Querying Encrypted XML Documents

Ravi Chandra Jammalamadaka, Sharad Mehrotra
Donald Bren School of Information and Computer Sciences
University of California, Irvine, CA 92697, USA
$\{rjammala, sharad\}@ics.uci.edu$

Abstract

This paper proposes techniques to query encrypted XML documents. Such a problem predominantly occurs in "Database as a Service" (DAS) architectures, where a client may outsource data to a service provider that provides data management services. Security is of paramount concern, as the service provider itself may be untrusted. Encryption offers a natural solution to preserve the confidentiality of the client's data. The challenge now is to execute queries over the encrypted data, without decrypting them at the server side. In this paper we develop: 1) primitives using which a client can specify the sensitive parts of the XML documents; 2) mechanisms to map the XML documents to encrypted representations that hides sensitive portions of the documents; and 3) techniques to run SPJ (Selection-projection-join) queries over encrypted XML documents. A strategy, where indices/ancillary information is maintained along with the encrypted XML documents is exploited, which helps in pruning the search space during query processing.

1 Introduction

Over the past decade, with the explosive growth of the internet, networking, and computing technologies, the software industry has witnessed the emergence of the software as a service paradigm. In the software as a service model clients, instead of installing, maintaining and running software on their computer systems, can purchase the software usage on a "rent an application" basis from software service providers. Motivated by the software as a service, recent database research has explored the viability of the "database as a service" (DAS) paradigm [1, 4, 5] in which clients store their databases on a remote service provider that offers full-blown data management services (storage, query, administration, backups, recovery, etc.). Many research and technological challenges in supporting DAS have been identified, the primary of which is the issue of data privacy and security. Since in the DAS model, the client's data, some of which may be sensitive, resides at the service provider outside the security perimeter of the client, mechanisms to prevent data misuse have to be developed. Encryption is the natural answer, but that raises a fundamental (and difficult) challenge of how to execute queries over encrypted data. Many innovative proposals addressing this challenge have recently emerged [1, 5, 4] in which the client and server collaboratively process queries. As much of the query as is possible to execute without decryption is executed in the untrusted domain, and when further processing is not possible without decryption, the data is brought to the trusted side and converted to plaintext. Techniques to handle many different classes of queries (selection, joins, aggregation) have been previously studied. Much of this has considered outsourcing in the context of the relational data model.

This paper focuses on outsourcing data in the context of the XML databases. XML data, beside content, also consists of a rich internal structure. A client, besides (or instead of) hiding the attributes and elements of the XML document may also desire to hide relationships among nodes. Our first contribution is to develop simple, yet powerful, encryption primitives using which clients (data owners) can specify a rich class of security policies for XML data. Encryption primitives are described in section 2. Our second contribution is to show how the security policies of the client are cryptographically enforced and the encrypted documents are hosted at the server. Server side encrypted XML storage model is described in section 3. To facilitate query processing, motivated by the strategy in [1], ancillary information along with the encrypted XML documents is stored. Instead of exactly utilizing the approach in [1], we develop a novel strategy of *multi dimensional partitioning* which overcomes some of the security limitations of the approach in [1]. Section 4 deals with ancillary information. This is our third contribution. Our fourth and last contribution is to show how a client side query is transformed to a server side query which can be executed at the server side. The

main motivation is to push the majority of the query processing work to the server side. Evaluation of the server side query results in a superset of encrypted results which are shipped back to the client. The client decrypts and filters out the relevant results. Query translation is handled in section 5. We developed an initial prototype to experimentally validate the techniques/mechanisms proposed in this paper. Due to space restrictions we report the experimental results in the full version of the current paper [16].

2 Encryption primitives for XML documents

This section briefly describes the encryption primitives using which a client can specify security requirements over both the structure as well as the content of XML documents. The client may not wish to protect the XML data in its entirety and may choose instead to protect only the sensitive information. For instance, in an XML document containing the name, address and credit card information of customers, the client may wish to protect the credit card information, but leave the address information unencrypted. There are performance benefits of partial encryption due to reduction in amount of decryption at the client side.

The encryption primitives we use in this paper have similarities with client based access control models on XML documents[12, 13]proposed in the literature. The primary purpose of the encryption primitives is to provide data confidentiality to the client, while the access control models were developed to provide access to different views of the XML documents to different recipients. The client's data confidentiality needs can be modeled using XML access control rules where the client wants to provide a partial view of the XML documents to the service provider. The encryption primitives described in this section provide more functionality to the client to express complex security policies than that are possible using the access control models proposed previously. During the course of the section, we will where applicable, highlight the similarities between primitives we use and the XML access control models.

