A Secure Comparison Technique for Tree Structured Data

Mohammad Ashiqur Rahaman
SAP Research
805, avenue Dr. Maurice du Donat
06250, Mougins, France
mohammad.ashiqur.rahaman@sap.com

Yves Roudier
EURECOM
2229, route des Crêtes
06560 Valbonne, France
Yves.Roudier@eurecom.fr

Andreas Schaad
SAP Research
Vincenz-Priessnitz-Str. 1, 76131, Karlsruhe
Germany, +49/62 27/78-43082
andreas.schaad@sap.com

Abstract

Comparing different versions of large tree structured data is a CPU and memory intensive task. State of the art techniques require the complete XML trees and their internal representations to be loaded into memory before any comparison may start. Furthermore, comparing sanitized XML trees is not addressed by these techniques. We propose a comparison technique for sanitized XML documents which ultimately results into a minimum cost edit script transforming the initial tree into the target tree. This method uses encrypted integer labels to encode the original XML structure and content, making the encrypted XML readable only by a legitimate party. Encoded tree nodes can be compared by a third party with a limited intermediate representation.

1. Introduction

Detecting changes in tree structured data has many applications such as aggregation of similar XML databases, difference queries, versioning, or merging of documents. When dealing with sensitive data, such as some organizational strategy, marketing, or financial transactions, a third party performing the comparison should preferably not be allowed to identify the XML structure or content. Although comparison techniques for tree structured data and the generation of a *minimum cost edit script* using intermediate normalized trees have been extensively studied [3, 4, 9], they fall short with respect to (1) enabling a partial comparison of large XML documents due to the memory footprint of the trees and their intermediate normalized forms; and (2) protecting sensitive data: a comparison is typically assumed to be performed by a trusted party.

The differences between two versions of hierarchically structured data can be described by a *minimum cost edit script* that is a sequence of edit operations performed sequentially over the initial tree. A well known approach to finding such a script is to have initial matches of node pairs computed over the full trees of two versions in memory for which comparison functions and approximations are further applied [4]. However, this requires parsing both the source and normalized trees multiple times (i.e. in pre-order, in-order, post-order) and requires more time and memory.

We claim that a comparison technique utilizing *encrypted breadth first order labeling* (or EBOL) [11], a method combining tree and event based parsing addresses these requirements. The comparison technique proposed in this paper has five key characteristics. It addresses *large XML documents*. One such large WSDL of a SAP purchase order can be found in [1]. This schema contains 442 element definitions, of which 36 may occur unboundedly. Existing approaches use at least a full representation of the trees in memory [4], or worse given intermediate representations [19]. EBOL based XML document parsing enables *partial comparisons*. For example, WSDL documents may only differ by an additional service operation resulting in the addition of only few XML nodes. In addition to plaintext XML tree comparison, our technique also supports the protection of *confidential information*. It enables the comparison of encrypted XML nodes without exposing sensitive structural information, like the number of nodes, size of the document, content information such as plaintext values, element names, attribute name-value pairs, or text content. Our approach describes differences based on a sequence (i.e. edit script) of *edit operations* (*update, insert, delete, move*) performed on encrypted tree nodes. The edit operations are defined in terms of siblings rather than parent-child hierarchies so as to enable partial comparison. Finally, our algorithm produces a *minimum cost edit script* (MCES) in a single pass algorithm over two versions of a document.

2. Related Work

Tree comparison techniques, which are generally based on string matching techniques [8, 16, 17, 18] introduce very diverse models. [6, 14, 20] supports insertion and deletion anywhere in the tree whereas in [6], insertion is supported only before deletion. In [12] insertion and deletion of single nodes at the leaf level and updating of nodes anywhere in the tree are allowed. In [4] a subtree movement (bulk operation) for the ordered trees is introduced. [3] introduces

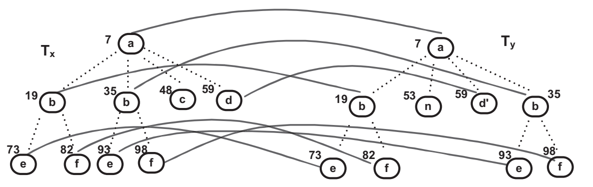


Figure 1. Comparing two trees, T_x(initial) and T_y(edited). Solid lines represent appropriate matches.

techniques for unordered tree comparison including copy operation. We define four atomic edit operations (i.e. update, insert, delete, and move) that can be performed independently except the update operation which needs to be performed before any of the other operations. A comprehensive survey of edit script computation (or tree edit distance) can be found in [2].

