_0 to T_4 of Fig 3.

There are three interesting features of this edit script. (1) The edit operations are defined with respect to sibling relationships. (2) The edit script generation algorithm does not require any initial matches. Instead it computes the appropriate matches during algorithm execution. (3) The algorithm requires only one pass to generate the *minimum cost edit script* and to compute the matches as it traverses one level of the trees in breadth first order.

There may be several edit scripts that transforms T_0 into the same resulting tree T_4. For example, the following edit script, $S' = \{Del(59), Ins(59, d', 48), Ins(53, n,$-$19), Del(93), Del(98), Del(35), Ins(35, b, 48), Ins(93, e, b_\epsilon), Ins(98, f, 93), Del(48)\}$,- when applied in Fig 3 it also transforms T_0 to T_4. Note that, for the insertion of $Ins(93, e, b_\epsilon)$ the dummy node b_ϵ is considered as mentioned in Section 5.2.

Clearly, the edit script, S', performs more work than that of S and thus it is an undesirable edit script to transform T_0 to T_4. In effect, to determine a *minimum cost edit script* a *cost model* is required. The cost of an edit operation depends on the type of operation and the nodes involved in the operation. Let $C_d(x)$, $C_i(x)$,

$C_u(x)$, and $C_m(x)$ denote respectively the cost of deleting, inserting, updating and moving operations respectively. In general, these costs may depend on the value of x, as well as its position in the tree. In particular, the encrypted value represented by node x and its position in the sibling order in a level.

In this paper, we use a simple cost model similar to [2] where deleting, inserting, and moving a node are considered to be unit cost operations, i.e. $C_d(x) = C_i(x) = C_m(x) = 1$ for all x. For the cost $C_u(x)$ of updating an encrypted value associated to a node x, a function $diff$ is defined as follows: $diff(val_x, val_y)$ that returns 0 if encrypted values represented by val_x and val_y are same otherwise a nonzero value is returned indicating that there has been an update.

6 Determining Edit Script

This section describes the algorithm to determine the *minimum cost edit script*. We refer to a level of XML nodes of T_x as $l(T_x)$, to a node x as a node in a level and to a two dimensional array M as consisting of matching node pairs (x_i, y_j), where $x_i \in T_x$ and $y_j \in T_y$. We define a function $exist()$ when applied on a tree T_y (or M), returns the val_y (or TRUE) if E_x matches any node y in T_y (or x_i matches any node in M as a peer node) i.e. $\exists E_y = E_x$ or $\exists x_k = x_i$ in M, where val_y is the encrypted node value associated to y. Recall from Section 5.1, E_x is the EBOL value of x.

6.1 Algorithm Overview

Given a tree T_x (the initial tree), a tree T_y (the edited tree), the algorithm generates a *minimum cost edit script* that transforms T_x to T_y. It does not take any initial matches as mentioned in Section 5.3 and as such it determines the matches during the execution of the algorithm. It starts with an empty edit script S and adds edit operations to S as it follows through. Consequently, the algorithm requires only one pass as opposed to multi phases on the trees as motivated in Section 1.

We assume two root nodes match without loss of generality. The algorithm takes one level of nodes at once consisting of all the sibling nodes from one tree and compares these with a level of sibling nodes of the other tree. Once a matching pair of nodes is found, denoted as (x_i, y_j), the pair is added to M. The corresponding edit operation is added to S as it is applied to $T_i = T_x$ which is then transformed to T_{i+1}. Here, x and y are the matched peer nodes and i and j are their respective position in the sibling order which might change upon an edit operation (details follow). When the algorithm terminates each node in T_x has an appropriate match in T_y as shown by the solid lines in Fig 9(V) (see Appendix).

6.2 Appropriate Matching

The straightforward way to find appropriate matches of node pairs is to start with an initial set of matches [7] determined by predefined similarity evaluation functions. Depending on the data set and domain of the matching this evaluation may vary. For example, if the comparing versions of the XML versions have keys or unique identifiers then evaluation would first match the keys of the versions. If the data set is keyless then some domain dependent matching rules or criteria are used as evaluation characteristics. Intuitively, a criteria stating that a match is better than other if the former incurs cheaper edit cost than the later.

We consider different versions of XML trees to be keyless. However, EBOL-based parsing associates each XML node with an encrypted integer value pair that acts as a unique identifier to that node as described in Section 5.1.