Specifically three primitives E_S(Encrypt structure), E_V(Encrypt Value), E_T(Encrypt Tag) are explored, using which the client can specify fairly complex security policies over the structure, content and the metadata of the XML documents. These primitives provide the user the power to strike an appropriate balance between security and performance. These primitives can be specified either on the schema of the XML documents or on the individual XML documents themselves. In this paper we assume that encryption primitives are specified on the XML schema. These primitives transform the original XML schema to a new server side schema to which all the encrypted XML documents adhere to. We will now individually explain the affect of each of these primitives. E_V primitive when specified on a node *n*, encrypts the *subtree(n)* (both the content and tag of node n), and replaces it by an encrypted node. This primitive is consistent with the W3C recommendation for encrypting XML documents[2]. Since *subtree(n)* contains other elements in its structure, the *subtree(n)* is streamlined into text and this streamlined text is encrypted. Consider fig 1 which demonstrates a Paper XML schema on which an E_V primitive is specified on the *author* node. Fig 3 shows the aftereffects of such a specification. E_T primitive encrypts the content of the tags of the XML documents. When specified on nodes which are not the leaf nodes, the primitive cascades all the way down to the leaf level and encrypts the tags which belong to *subtree(n)*. Access control models for XML operated at the node level. Semantically, prohibiting/providing access to a node meant prohibiting/providing access to all the nodes belonging to the subtree of the node. These semantics were dictated by the containment relationships inherit in the XML documents. The E_V mimics the functionality of such access control models.

In some situations, it becomes more useful to hide the relationship shared between nodes than explicitly encrypting either the tags or content. The client could prefer to publish the information publicly, as long as the relationship between some nodes is hidden. Consider the XML schema introduced in fig 1. For blind reviewing[1] to be successful, only the relationship between the *author* and the *paper* nodes needs to be hidden. The community for a conference is well known in most cases. The client may be willing to publish the author name and email information as long as the relationship between the *paper* node and the *author* node is hidden. $E_S(parentNode, childNode)$ primitive achieves the above objective by fragmenting the original XML document. Fig 2 shows the affect of $E_S(paper, author)$ primitive on the XML schema introduced in fig 1. If fragmentation was not followed, the other option would have been either to encrypt the *subtree(Author)* or encrypt *subtree(Paper)- subtree(Author)*. By fragmenting the document, queries such as "find the email of Jeff Ullman" can be executed on the documents stored at the server. In fact, query that can be answered by *subtree(Author)* can be executed at the server. Reader should note that XML documents stored at the server side do not exactly correspond to the changes described in this section due to the primitives. The changes demonstrated in this section due to the primitives describe the ideal view that the client wants the server to see. For facilitating query processing, some additional

[1] A process used by some conferences to keep the name of the author secret from the reviewer and vice versa

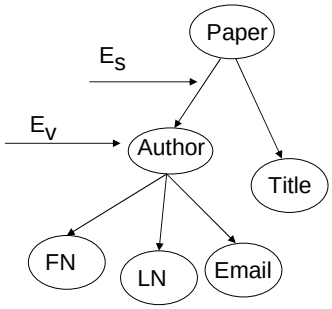

Figure 1. Original XML file with Encryption primitives specified on them

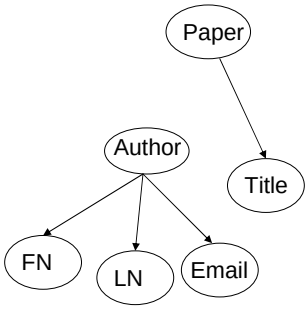

Figure 2. Affect of the E_S primitive

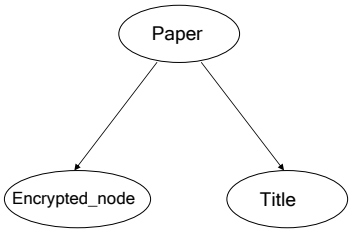

Figure 3. Affect of the E_V primitive

metadata will need to be stored. This will be explained in more detail in the next section.

To the best of our knowledge no research work has explored hiding relationships between nodes (i.e. E_S primitive) and hiding metadata (i.e. E_T primitive) of XML nodes in the context of XML data outsourcing. Note that functionality provided by primitives can be used to develop a comprehensive XML access control model. Such a study is outside the scope of this document.

3 Encrypted XML storage model

The previous section developed primitives on the XML schema using which the client can specify the security policy on the XML schema. This section develops a mapping that takes as input, the XML schema as well as the security policy specified using the primitives developed in section 2 and outputs a server side XML representation that facilitates query processing [2]. In our approach similar to [1], as well as the work on secure indexes for enabling keyword search on encrypted data [14, 15], the client stores ancillary information at the server to facilitate query processing. However, the specific ancillary information stored differs from [1] and this will become clear in the following sections.