Matching algorithms for finding initial matches of node pairs for ordered trees are presented in [13, 20]. The algorithm of [20] runs in $O(n^2 log^2 n)$, which is further improved by [4] into $O(ne+e^2)$, n and e being the number of leaf nodes and 'weighted edit distance' respectively. The *minimum cost edit script* algorithm of [4] runs in $O(ND)$ time, N being the total number of nodes of the two trees and D the number of misaligned nodes. Our algorithm differs in that it does not consider any initial match of node pairs between tree nodes; instead matches are computed as a side effect of the *minimum cost edit script* computation. It runs in $O(N)$ time, N being the maximum number of nodes of the two levels of the source trees [10].

[5] proposes the tree edit distance between two trees should be computed based on the so called 'string edit distance' whereas [19] suggests 'binary branch distance'. Both techniques require intermediate representations of the source trees: two sequences of nodes by pre-order and post-order traversal for [5], two binary tree representations of the source trees for [19]. Our algorithm only requires a FIFO queue storing one level of tree nodes.

[4] describes scenarios in which tree nodes, although ordered, contain unidentified (keyless) data. Tree comparison relies on the semantic but unprotected tagging of the tree. Our EBOL-based parsing technique realizes a similar but protected tagging by associating unique encrypted identifiers with parsed nodes.

3. Solution Model

Consider two XML trees, T_x (initial) and T_y (edited) of Fig 1, each having two levels of nodes, any comparer (e.g. third party) determines *minimum cost edit script* by finding appropriate matches among encrypted nodes of these two trees as shown by the solid lines. One level of a tree T_x is said to be isomorphic [4] to a level of another tree T_y if they are identical except for encrypted node names.

3.1. XML Parsing Model

We parse two XML document tree versions to be compared in breadth first order. For each level of a tree version, we take the sibling nodes (having the same parent) in a FIFO queue and associate an integer pair called breadth first order labels (BOL) to those nodes as these are stored in FIFO order. Each such node, having associated integer pair, captures various structural properties with a minimal memory footprint for hierarchical relationships (i.e. parent-child, siblings, left/right child) of the parsed XML node [10].

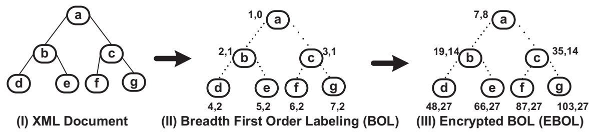


Figure 2. Solid and dotted lines represent explicit memory footprint and no memory footprint respectively. (II,III) are implicit hierarchy representations of (I).

In Fig 2, let a be the parent of nodes b, c. We denote a's BOL as B_a. Let f_{order} and f_{level} be two functions operating on a BOL respectively returning the BOL order (1st attribute of the BOL pair) and BOL depth (2nd attribute). Let b be the last parsed child of a and that c to be parsed next. BOL of c will be: $f_{order}(B_c) = f_{order}(B_b) + 1$; $f_{level}(B_c)$ uniquely identifies the depth level of the node c in a document tree (i.e. $B_c = (3, 1)$).

A BOL is a plaintext and may reveal structure specific information (number of nodes and thus the size of the document), hierarchy relationship among the nodes to an adversary. Encryption over such BOL (i.e. order and depth), denoted as EBOL, by preserving their original order, protects this undesired disclosure [11]. In Fig 2 (III) is the EBOL representation of (II). The EBOL of c is: $E_c = (35, 14)$. For simplicity, following figures skip the depth number.

An EBOL-based parsed XML node in a level has a unique encrypted identifier and 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 edit operations.

By storing the parsed children of a node in a FIFO, EBOL-based parsing implicitly preserves a node's hierarchy information that allows to define reach edit operations based on solely node's sibling relationship. For example, in Fig 4, 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 $\xrightarrow{II}$. Consequently, we can delete the internal node b (i.e. (*Del(19)*)) without deleting its children. Moreover, the memory required for parent-child relationship of b and its children; and their sibling relationship can be freed as the sibling nodes, i.e. children, are stored in sibling order in the queue. Similarly, an internal node can be moved as its children are queued for the next level parsing.