As mentioned earlier, we would like to find appropriate matching pairs of nodes during the execution of the *minimum cost edit script* algorithm rather than finding an initial match and then updating that. The rationale is as follows: (1) Initial match finding requires parsing the large XML documents and their normalized forms into memory before the algorithm execution which is undesirable in our context. (2) We want to enable partial comparison of level of trees as motivated in Section 1 that requires appropriate matching of sibling nodes without knowing their descendants. (3) In our case, the matching should be performed over encrypted values as opposed to plaintext values and thus it is not straightforward. For (1), we utilize the EBOL based parsed nodes of a level as a first class values for comparison without generating any memory and CPU intensive intermediate normalized form of the trees. For (2), we define matching criteria for a node that do not require comparing descendant nodes of the other node except its direct children that are stored in the queue. For (3), matching criteria are applied over the encrypted values as the XML node names are not plaintext. Therefore we formalize the criteria that allow us to match nodes as a side effect during the algorithm execution. The first matching criteria can be stated informally as follows: nodes having dissimilar encrypted values should not be matched with each other.

Criteria 1: Sibling nodes $x \in T_x$ and $y \in T_y$ can match only if $E_x = E_y$. Recall that E_x denotes the EBOL value of the node x.

Given the first criteria is fulfilled two nodes can match after invoking the function $diff(val_x, val_y)$. Recall that $diff()$ returns 0 if val_x and val_y are same otherwise a nonzero value, where they denote the encrypted values of the nodes x and y respectively. For the latter an updated node is detected and is used in the algorithm as an update edit case. As mentioned in Section 1, we rely on symmetric and deterministic encryption, this check is then only matching the corresponding ciphertexts.

The second matching criteria is about the similarity of sibling nodes having direct children. To state informally, two nodes can match if their direct children (as stored in their corresponding queues) also potentially match. Two functions $same(x, y)$ and $max(|x|, |y|)$ are defined where x and y are the nodes to be compared and $|x|$ and $|y|$ are their number of children. The former returns the number of child nodes having the same EBOL and the latter returns an integer representing the maximum number of child nodes of the two nodes.

Criteria 2: Sibling nodes $x \in T_x$ and $y \in T_y$ can match only if

$$\frac{same(x,y)}{max(|x|,|y|)} > t; \quad where \quad 0 \leq t \leq 1.$$

The value of t is a threshold that depends on the domain and chosen by the comparer. For instance, if the comparing XML trees are two purchase order documents having lot of `item`, `price`, `quantity` elements then it is quite likely that two documents may have lot of same elements in a level and as such, the comparer can choose a higher value for $t \geq \frac{1}{2}$. If two WSDL documents are compared to check for operations change (addition or remove) then probably the value of t is lower, i.e. $t \leq \frac{1}{2}$ as the number of operations are less.

Finally, we assume that the number of nodes in a level of a tree that are similar to a level of nodes of another tree is not smaller then that of dissimilar nodes in the same level. As such, one node has bigger chance to match with another node if their sibling nodes also potentially match. This assumption reflects the goal of partial comparison where two versions of a document differs mostly in the same level as mentioned in Section 1.

Assumption: For a domain dependent threshold value t, nodes $x \in l(T_x)$ and $y \in l(T_y)$, the number of nodes that satisfy $(E_x = E_y)$ is $\geq t$, for t as defined in criteria 2.

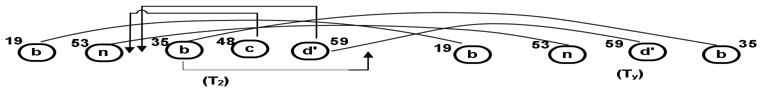
Now, we describe the insight of the algorithm using different edit cases, i.e. *update, insert, delete,* and *move*. While the first criteria is used for the first three cases, the second criteria is used for the move case.

6.3 Edit Cases

We illustrate with the running example in Fig 9 to determine a *minimum cost edit script* incrementally (level by level) that transforms $T_x = T_0$ to T_4 by finding appropriate matches shown by the solid lines. Intuitively, T_4 is level wise isomorphic to T_y.

Update Case. In update case, for each node x of T_0 the function $exist()$ is invoked to find whether x_i exists in T_y. If successful the function returns val_y, then the function $diff(val_x, val_y)$ is called. For a nonzero value, we add the edit operation $Upd(x, val_y)$ to S, and we add a match (x_i, y_j) to M. Consequently, we apply the update operation to T_0. Ultimately, we have transformed T_0 to T_1 by assigning $val_x = val_y$ such that $E_x = E_y$ for each node x in T_0 which has a corresponding identifier in T_y ($exist(x_i, -, U, -)$ in T_y). Note that, even if there is no updated node in T_y meaning a 0 is returned from $diff$, M may have pairs where each peer node in a pair has a corresponding matched node from the other tree. Fig 9(II) shows $Upd(59, d')$ when applied to T_0, the transformed tree is T_1. Fig 9(II) also shows all the matching node pairs in M.