Fig 4 illustrates our overall mapping strategy. For an XML schema, all the E_V primitives are handled first. When the client specifies $E_V(n)$ primitive ideally he/she desires to replace *subtree(n)* with an encrypted node. In the approach proposed in this paper the *subtree(n)* is replaced by $estub^m$, whose structure is shown in fig 6. Let L = $\{ L_1, L_2 \ldots L_N \}$ be the set of leaf nodes that are affected by the specification of $E_V(n)$ primitive, i.e all the leaf nodes that belong to *subtree(n)*. Node E($L_i \ldots L_N$) stores the the encrypted string of the concatenation of all the leaf node values. Nodes $Ancillary^1(L_V)$ and $Ancillary^2(L_{nv})$ store the ancillary information required for querying on the leaf nodes in L. L_V is the set of leafNodes which do not have a "*" or "+" from the path from the node where the E_V primitive is imposed, to themselves. Set L_{nv} contains the other leaf nodes which have a "*" or "+" in the path from the node where the E_V primitive is imposed to themselves. For the nodes in L_{nv} their cardinality is not known. The content of this ancillary information is explained in the next section.

[2]The server can store the encrypted XML documents using a native XML database or an RDBMS using the translation techniques proposed in [6, 9]. Our techniques are independent of the type of the database employed at the server

Input:
XML schema S of the unencrypted XML documents
E_{Prim} = $\{ E_1, E_2 \ldots E_p \}$ where $E_i \in \{ E_S, E_V, E_T \}$
Mapping:
1. For every $E_V(n)$ primitive $\in E_{prim}$ {
Replace *subtree(n)* with $estub^m$ }
2. For every $E_S(parentNode, childNode)$ primitive $\in E_{prim}$ {
Fragment the schema into two different trees and create
node *id* as the child of *parentNode* and nodes *parentid*
and *pid* as the children of *childNode* }
3. For every $E_T(n_1)$ primitive {
Encrypt the tag values of all the nodes $\in subtree(n_1)$ unless
the node is an encrypted node obtained from step 1. }

Figure 4. Schema mapping

The E_S primitives are handled next. $E_S(parentNode, childNode)$ fragments the original document tree structure into two different trees, one that ends at *parentNode* and another that is rooted at *childNode*. This implies isolation of subtree rooted at *childnode* from the original document. Two nodes *id* and *parentid* are created, children of *parentNode* and *childNode* respectively. When *id* = *parentid*, it implies that both these nodes belong to the same document. Both the *id* and *parentid* nodes are self generating attributes (i.e. automatically generated

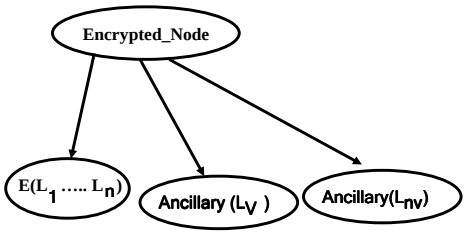

Figure 6. $estub^m$

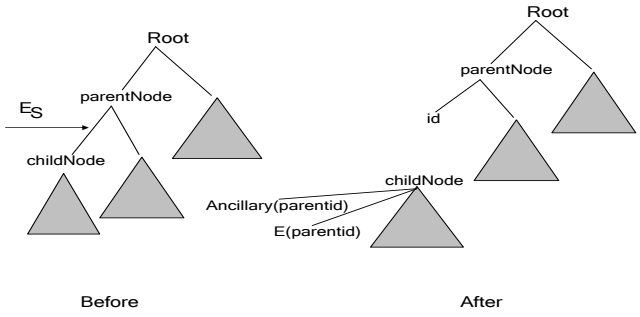

Figure 7. Affect of the E_S primitive

Input:
XML query Q represented as $< Q.T, Q.F >$
XML schema S of the unencrypted XML documents
E_{Prim} = { $E_1, E_2 \ldots E_p$ } where $E_i \in$ { E_S, E_V, E_T }
Output:
Q^S = { $< Q_1^S.T, Q_1^S.F >, \ldots < Q_N^S.T, Q_N^S.F >$ }

QueryTranslation:
1. $< Q.T^{'}, Q.F >$ ← unravel($Q.T$)
2. S ← Schema_Mapping($< Q.T^{'}, Q.F >$, E_{prim})
where S = { $< Q_1.T, Q_1.F >, \ldots < Q_N.T, Q_N.F >$, }
3. For every pattern tree $< Q_i.T, Q_i.F >$ from step 2 {
For every predicate in $Q_i.F$ {
Translate the predicate to the server side predicate } }

Figure 5. Query Translation

by the client [3]) and one of them needs to be encrypted to enforce the E_S primitive and hide the relationship between *parentNode* and *childNode*. The content of *parentid* is encrypted as the node *parentNode* could potentially have more than one child. Another node *Ancillary(parentid)* is created, which stores ancillary information required to process a join operation between the nodes *parentNode* and the *childNode* at the server. The content of *Ancillary(parentid)* and its use is explained in the next section. Fig 7 shows the effect of the E_S primitive.