3.2. Edit Operations Model

We refer to a node x's encrypted name value with the val_x and to an EBOL-based parsed XML document with

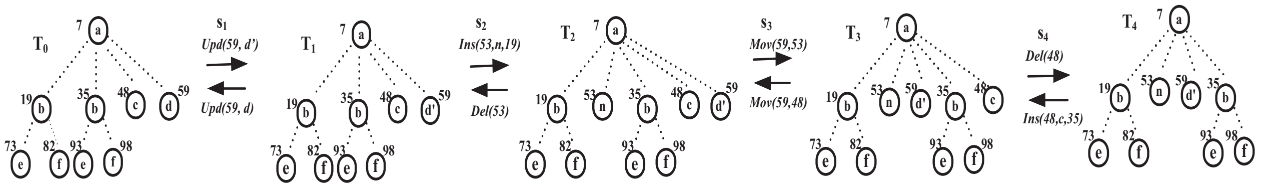


Figure 3. Basic edit operations on encrypted tree structured data.

the tree, T_i. T_{i+1} refers to the resulting tree after performing an edit operation on T_i.

- **Update:** The *update* operation of the value of a node x in T_i, denoted as *Upd(x,val)*, leaves T_{i+1} as of T_i except the value of x is val in T_{i+1}. 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 after the node k of T_i is denoted as *Ins(x,v,k)*. The value v is inserted after k as its immediate 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 3.1. Insertion can be performed after any leaf or internal node. (T_1 and T_2 of Fig 3 for *Ins(53,n,19)*.

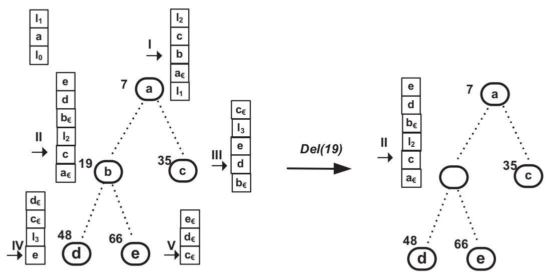


Figure 4. Deleting an internal node, b, (*Del(19)*). The FIFO stores the sibling nodes (d, e) of the 2nd level $\xrightarrow{II}$. The nodes including the dummy nodes in one level are delimited by two l_i entries.

- **Delete:** The *deletion* of a node x from T_i, is 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, is 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)*)

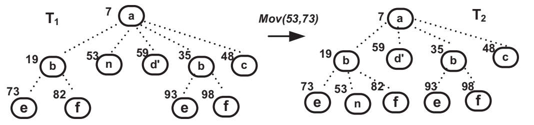


Figure 5. Inter level moving of node n in 1st level of T_1 to 2nd level of T_2.

If a node is moved in the same level then it is an intra level move (as in Fig 3). However, for any inter level move, as in Fig 5, the node n of T_1 is moved after the node e to the lower level, requires different strategy [10]. In particular, for the first level comparison it will be identified as n is deleted whereas it is moved to another level. Intuitively, when a node is moved upwards in a higher level it would be matched for the *insert* case as it is a new node for that level.

3.3. Edit Script and Cost Model

We formalize *edit script* and its *cost model* in similar fashion of [4, 7, 15, 19]. An edit script, S, is a sequence of edit operations when applied to T_0 transforms it to T_i. For 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.

Several edit scripts may transform T_0 into the same resulting tree T_4. For example, the 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.

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 (1) the type of operation and (2) 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. Regarding (2), the cost may depend on the value of 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 [4] 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: $diff(val_x, val_y)$ 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.

4. Determining Edit Script (MCES)

We refer to a level of EBOL-based parsed 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 matched node pairs (x_i, y_j), where $x_i \in T_x$ and $y_j \in T_y$ for $i, j \in \mathbb{N}$. We define a function $exist(args)$ when applied on a tree T_y (or array M), returns the val_y (or TRUE) if E_x matches E_y of a 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. The function $exist(args)$ takes arguments depending on the invoking edit operation.

4.1. Appropriate Matching

We assume two root nodes match without loss of generality. We want to find appropriate matching pairs during the execution of the MCES algorithm rather than finding initial matches and then updating those. The rationale is: (1) Initial match finding requires parsing the large XML documents and their normalized forms into memory before any comparison may start which is undesirable in our context. (2) Partial comparison requires appropriate matching of sibling nodes without knowing their descendants. (3) The matching should be performed over encrypted values as opposed to plaintext values. For (1), we utilize the EBOL based parsed nodes of a level as a first class values for comparison. For (2), we define matching criteria for a node that do not require comparing descendant nodes except its direct children that are stored in the queue. For (3), matching criteria are applied over the encrypted values of XML nodes. A first criterion determines whether an attempt to match nodes should be made based on the similarity of their encrypted values.