Insert Case. To find the inserted nodes in T_y, we take the nodes, w of T_y such that w is not a peer in any of the pairs in M. For each such w we add the edit operation $Ins(w, val_w, k)$ to S, meaning w will be inserted after node k in T_x with



X_i	Y_j
19_1	19_1
53_2	53_2
35_3	35_4
59_5	59_3

Figure 6: An appropriate match with rearranging the sibling nodes.

the encrypted value val_w. The position k is determined with respect to the sibling relationship of already matched pairs of M. In particular, the peer node x_i of T_x in $M(x_i, y_j)$ for which w is the immediate right sibling of y_j, is the node k in T_x. We apply the insert operation to T_x and add the node pair, (x_{k+1}, w_{j+1}) to M. If w is the first sibling in T_y, i.e. left most child, then k is considered to be the dummy child node of the level in question of T_x. In effect, insertion operation changes the sibling positions of existing peer nodes of T_x in M. The running example of Fig 9(III) shows the resulting tree T_2 after $Ins(53, n, 19)$ and the updated sibling positions of peer nodes in M. For clarity, only the new solid line resulted for the new matched pair is shown in the figure.

Move Case. In this case we consider the pairs of M for which peer node's sibling positions are not the same. If it is the case we say peer nodes are miss arranged. In the Fig 9(III) nodes $35, 59$ in T_2 are miss arranged with respect to their respective sibling positions in T_y as depicted in M. We add move operations to S to arrange the sibling order. We explain the details in Section 6.4. In the running example a $Mov(59, 53)$ is added to S, and applied to T_2 to transform it to T_3 (Fig 9(III) to (IV)). Note that no new match is found by this operation, however the sibling position is changed as depicted in M.

Delete Case. To find the deleted nodes of T_x, we take the nodes x in T_x such that x is not a peer in any of the pairs in M. For each such node x, we say that either it is deleted from the level it was in T_x or it is moved to some other level. For the partial comparison purpose as motivated in Section 1 we can safely conclude the former. Accordingly we can add edit operation $Del(x)$ to S which in turn changes the existing sibling positions in M as insertion and move cases. Fig 9(V) shows the resulting tree T_4 after performing $Del(48)$ on T_3.

While the insert, move and delete cases can be applied independently of each other, the update case is required to be performed first so that first set of matching pairs is determined which are then used in other cases. After applying the edit script $E = (Upd(59, d'), Ins(53, 19), Move(59, 53), Del(48))$ the first level of the initial tree T_0 is transformed to T_4 which is isomorphic to T_y with respect to first level and S is the final edit script conforming to the first level, and M contains the matched peer nodes in that level.

6.4 Rearranging Sibling Nodes

As mentioned in the move case and shown in Fig 9(III) there might be miss arranged peer nodes in M. In general, (1) there may be more than one sequence of moves that will arrange the siblings in the order of the edited tree. For example, Fig 6 that shows the siblings of Fig 9(III), there are at least three sequences to arrange the sibling nodes of T_2 to transform to T_3. The first consists of moving nodes c and d' after n in that order. The second consists of moving the node b after d'. The third consists of moving the node d' after n. All yield the same transformed tree. Clearly, the first is undesirable as it requires more moves and thus concedes more cost. However, to pick the desired one from the rest two having one move is also tricky as the former has direct children as opposed to the latter and thus the former potentially require more moves than the latter. In case of several sequences having the same number of moves any one can be picked.

To ensure that the edit script generated by the algorithm is of minimum cost, we must first find the shortest sequence of moves to arrange the siblings. In this context, we may utilize EBOL-based encrypted identifiers for XML nodes. Recall from Section 5 that identifiers are integers that follow an equality from left to right in a sequence of siblings. The idea is to find a common sequence of sibling nodes that maintain this inequality and move the nodes that are not in the common sequence. This strategy might be useful for a potentially smaller number of missarranged nodes, indicating majority of the sibling nodes are in order with respect to EBOL inequality. However, we can not rely on the EBOL inequality property as there might be lot of missarranged nodes due to multiple insertion, deletion and moving of the nodes. As such, we use the second criteria of Section 6.2 as part of the common sequence definition.