We have previously shown how $E_T(n)$ primitive encrypts tag value and this primitive cascades down to the leaf level unless it encounters an already encrypted node. No other changes are required to this mapping to facilitate query processing.

4 Ancillary Information

This section explains the content of ancillary information stored at the server to support query processing. The ancillary information stored when the E_V primitive is imposed (i.e. the content of the nodes $Ancillary^1(L_V)$ and $Ancillary^2(L_{nv})$) is discussed first. This is discussed under two situations: a) when the *subtree(n)* effected by the E_V primitive does not contain multivalued operators (i.e "*" or "+" operator); and b) when the *subtree(n)* affected by the E_V primitive contains multivalued operators. After the above two cases are handled, we will discuss the ancillary information stored for the E_S primitive.

Subtree without multivalued operators: In this situation ancillary information is only stored in the node $Ancillary^1(L_V)$. Let the set L = { $L_1 \ldots L_N$ } be the set of leaf nodes belonging to *subtree(n)*. Since there are no multivalued operators in *subtree(n)*, set L_v (all the leaf nodes which do not have a "*" or "+" from the path from the node where the E_V primitive is imposed, to themselves) is equal to L. Let dom(L_i) represent the domain of the leaf node L_i. Let dom(L) = dom(L_1) × dom(L_2) × ... dom(L_N) be the cartesian product of all the domains of leaf nodes elements in L.

The domain of L can be viewed as a N-dimensional space where each leaf node L_i corresponds to a dimension. This N-dimensional space is partitioned into a set of partitions and associated a random identifier for each partition. The partitions should cover the whole domain and should not overlap. While any partition multi-dimensional histogram based technique[11] could be used to partition the multidimensional space, the choice of specific partitioning policy has security implications. An attacker can analyze the frequency distribution of the partition identifiers to gather information of the underlying data. Therefore, we propose the usage of the equi-width multidimensional partitioning strategy to combat against such frequency attacks. Detailed security analysis of our partitioning strategy can be in the full version of the paper[16]. Due to space restrictions we do not report the results here. The knowledge of the partition boundaries and extent associated with a partition identifier if revealed to the adversary will result in information leakage. Thus, the partitioning information including the mapping of the partition to the partition identifier is stored at the client and hidden from the server.

Table 8 shows an example multi dimensional partition-

[3]The values of such self-generating attributes should be unique for different for XML document instances.

Id	FN	LN	Email	Partition Identifier
0-15	0-700	0-600	0-900	28
15-30	0-700	0-600	900-1600	99
30-35	0-700	0-500	160000-1800	5000
…	…	…	…	…

Figure 8. Partitioning the *subtree(Author)* domain

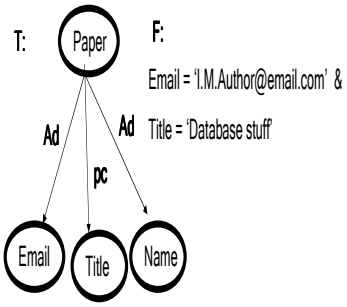

Figure 9. Example of a Pattern

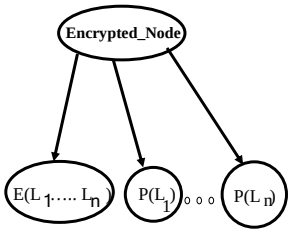

Figure 10. $estub^s$

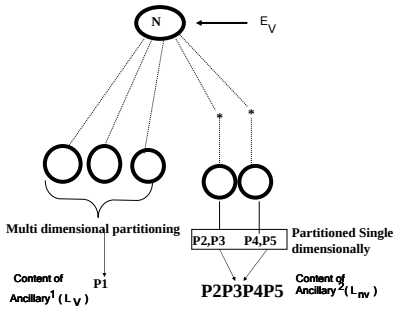

Figure 11. Construction of the partitioning information

ing strategy. Note that string dimensions (FN, LN, Email) have to be first mapped to an integer domain using hashing techniques. The unencrypted leaf node values of *subtree(n)* are points in the multi-dimensional space corresponding to dom(L). The partition identifiers of these points are stored as the content of the node $Ancillary^1(L_V)$ along with the encrypted information i.e. E($L_1 \ldots L_N$). The ancillary information maintained in this paper is used for processing Selection-projection-join queries only.