Criterion 1: Sibling nodes $x \in T_x$ and $y \in T_y$ match only if $E_x = E_y$.

Given the first criterion is fulfilled, the function $diff(val_x, val_y)$ is called to check whether nodes have been updated. As we rely on symmetric and deterministic encryption, this check is merely matching the corresponding ciphertexts. Further to this verification, a second criterion is applied: two nodes match only if their direct children also potentially match. We define two functions $same(x, y)$ and $max(|x|, |y|)$ where x and y are the nodes to be compared and $|x|$ and $|y|$ are their number of children. $same(x, y)$ returns the number of child nodes having the same EBOL and $max(|x|, |y|)$ returns an integer representing the maximum number of child nodes of the two nodes.

Criterion 2: Sibling nodes $x \in T_x$ and $y \in T_y$ match only if $\dfrac{same(x,y)}{max(|x|,|y|)} > t; \quad where \quad 0 \leq t \leq 1.$

t is a threshold value 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 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 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 similar nodes of a level of a tree with a level of another tree is not smaller than that of dissimilar nodes. 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 differ mostly in the same level.

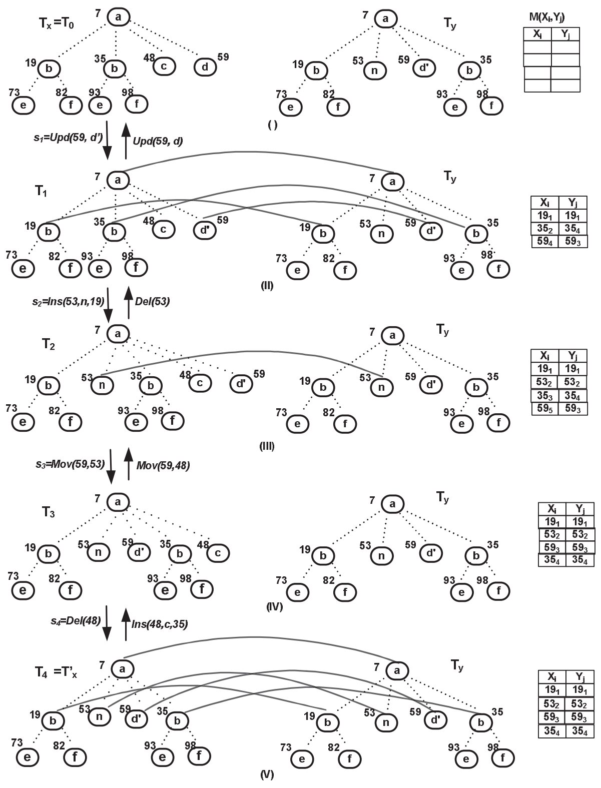


Figure 6. (I) The tree T_0 is transformed to T_4 which is isomorphic to T_y. (II,III,IV,V) The transformed trees T_1, T_2, T_3, and T_4 after edit operations $Upd(59, d')$, $Ins(53, n, 19)$, $Move(59, 53)$, and $Del(48)$ respectively.

4.2. MCES Algorithm

The algorithm, shown in Fig 7, takes one level of tree nodes from T_x and T_y and combines all the edit cases in one breadth-first traversal of T_x and T_y. It makes use of auxiliary functions *exist*, *UpdateMatch*, *ArrangeSibling*, and *FindSibling* described in Fig 8 and detailed in [10]. 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. Fig 6 illustrates how to determine a *MCES* that transforms $T_x = T_0$ to $T'_x = T_4$ by finding appropriate matches.

```
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 misarranged 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)*

```
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 $\forall x \notin L_{xy}$; /*misarranged 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.

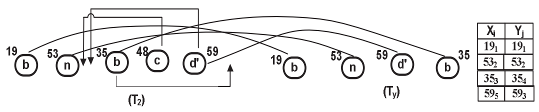
Update Case. For each node x of T_0 the function $exist()$ is invoked to find whether x_i exists in T_y. If so, the function returns val_y, then the function $diff(val_x, val_y)$ is called. If a nonzero value is returned, we add the edit operation $Upd(x, val_y)$ to S, and a matched pair (x_i, y_j) to M. Consequently, we apply the update operation to T_0. Ultimately, T_0 is transformed 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). Even if there is no updated node in T_y meaning a 0 is returned from $diff$, M may have pairs in which each peer has a corresponding matched node in the other tree. Fig 6(II) shows that applying $Upd(59, d')$ to T_0 results into T_1. Fig 6(II) also shows the matching node pairs in M.