Let the sequence of EBOLs of siblings in a level of T_x and T_y are $L_x = x_1, ..., x_i$ and $L_y = y_1, ..., y_j$ where i and j are the respective position of the peer nodes x and y in the sibling order and let $(x_i, y_j) \in M$ for any $i, j \in \mathbb{N}$. A subsequence of L can be found by removing any number of EBOLs from L.

The longest sibling subsequence of L_x and L_y, denoted as $\mathit{LSS}(L_x, L_y)$ is a sequence $L_{xy} = (x_1, y_1)...(x_i, y_i)$ of matched pair nodes such that (1) $x_1...x_i$ is a subsequence of L_x; (2) $y_1...y_i$ is a subsequence of L_y; (3) for $1 \leq k \leq i$, $y_k = exist(x_k, -, U, -)$; (4) for $1 \leq k \leq i$, $\frac{same(x_k, y_k)}{max(|x_k|, |y_k|)} > t$; where $0 \leq t \leq 1$; and (5) there is no sequence L' that satisfies previous four conditions and longer than or equal to L_{xy}. Note that, the condition four refers to the second matching criteria.

We are required to rearrange the sibling nodes, x_i in T_x, which are not in the longest sibling subsequence L_{xy}, i.e. $x_j \notin L_{xy}$, using move operations. Accordingly, we move such nodes, x_j of T_x that transforms T_x to an isomorphic tree of T_y. In Fig 6, for $L_x = 19, 53, 35, 59$ and $L_y = 19, 53, 59, 35$, the LSS is $19, 53, 35$ that corresponds the matching pairs $(19_1, 19_1), (53_2, 53_2), (35_3, 35_4)$ respectively. Then the shortest sequence of move operations is $Mov(59, 53)$ as 59 is not in $19, 53, 35$.

In [2], a similar strategy is used to determine the shortest sequence of move operations. Due to the condition four which only considers the direct children as opposed to all the descendants, we are able to determine the shortest sequence of

```
1. Input: $l(T_x), l(T_y)$; Output: $M$ and $S$.
2. $M = \epsilon$; $S = \epsilon$
3. Load the nodes of $l(T_x)$ and $l(T_y)$. /*load one level of $T_x$ and $T_y$*/
4. **Update Case:** for each node $x \in l(T_x)$
   (a) $val_y = exist(x, -, U, -)$
   (b) if $val_y! = NULL$
       $UpdateMatch((x, y), Update)$.
       $v = diff(val_x, val_y)$. /*appropriate matching*/
       i. $if(v! = 0)$
          A. Append $Upd(x, val_y)$ to $S$.
          B. Apply $Upd(x, val_y)$ to $T_x$.
5. **Insert Case:** for each $y_j \in l(T_y)$; if $exist(-, y_j, -, M) = FALSE$ /*$y_j \notin M$; $y_j$ as a
   peer node*/
   (a) $k = FindSibling(y_j)$.
   (b) $UpdateMatch((k, y_j), Insert)$.
   (c) Append $Ins(y_j, val_{y_j}, k)$ to $S$.
   (d) Apply $Ins(y_j, val_{y_j}, k)$ to $T_x$.
6. **Move Case:** Take the sequences of miss arranged siblings: $L_x, L_y$;
   (a) $X = ArrangeSibling(L_x, L_y)$ /*Missarranged nodes of $T_x$*/
   (b) for each $x_i \in X$
       i. $k_n = FindSibling(x_i)$.
       ii. if $n > i$ then $UpdateMatch((x_{n+1}, y_j), Delete)$.
           /*if moved to right*/
           if $n < i$ then $UpdateMatch((x_{n+1}, y_j), Insert)$.
           /*if moved to right*/
       iii. Append $Mov(x_i, k)$ to $S$.
       iv. Apply $Mov(x_i, k)$ to $T_x$.
7. **Delete Case:** for each $x_i \in T_x$; if $exist(x_i, -, -, M) = true$ /*if $x_i \notin M$; $x_i$ as a peer
   node*/
   (a) $UpdateMatch((x_i, \_), Delete)$.
   (b) Append $Del(x_i)$ to $S$.
   (c) Apply $Del(x_i)$ to $T_x$.
```