Note that we could have used the strategy in [1] where the domain of every leaf node $L_i \in subtree(n)$ is partitioned and partition identifier is stored as the ancillary information. If we were to follow this strategy then *subtree(n)* will be replaced by the $estub^s$ which is shown in fig 10. $P(L_i)$ stores the partition identifier for leaf node L_i. There are two reasons for adopting $estub^m$ instead of $estub^s$: a) Multi dimensional partitioning is more secure than the single dimensional partitioning (The relative relative security merits of multidimensional and single dimensional partitioning schemes is discussed in the full version [16]), and b) $estub^s$ cannot handle the case when *subtree(n)* contains a multivalued operators (i.e. "* " or "+").

Subtree with multivalued operators: In this situation ancillary information on both the nodes $Ancillary^1(L_v)$ and $Ancillary^2(L_{nv})$ is stored. The contents of set L_V and L_{nv} have been explained in section 3. All the nodes in L_V can now be treated as a dimension and this space can be multi dimensionally partitioned using the strategy we discussed above. The partition identifier is stored as the content of $Ancillary^1(L_V)$.

Now we will explain the content of the $Ancillary^2(L_{nv})$ node. For nodes in L_{nv} the cardinality of the leaf nodes is not known. It is difficult to use the multi dimensional partitioning strategy for set based elements since the cardinality of the leaf nodes is not fixed. We could have mined the maximum cardinality of the leaf nodes and used it to partition the content space, but such an approach is not scalable. For all leaf nodes in L_{nv} we will partition their multiple values single dimensionally and store the string concatenation of all the partitions as the content of the $Ancillary^2(L_{nv})$. Fig 11 illustrates the construction for the content of the nodes $Ancillary^1(L_V)$ and $Ancillary^2(L_{nv})$.

Content of the node *Ancillary(parentid)*: To preserve the privacy of the $E_S(parentNode, childNode)$ primitive, a new node *Ancillary(parentid)* was introduced as the child of the node *childNode*. This section will explain the content of the node *Ancillary(parentid)*.

Definition 1: *Given a parentNode P_1 and any two childNodes C_1 and C_2 , let the function Prob(n_1, n_2) give the probability that n_1 is the parent of n_2 , then the privacy of E_S(parentNode, childNode) is only preserved if Prob(P_1 , C_1) = Prob(P_1 , C_2).*

The above definition explains the requirement for the

preservation of the privacy of the E_S primitive. Two groups of trees can be easily stored at the server. To execute a path query which traverses the edge between *parentNode* and *childNode*, a join needs to be performed between *parentNode* and *childNode*. If the join is performed at the server it would compromise the primitive. An approach to preserve the privacy is to ship all the documents rooted at *childNode* node to the client. This is infeasible for large databases. A similar problem is addressed by the PIR (private information retrival) research [3]. In PIR, the client wishes to retrieve a record from a database which belongs to the server, without revealing any information about the record that has been retrieved. The solutions involve using non colluding duplicate severs or secure coprocessors which scan the whole database to fetch the required data. Both these solutions are impractical in our setting.

Our technique is to ship only a subset of documents rooted at *childNode*. The client can control the number of documents being shipped and this becomes a security parameter. Recall the creation of the nodes *id* and *parentid*. The content of the node *id* is kept unencrypted and the content of the node *parentid* is encrypted. The domain of the node *id* is single dimensionally partitioned as explained before and the partition identifier stored as the content of the node *Ancillary(parentid)*. The metadata regarding the *id* partitioning is stored with the server. When the join has to take place between the *parentNode* and the *childNode*, the partition identifier *p* where the content of the *id* node maps to is found, and the XML documents rooted at *childNode* and having the content of the node *Ancillary(parentid)* to be *p* are shipped to the client. The client can now decrypt all the *parentid* nodes and execute the join operation.

5 Query Translation

This section explores how a client query Q can be transformed into a set of queries Q^S = { Q_1^S, Q_2^S ... Q_N^S }, that can be executed at the server side over the encrypted data representation. The results of all the queries in Q^S will be decrypted and filtered at the client side to compute the actual answer. Our objective is to push the majority of the work to the server side.