Insert Case. To find the inserted nodes in T_y, we take the nodes, w of T_y where 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 the encrypted value val_w. The position k is determined with respect to the sibling position 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. Fig 6(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 misarranged. In Fig 6(III) nodes $35, 59$ in T_2 are misarranged 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 4.3. In Fig 6(III), a $Mov(59, 53)$ is added to S, and applied to T_2 to transform it to T_3 (Fig 6(IV)). Note that no new match is found by this operation, however the sibling positions of $35, 59$ are 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, we can safely conclude the former. Accordingly, we can add operation $Del(x)$ to S which in turn changes the existing sibling positions in M as insertion and move cases. Fig 6(V) shows the resulting tree T_4 after performing $Del(48)$ on T_3.

When the algorithm runs for the first level, the *MCES*



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

Figure 9. Appropriate matching by rearranging sibling nodes.

$S = (Upd(59, d')$,$Ins(53, 19)$,$Move(59, 53)$,$Del(48))$ is generated that transforms T_0 to T_4 which is isomorphic to T_y with respect to first level and M contains the matched pairs nodes of that level (Fig 6). 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 6 the tree T_4 happens to be isomorphic to T_y for the second level also.

4.3. Rearranging Sibling Nodes

As mentioned in the move case (Fig 6(III)) there might be misarranged peer nodes in M. In Fig 9 (shows the siblings of Fig 6(III)), there are at least three sequences of moves to arrange the sibling nodes of T_2 to transform to T_3: (1) moving nodes c and d' after n in that order. (2) moving the node b after d'. (3) 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. In case of several sequences having the same number of moves any one can be picked. To ensure the edit script incurs minimum cost, the shortest sequence of moves to arrange the siblings is determined by finding the longest sibling sequence (LSS) of nodes complying with the second criteria of Section 4.1 (see [10] for details).

5. Conclusion

We have provided a comprehensive technique and an algorithm to compare sanitized tree structured data and generate a *minimum cost edit script*. We showed how to achieve partial comparison over such trees without memory intensive intermediate representations. While the solution is geared to sanitized XML data it is equally applicable to any plaintext tree.

References

[1] A Purchase Order WSDL Document, SAP Enterprise Services Workplace, http://esoadocu.sap.com/socoview(bd1lbizjptgwmczkpw1pbg==)/get_wsdl.xml?packageid=dbbb6d8aa3b382f191e0000f20f64781&id=0afcbb068cee3d59a67b420bc73f2f1b.

[2] P. Bille. A survey on tree edit distance and related problems. *Theoretical Computer Science*, 337:217–239, 2005.

[3] 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*, pages 26–37, New York, NY, USA, 1997. ACM.

[4] 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*, pages 493–504, New York, NY, USA, 1996. ACM.

[5] S. Guha, H. V. Jagadish, N. Koudas, D. Srivastava, and T. Yu. Approximate xml joins. In *SIGMOD '02: Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 287–298, New York, NY, USA, 2002. ACM.

[6] 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*, pages 75–86, London, UK, 1994. Springer-Verlag.

[7] 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*, pages 195–206. VLDB Endowment, 2007.

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

[9] A. Nierman and H. V. Jagadish. Evaluating structural similarity in xml documents. pages 61–66, 2002.

[10] M. A. Rahaman and Y. Roudier. An Efficient Comparison Technique for Sanitized XML Trees. Technical Report RR-09-229, Eurécom, 05 2009.

[11] M. A. Rahaman, Y. Roudier, P. Miseldine, and A. Schaad. Ontology-based Secure XML Content Distribution. In *IFIP SEC 2009, 24th International Information Security Conference, May 18-20, 2009, Pafos, Cyprus*, May 2009.

[12] S. M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977.

[13] 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*, pages 117–126, New York, NY, USA, 1989. ACM.

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

[15] E. Ukkonen. Approximate string matching with q-grams and maximal matches. Technical report, 1991.

[16] 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*, pages 218–223, New York, NY, USA, 1975. ACM.

[17] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *J. ACM*, 21(1):168–173, 1974.

[18] S. Wu, U. Manber, G. Myers, and W. Miller. An o(np) sequence comparison algorithm. *Inf. Process. Lett.*, 35(6):317–323, 1990.

[19] 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*, pages 754–765, New York, NY, USA, 2005. ACM.

[20] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989.