Figure 7: Algorithm *Minimum Cost Edit Script(MCES)*

moves without parsing and loading the descendants into memory. We also use the algorithm of [8] in similar fashion of [2] that can compute the *LSS* of two sequences in time $O(ND)$, where $N = |L_x| + |L_y|$ and $D = N - 2|LSS(L_x, L_y)|$.

```
1. Function exist($x_i, y_j$,U,M)
    (a) if (U) then for each node $y_j \in l(T_y)$; /*update case*/
        do if $E_y = E_x$ return $val_{y_j}$;    else return $NULL$; endfor
    (b) if (M) then for each node pair $\in M$
        if $y_j \notin M$; return true; /*insert case*/
        if $x_i \notin M$ return true; /*delete case*/
2. Function UpdateMatch($(x_i, y_j)$, editcase)
   $q, t, u, v$ are integers
    (a) if (editcase=Update)
        then $M[q] = (x_i, y_j)$, such that $\forall t$, $0 < t < q$; $M[t] = (x_u, y_v)$ and $i > u$. /*adding
        pair nodes in M*/
    (b) if (editcase=Insert) for each pair $M[q] = (x_u, y_v)$, such that $u > i$, do $M[q + 1] =$
        $(x_u, y_v)$. endfor
        $M[i] = (x_i, y_j)$ /*updating sibling position*/
    (c) if (editcase=Delete) for each right sibling node, $x_{u>i}$ of $x_i$, such that $(x_u, y_v) \in M$ do
        /*updating sibling position*/
        replace $u$ with $u - 1$; i.e. $(x_{u-1}, y_v) = (x_u, y_v)$. endfor
3. Function ArrangeSibling($L_x, L_y$)
   Compute $L_{xy} = LSS(L_x, L_y)$. return $\forall x \notin L_{xy}$; /*miss arranged peer node*/
4. Function FindSibling($y_k$)
   for each $(x_i, y_j) \in M$
   if ($y_k$ is the right sibling of $y_j$) return $x_i$. /*left peer node*/
```

Figure 8: Functions *exist, UpdateMatch, ArrangeSibling, FindSibling* invoked by *MCES* algorithm.

6.5 Minimum Cost Edit Script Algorithm

Now we present the complete algorithm to generate a *minimum cost edit script*, S, that transforms a level of an initial version, $l(T_x)$ to be isomorphic to a level of the target version $l(T_y)$ by computing the necessary matching node pairs. The algorithm, shown in Fig 7, takes one level of tree nodes from T_x and T_y and uses four other functions, namely, *exist, ArrangeSibling, UpdateMatch,* and *FindSibling* shown in Fig 8. Additionally, it makes use of a two dimensional array, M, to compute the matching pairs in sibling order. It combines all the edit cases in one breadth-first traversal of T_x and T_y.

As mentioned in Section 6.3, edit cases are independent and as such any two edit cases may update the matches in M and the edit script, S in parallel. We assume there is a multi threading control mechanism exist that disallows updating M and S by an edit case while another is updating them and thus is not depicted in the algorithm. An edit case adds the edit operation to S and applies the edition to T_x only if it has access to M and S. As mentioned before that two roots of T_x and T_y are matched without any loss of generality.

The function *exist*(x_i, y_j,*U,M*) is called from update, insert, and delete cases where parameters are filled in depending on a calling case. For update x_i, U, for insert y_j, M and for delete x_i, M are filled in. For update, the nodes in $l(T_x)$ and for others the node pairs in M are compared. Each edit case generates the matching node pairs and preserves the current sibling order of T_x in the array M by invoking the function *UpdateMatch*. Recall from Section 5.2 an edit operates with respect to node sibling order. Depending on the invoking edit case *UpdateMatch* takes care of the sibling order (see Fig 8). For insert case, the function inserts the new pair in the right position by moving the existing pairs and for the delete case, updates the sibling position of the pair in M according to the sibling position of T_x. The move case invokes the *UpdateMatch* with delete or insert parameter when a node is moved to right or left respectively. The insert and move cases invoke the *FindSibling* function to determine the node after which node it should insert or move. The *ArrangeSibling* function computes the *LSS* as described before and returns the nodes of T_x to be moved when invoked by the move case. All edit cases append the edit operation to S and apply it to T_x after successful matching.

When the algorithm runs for the first level T_4 becomes first level isomorphic to T_y (Fig 9). Intuitively, the algorithm can be applied repeatedly for other levels and as such T_x can be transformed to an isomorphic tree of T_y in one pass of the algorithm. In Fig 9 the tree T_4 happens to be isomorphic to T_y for the next level also. To address the inter level moving as shown in Fig 5 a simple approach is followed. For a node $x \in T_x$ which is detected as deleted in a level, the $exist(x, -, U, -)$ function can be called in T_y for other levels in T_y. If a non null value, y, is returned then $FindSibling(y)$ is called further to get the sibling position k for which y is the immediate right sibling (like an insert case). Now we can replace the $Del(x)$ with the $Move(x, k)$ in S.