The XML query model developed in [10] is used to model XML queries as pattern trees. Pattern trees are pairs $P = (T, F)$ where T is a node labeled tree and F is a boolean combinations of predicates on nodes that belong to T. Fig 9 shows the example of a pattern tree, that corresponds to a query Q seeking the content of *Name* in an XML document where the *email* and *title* of the paper has been specified as "I.M.Author@email.com" and Title = "Database stuff" respectively. This is an example query on the XML database conforming to the schema in fig 1. Every edge of the pattern tree is either labeled as PC (parent-child) and AD (Ancestor-dependency), which describes the relationship between the nodes. For the rest of the paper, XML queries are viewed as pattern trees. Given XML query Q and its pattern tree representation $(Q.T, Q.F)$, our objective is now is to map the query to a set of server side queries Q^S = { $< Q_1^S.T, Q_1^S.F >$, $< Q_2^S.T, Q_2^S.F >$, ... $< Q_N^S.T, Q_N^S.F >$}, such that queries in Q^S can be evaluated over the encrypted XML representation. Note that all the queries in Q^S are sent to the server at once without needing multiple rounds of communication with the client.

Intuitively, the security primitives can be re applied on Q to get the required set of queries to be executed at the server side. Fig 5 describes our overall query translation strategy. In step 1, the implicit structure hidden in $Q.T$ is unraveled. The edges with AD relationship in $Q.T$ are resolved into a path of PC edges using the original XML schema of the unencrypted documents. This is done to uncover potential nodes hidden in the AD edge that could have encryption primitives specified on them. There could be leaf nodes in $Q.T$ which do not correspond to leaf nodes in the schema of the unencrypted documents. For instance, the node *Name* in fig 9, is not a leaf node in the XML schema in fig 1. The *subtree(Name)* from the original XML schema is now placed under the node *Name* in $Q.T$, to uncover other encryption primitives specified on them. Fig 14 illustrates the pattern tree resolved from the initial pattern tree in fig 9. In step 2, the encryption primitives are re applied on the resolved pattern tree from step 1 to get a set of pattern trees which have the same tree structure as the encrypted XML documents. In step 3, every predicate in the pattern trees derived from step 2 is mapped to the corresponding server side conditions. We will now explain the tree structure mapping and the predicate mapping procedures in more detail.

Mapping the tree structure of the pattern tree: This section describes the individual affect of E_S, E_V and E_T primitives on $Q.T^{'}$ (See fig 5). An E_S(parentNode, childNode) will impact $Q.T^{'}$ only if both the *parentNode* and *childNode* belong to it. If that is the case, Q is split into two pattern trees Q_1 and Q_2, similar to the effect of the E_S primitive on XML schema described in section 2. Node *id* is introduced as the child of the node *parentNode* and nodes *parentid* and *Ancillary(parentid)* are introduced as the children of the node *childNode*. Recall that content of the node *Ancillary(parentid)* is a function of the node *id*, as during encryption of the XML document, the content of the *id* node is partitioned and partition identifier was stored as the content of *Ancillary(parentid)*. Clearly, at the server side, the server should first execute the pattern tree which contains the *id* node, partition the contents of all *id* nodes in the result set (by using the metadata stored locally, see section 4)

and then do a selection on the node *Ancillary(parentid)* for these partitions. The Formula $Q.F$ also needs to be split into $Q_1.F$ and $Q_2.F$, but this is trivial as the pattern tree can only have predicates in its formula on the nodes which are present in its tree structure. For example, the effect of $E_S(paper, author)$ on the pattern in fig 14 is shown in fig 12. The predicate *Ancillary(parentid)* = Part(Q_1.id) in fig 12 is now explained. Function part(Q_1.id) returns the set of partitions to which the content of the *id* node fetched from the execution of Q_1 at the server. These set of partitions are placed as a predicate on the *Ancillary(parentid)* node. The E_S primitive is primarily responsible for splitting the original query Q into a set of server side queries.

The $E_V(n)$ primitive will effect $Q.T^{'}$ if the node *n* belongs to $Q.T^{'}$. If that is the case the *subtree(n)* in $Q^{'}_T$ is replaced by $estub^m$. Fig 13 shows the effect of the $E_V(Author)$ primitive on the pattern tree in fig 14.

The $E_T(node)$ primitive will also encrypt all the nodes in $subtree(n)$ in $Q.T$, using the same encryption keys used during the enforcement of the security primitives.