7 Complexity Analysis

The complexity of the *minimum cost edit script* algorithm is analyzed space and time wise. Let the number of nodes in a level of T_x and T_y be m and n respectively; and let d be the maximum number of levels of both trees.

Space. As described in Sections 1 and 5 EBOL-based XML parsing avoids explicit hierarchy representation of XML nodes in memory, clearly the algorithm requires less memory as opposed to existing techniques. As such, the space complexity is only about the nodes that are stored in the FIFO queue. Let the average space required for a node x and its EBOL identifier be s_x and s_e respectively. As mentioned in Section 6, the algorithm takes one level of nodes from T_x to be compared with one level of nodes from T_y at once and free the memory afterwards. So, the required space is proportional to $O(ms_x ns_y {s_e}^2)$ which would have been $2d$ times of this proportion if the full trees and their normalized trees would be in memory.

Running Time. Let c_d and c_e be the average cost of computing $diff()$ and $exist()$ respectively; let p and q be the number of comparisons required to find the right position of a pair in M and for the function *FindSibling* (i.e. to find the node in T_x after which a node is inserted or moved) respectively.

The *update* case takes time $O(m(c_e+c_d)p)$ to match the nodes of a level of T_x with the nodes of a level of T_y. Now, let n_x and n_y be the number of unmatched nodes of a level of T_x and T_y for *update* case respectively. From the assumption of Section 6.2, we know that such number is bounded by $< t$. Consequently, n_x and n_y nodes will be matched by *delete* and *insert* cases respectively.

The *insert* case running time is proportional to $O(nc_epq)$ and for the *delete* case it is $O(mc_ep)$.

For the *move* case, it requires to compute the $same()$ function on the FIFO queues for the missarranged nodes x and y. From the same assumption of Section 6.2, we know that the number of nodes having equal EBOL is bounded by $> t$ and as such considering the number of missarranged nodes is $|X|$ (see Fig 7), we may approximate the time takes for moving nodes of T_x is $O(|X|qpt)$.

Based on this analysis we prove the following theorems (see Appendix).

Theorem 1 The *MCES* algorithm computes the minimum cost edit script S and does it in time $O(N)$ where N is the maximum number of nodes in a level of T_x and T_y.

Theorem 2 The edit script S that is generated by the *MCES* algorithm transforms T_x to an isomorphic tree of T_y.

8 Conclusion

We have provided a comprehensive technique and an algorithm to compare two encrypted tree structured data and generate a *minimum cost edit script*. The script transforms one initial tree into a tree which is isomorphic to the edited tree. The script is comprised of edit operations that are performed on single sibling nodes. The algorithm does not take any input matching pairs of nodes but generates a complete set of matched node pairs along with the *minimum cost edit script*. We showed how to achieve partial comparison over such trees without any of its intermediate normalized forms. While the solution is geared to the sanitized XML data it is equally applicable to any plaintext tree. To this end, we showed the space and time required for the algorithm are $O(ms_xns_ys_e{}^2)$ and $O(N)$ respectively.

References

[1] S. S. Chawathe and H. Garcia-Molina, "Meaningful change detection in structured data," in *SIGMOD '97: Proceedings of the 1997 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1997, pp. 26–37.

[2] S. S. Chawathe, A. Rajaraman, H. Garcia-Molina, and J. Widom, "Change detection in hierarchically structured information," in *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 1996, pp. 493–504.

[3] A. Nierman and H. V. Jagadish, “Evaluating structural similarity in xml documents,” 2002, pp. 61–66.

[4] P. L. Hgaret, R. Whitmer, and L. Wood, “Document Object Model (DOM), http://www.w3.org/dom/.” [Online]. Available: http://www.w3.org/DOM/

[5] “About Sax, http://www.saxproject.org/.” [Online]. Available: http://www.saxproject.org/

[6] “A Purchase Order WSDL Document, SAP Enterprise Services Workplace, http://esoadocu.sap.com/socoview(bd1lbizjptgwmczkpw1p bg==)/get_wsdl.xml?packageid=dbbb6d8aa3b382f191e00 00f20f64781&id=0afcbb068cee3d59a67b420bc73f2f1b.”

[7] R. Yang, P. Kalnis, and A. K. H. Tung, “Similarity evaluation on tree-structured data,” in *SIGMOD ’05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2005, pp. 754–765.

[8] E. W. Myers, “An o(nd) difference algorithm and its variations,” *Algorithmica*, vol. 1, pp. 251–266, 1986.

[9] R. A. Wagner, “On the complexity of the extended string-to-string correction problem,” in *STOC ’75: Proceedings of seventh annual ACM symposium on Theory of computing*. New York, NY, USA: ACM, 1975, pp. 218–223.

[10] R. A. Wagner and M. J. Fischer, “The string-to-string correction problem,” *J. ACM*, vol. 21, no. 1, pp. 168–173, 1974.

[11] S. Wu, U. Manber, G. Myers, and W. Miller, “An o(np) sequence comparison algorithm,” *Inf. Process. Lett.*, vol. 35, no. 6, pp. 317–323, 1990.

[12] P. Bille, “A survey on tree edit distance and related problems,” *Theoretical Computer Science*, vol. 337, pp. 217–239, 2005.

[13] T. Jiang, L. Wang, and K. Zhang, “Alignment of trees - an alternative to tree edit,” in *CPM ’94: Proceedings of the 5th Annual Symposium on Combinatorial Pattern Matching*. London, UK: Springer-Verlag, 1994, pp. 75–86.

[14] D. Shasha and K. Zhang, “Approximate tree pattern matching,” in *In Pattern Matching Algorithms*. Oxford University Press, 1997, pp. 341–371.

[15] K. Zhang and D. Shasha, “Simple fast algorithms for the editing distance between trees and related problems,” *SIAM J. Comput.*, vol. 18, no. 6, pp. 1245–1262, 1989.

[16] S. M. Selkow, “The tree-to-tree editing problem.” *Inf. Process. Lett.*, vol. 6, no. 6, pp. 184–186, 1977.

[17] D. Shasha and K. Zhang, “Fast parallel algorithms for the unit cost editing distance between trees,” in *SPAA ’89: Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 1989, pp. 117–126.

[18] M. Kay, “An anatomy of an xslt processor, http://www.ibm.com/developerworks/xml/library/x-xslt2/.” [Online]. Available: http://www.ibm.com/developerworks/xml/library/x-xslt2/

[19] “Java Document Object Model, http://www.jdom.org/.” [Online]. Available: http://www.jdom.org/

[20] “The Streaming Api for XML, http://stax.codehaus.org/.” [Online]. Available: http://stax.codehaus.org/

[21] B. Siddiqui, “Understanding DOM, IBM Developerworks,http://www.ibm.com/developerworks/edu/x-dw-xudom-i.html.”

[22] C. E. L. Thomas H. Cormen and R. L. Rivest, “Introduction to Algorithms, http://highered.mcgraw-hill.com/sites/0070131511/.” [Online]. Available: http://mitpress.mit.edu/catalog/item/default.asp?ttype=2&tid=3440

[23] R. Agrawal, J. Kiernan, R. Srikant, and Y. Xu, “Order preserving encryption for numeric data,” in *SIGMOD ’04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2004, pp. 563–574.

[24] H. Lee, R. T. Ng, and K. Shim, “Extending q-grams to estimate selectivity of string matching with low edit distance,” in *VLDB ’07: Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 195–206.

[25] E. Ukkonen, “Approximate string matching with q-grams and maximal matches,” Tech. Rep., 1991.

9 Appendix

Theorem 1 *The MCES algorithm computes the minimum cost edit script S and does it in time $O(N)$ where N is the maximum number of nodes in a level of T_x and T_y.*

Proof: *To proceed with the proof we simplify the individual edit case analysis as follows. We take $N = max(m, n)$ instead of m, n, and use $C_{ed} = c_e + c_d$.*

With this simplification the required time for update, insert and delete cases are $pO(NC_{ed})$, $pqO(Nc_e)$, $pO(Nc_e)$ and $|X|pq$ respectively. p, q and $|X|$ refer to constant amount of work by the algorithm. As such combining the later three