Mapping predicates: Previously, it was shown how query Q represented as $< Q.T, Q.F >$ is split up into a set of pattern trees $\{< Q_1.T, Q_1.F >, < Q_2.T, Q_2.F >, \ldots < Q_N.T, Q_N.F >\}$. This section deals with the translation of the predicates on each these pattern trees to their corresponding server side predicates. Each predicate in $Q_i.F$, where $i \in \{1, 2, \ldots N\}$, is mapped individually to the server side. The predicates specified on the leaf nodes are typically mapped to predicates on the ancillary information. The predicate could be any one of the following 3 types a) *leafnode.content = Value* b) *leafNode.content < Value* c) $leafNode_k$*.content = *$leafNode_l$*.content, where* $k \neq l$. These are the types of predicates that are possible in the XML query model[10] used in this paper. Notable exception to the predicates mentioned above is the theta-join that is not handled in this paper. We will only handle a subset of these cases in the interest of brevity. It is hoped that this gives the reader enough intuition to understand the condition mapping procedure.

leafNode.content < Value: If the *leafNode* is not encrypted when it was stored at the server, there isn't any requirement for condition mapping. Only if the *leafNode* $\in estub^m$ there is a necessity to transform this condition to the server side condition. If the leafNode $\in estub^m$, and if the path from the node where the E_V primitive was specified to the *leafNode* does not contain a multivalued operator, then the *leafNode* participated in the multi-dimensional partitioning. All the multi-dimensional partitions which contain points less than *Value* in the *leafNode* dimension need to be fetched from the server side. The condition now is transformed onto the $Ancillary^1(L_1 \ldots L_N)$ node. For instance consider the Multi-dimensional partitioning of the *subtree(Author)* domain introduced in fig 8. For the condition $id < 36$, partitions 28,99 and 5000 have at least one value less than 36 in the *id* dimension. The server side condition now is $Ancillary^1(L_1 \ldots L_N) \in \{28,99,5000\}$. If the path from the node on which the E_V primitive to the *leafNode* contains a multi-valued operator, then the *leafNode* was single dimensionally partitioned. A LIKE query is executed on the $Ancillary^2(L_1 \ldots L_N)$ node for all the partitions in the *leafNode* dimension, which contain points less than *Value*. For example if the path from node where the E_V primitive was specified to *leafNode* contained a "*" operator, the conditions $id < 36$ is mapped to $Ancillary^2(L_1 \ldots L_N) \in \{\%28\%,\%99\%,\%5000\%\}$. Where the charecter % refers to zero or more other partition identifiers.

$leafNode_l$*.content = *$leafNode_k$*.content*: There is a requirement to map such a join condition when at least one of $leafnode_l$ or $leafnode_k$'s ancestor node has an E_V primitive imposed on it. The most complicated join condition occurs when both the nodes belong to different encrypted subtrees. Such a case is possible when a join is requested between two different XML documents conforming to different XML schemas. Further complication occurs if both $leafnode_l$ and $leafnode_k$ are multi-valued attributes. We will briefly explain only this particular case, since it is the most complex and should give the reader enough intuition for the other cases. To execute such a join condition the server has to make sure there is a atleast one partition id match between the ancillary information associated with the *estubs*. For instance, let $\{200,40,60\}$ be the ancillary information of the multi-valued attributes associated to $subtree(n_1)$ (i.e. $Anclillary^2L_{nv}$) to which the node $leafnode_l$ belongs. Let $\{40,78,1000\}$ be the ancillary information of the multi-valued attributes associated to $subtree(n_2)$ to which the node $leafnode_k$ belongs. Now the server can check that there exists an intersection between the two sets of partition ids (i.e. id 40) and will conform a potential match exits.

6 Related Work

There have been two previous works in the literature that dealt with processing queries over encrypted XML documents [7, 8]. Schrefl et al[7] propose a technique that allows xpath selection queries to be executed at the server side. The limitations of the proposed strategy that are addressed in our paper are: a) cannot support range queries; and b) requires multiple rounds of communication between the client and the server to answer a single query, thereby potentially undermining the performance. Yang et al[8] propose XQEnc an XML encryption technique based on vectorization and skeleton compression of XML paths. The proposed XML encryption technique is in accordance with W3C standard [2]. To facilitate query processing the authors propose to use any of the existing techniques such as single dimen-

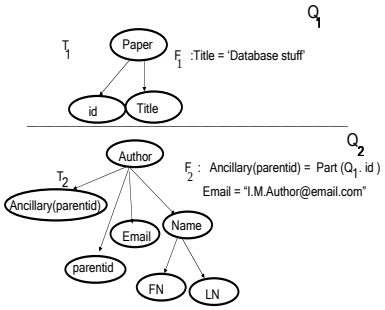

Figure 12. Effect of the E_S primitive on the pattern

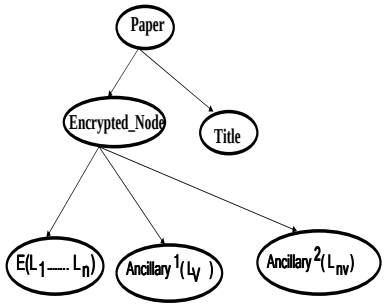

Figure 13. Effect of the E primitive on the pattern

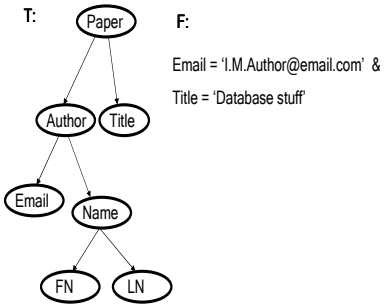

Figure 14. Resolved pattern

sional partitioning or order preserving encryption.