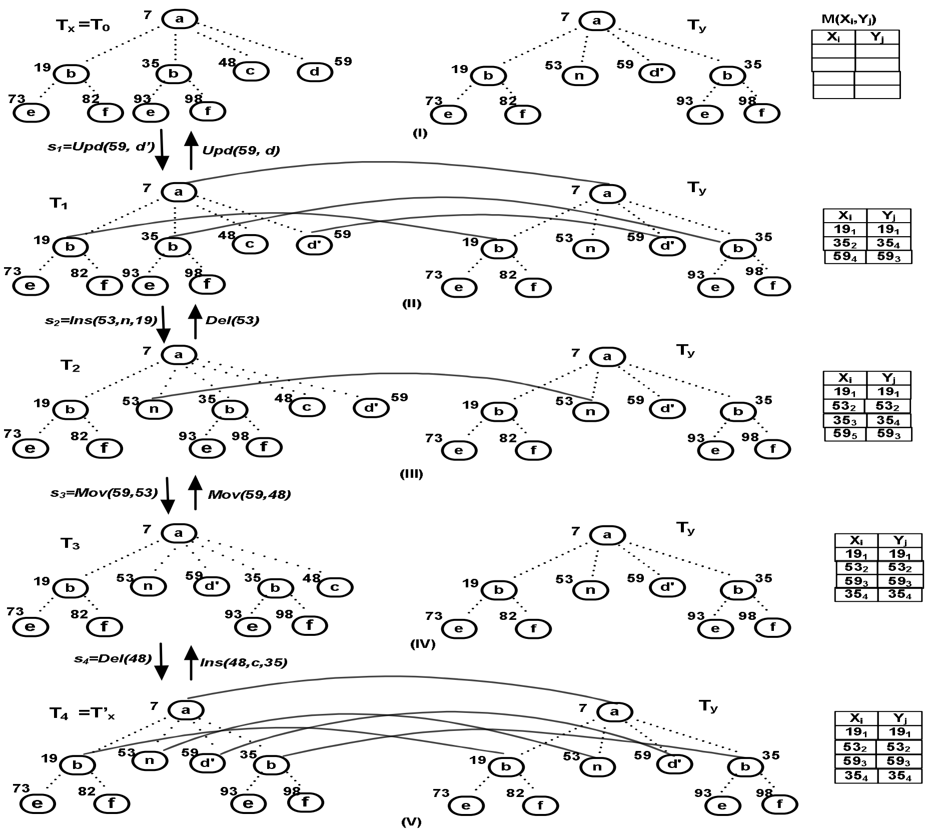


Figure 9: (I) The tree T_x to be transformed to T_y. (II,III,IV,V) The transformed trees T_1, T_2, T_3, and T_4 after edit operations $Upd(59, d')$, $Ins(53, n, 19)$, $Move(59, 53)$, and $Del(48)$ respectively and $T_4 = T_y$.

cases we get the time proportional to $O(Nc_e)$ which then can be combined with the update case time and the total time required for the algorithm is $O(NC_{ed} + Nc_e)$. Further simplifying C_{ed} and c_e as being constant time the algorithm takes $O(N)$ in total.

We now show that S is a minimum cost edit script. Any edit script computed by MCES must contain at least one of the followings:

- *one update operation for each node $x_i \in T_x$ such that $y_j = exist(x_i, -, U, -)$ and $val_{x_i} = val_{y_j}$;*
- *one insert operation for each unmatched node in T_y;*
- *one delete operation for each unmatched node in T_x; and*

- *one move operation for each pair of matched nodes* $(x_i, y_j) \in M$ *such that* $i! = j$*(considering leaf nodes).*

This is straightforward that MCES generates one edit operation for each of the above mentioned operations. Now, considering the nodes having direct children stored in the FIFO queue to be moved using fewest sequence of moves. The ArrangeSibling function that in turn uses the longest sibling sequence LSS conditions ensure such a shortest sequence. Hence, S *is a minimum cost edit script.*

Theorem 2 *The edit script* S *that is generated by the MCES algorithm transforms* T_x *to an isomorphic tree of* T_y*.*

Proof: *We prove this by edit cases during one breadth first traversal of one level of nodes* $x_i \in l(T_x)$ *and* $y_j \in l(T_y)$ *inductively. Recall one level of nodes are stored in the respective FIFO queues. In the update case a first set of matching pairs* $M = (x_i, y_j)$ *are determined for all nodes* x_i *with the* y_j *that have same EBOL.*

The unmathced nodes of $l(T_x)$ *are deleted from* T_x *in the delete case and similarly the unmatched nodes in* $l(T_y)$ *are inserted in* T_x *in insert case. At this point, the* $l(T_x)$ *contains exactly the same nodes of* $l(T_y)$*. The missarragned nodes get arranged in the move case. As nodes are deleted, inserted and moved the respective sibling positions are ensured by the invoked UpdateMatch and FindSibling functions. So, the algorithm transforms one level of* T_x *to isomorphic to a level of* T_y*.*

Inductively, for the further levels of comparison similar arguments apply. So after the algorithm runs for all the levels of T_x *and* T_y*,* T_x *is transformed into an isomorphic tree of* T_y*.*