We differ ourselves from the previous work on XML data outsourcing in the following manner: a) We explore a set of encryption primitives using which the client can provide fairly complex set of security policies on the xml documents. More specifically, out of E_S, E_V and E_T primitives explored in this work, only E_V has been handled previously in the literature; and b) We introduce a novel multi-dimensional partitioning technique that facilitates query processing to take place at the server side on the encrypted documents. The multi-dimensional partitioning technique allows range queries to be processed at the server and overcomes the security limitations of the single dimensional partitioning techniques. Previous approaches to XML data outsourcing cannot support range queries.

7 Conclusions

This paper presented techniques to support query processing on encrypted XML data in an outsourced database model. We proposed a set of encryption primitives using which a client can propose fairly complex set of security policies on XML documents. We introduced a novel multi-dimensional partitioning strategy that allows query processing to take place at the server side and overcomes the limitations of the single dimensional partitioning techniques previously proposed in the literature. Some of the techniques proposed here could also be used in the context of relational databases, but such a study is out of the scope of this paper.

References

[1] H.Hacigumus, B.Iyer, C.Li and S.Mehrotra. Executing SQL over Encrypted Data in the Database-Service-Provider Model. *ACM SIGMOD Conference on Management of Data.* Jun 2002.

[2] *http://www.w3.org/Encryption/2001/*

[3] B.Chor, O.Goldreich, E.Kushilevitz and M.sudan 1995. Private information retrieval. *In Proceedings of the thirty-sixth Annual Foundations of Computer Science. IEEE Computer Society Press.* Los Alamitos, California, pp. 41-50.

[4] E.Damiani, S.De Capitani di Vimercati, Sushil Jajodia, Stefano Paraboschi, Pierangela Samarati. Balancing Confidentiality and Efficiency in Untrusted Relational DBMSs. *Proceedings of the Tenth ACM conference on Computer and Communication security.*

[5] L.Bouganim and P.Pucheral. Chip-secured data access: Confidential data on untrusted servers. *In Proc. of the 28th International Conference on Very Large Data bases.* Pages 131-142, Hong Kong, China, August 2002.

[6] J. Shanmugasundaram, K. Tufte, C. Zhang, G. He, D. J. DeWitt, and J. F. Naughton. Relational databases for querying XML documents: limitations and opportunities. *In The Very Large Databases Journal.* Pages 302-314, 1999.

[7] M.Schrefl, K.Grun, J. Dorn. "SemCrypt - Ensuring Privacy of Electronic Documents Through Semantic-Based Encrypted Query Processing," icdew, p. 1191, 21st International Conference on Data Engineering Workshops (ICDEW'05), 2005.

[8] Y.Yang, W.Ng, H.Lam Lau, and J.Cheng. An efficient Approach to Support Querying Secure Outsourced XML Information. Caise 2006.

[9] P.Bohannon, J. Freire, P.Roy, J.Simeon. From XML Schema to relations: A cost based approach to XML Storage. In the proceedings of ICDE 2002.

[10] H. V. Jagadish, Laks V. S. Lakshmanan, Divesh Srivastava, Keith Thompson, TAX: A Tree Algebra for XML, In: Proceedings of 8th International Workshop on Databases and Programming Languages, Rome, Italy, September 2001.

[11] M. Muralikrishna, David J.Dewitt. Equi-depth multidimensional histograms. *Proceedings of the 1988 ACM SIGMOD International conference on management of data.*

[12] E.Damiani, S.C.Vimercati, S.Paraboschi, P. Samarati. Fine-grained Access Control System for XML Documents.ACM Transactions on Information and System Security (TISSEC). Volume 5, Issue 2(May 2002) table of contents. Pages: 169 - 202.

[13] S.Hada, M.Kudo. Provisional Authorization for XML Documents. http://www.trl.ibm.com/projects/xml/xss4j/docs/xacl-spec.html

[14] E.Goh. Secure Indexes. In submission

[15] D. Xiaodong Song, D.Wagner, A.Perrig. Practical Techniques for Searches on Encrypted Data. IEEE Symposium on Security and Privacy

[16] R.Jammalamadaka, S.Mehrotra. Querying Encrypted XML Documents. Technical Report TR-RESCUE-06-15. http://www.ics.uci.edu/řjammala.