

Access Control for XML - A Dynamic Query Rewriting Approach

Sriram Mohan

Arijit Sengupta

Yuqing Wu

Jonathan Klinginsmith

Indiana University, Bloomington, Indiana, USA
{srmohan, asengupt, yuqwu, jklingin}@indiana.edu

Abstract

We introduce the notion of views as a mechanism for securing and providing access control in the context of XML. Research in XML has explored several efficient querying mechanisms. Hiding sensitive data from unauthorized users is as important as supporting efficient querying of visible data. However, given the semi-structured nature of XML data, this is non-trivial, as access control can be applied on the values of nodes as well as on the structural relationship between nodes. In this context, we present an algebraic security view specification language SSX for DBAs to specify security constraints for different user groups. A Security Annotated Schema (SAS) is proposed as the internal representation for the security views and can be automatically constructed from the original schema and the security view specification sequence used to define the security constraint. We also propose a set of rules that can be used to rewrite user XPath queries on the security view into an equivalent XQuery expression that can be executed against the original data, with the guarantee that the users only see information in the view and not infer any data that was blocked. Experimental evaluation demonstrates that our approach is expressive and efficient.

1 Introduction

XML is one of the most extensively used data representation and data exchange formats. The number of XML related applications developed and under development is significant. Much of the research on XML has focused on developing efficient mechanisms to store and manage XML data either as a part of a relational database or using native XML stores. However, hiding sensitive data is as important

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

as making the data efficiently available. For instance, given an XML document tree, it is possible that there are different user groups with varying permissions to access various parts of the document. Any security specification model must ensure that these policies are enforced correctly and efficiently. Given a query q over a secured XML document tree, it is important that the result of the query just contain the nodes that the user has permissions to see, in the context that the user has permissions to see. The enforcement of the policy, commonly denoted as *Access Control* must also ensure that the user not be able to indirectly reference the data through a combination of queries on the view tree. In this paper we propose the notion of views as a mechanism for access control for XML. We introduce a security specification language (SSX) and rules to rewrite user queries to enforce security constraints.

1.1 Challenges

The semi-structured nature of XML data implies that the data is not in a normalized structure and makes the task of defining security views non-trivial. XML data can have duplicate/missing elements and/or missing attributes. Furthermore, the identification of an element is no longer restricted to the value of the element itself (like a record in the relational model) but depends on the context, the form of the path (from the root element to the element) and the children/descendants of the element. In XML, it is frequently the case for the content of a certain element to be visible only to a given user group, or the visibility to be conditional, based on the value/structure of elements outside the sub-tree rooted at the element in question. It is also possible that a given element has differing structures for different user groups. Hence XML Access Control should also consider the structural relationship between nodes.

Another challenge is in the presence of multiple access control policies. With constant change in data, it is expensive to actually materialize and maintain each view that implements a security specification. When such materialized views are used to address the security concerns, the security model has to guarantee that the results returned to the users is exactly what would be returned if the view were materialized at the same instant the query was issued.

Research on XML access control has handled some of the above issues with differing degrees of success and efficiency. Approaches proposed vary from XML access

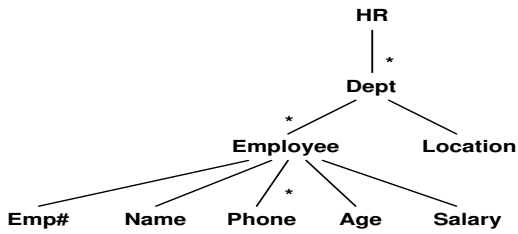


Figure 1: Example Tree Structure for an HR XML Database

control languages [14, 12], cryptography [9], execute-and-check method [11], to materialized security views [1, 5]. The most recent work by Fan et al. [6] introduced an approach that annotated the security restrictions on the schema structure, and rewrites XPath queries issued against the new schema structure into queries against the original XML document. View materialization is thus avoided. However, the expressiveness of the security annotation is limited to hiding node/sub-tree values. Enforcing security constraints based on the structural relationships between elements, which is at least as important as the values, remains an open question in the XML context and is one of the main contributions of this paper.

1.2 Motivating Scenarios

Consider an XML database that contains the human resource information of a university. The schema structure to which the XML documents conform is shown in Figure 1. In the hierarchical structure, the university has multiple departments (**dept**); each department has a list of **employees** and a **location**. The information about each employee includes **emp#**, **name**, multiple phone numbers (**phone**), **age** and **salary**.

Example 1.1 *As one may observe, some of the information about an employee can be considered sensitive, such as **age** and **salary**; and the rest can be made public, such as **name** and **phone**. In case an element is classified as only sensitive or not, the approach proposed by Fan et al. [6] is sufficient, for it would block the access to the sub-trees rooted at **age** and **salary**. (Let’s name this security concern (0) for it be referred in later discussion). However things are not always so simple.*

Example 1.2 *Consider the following security concerns: (1) Users should not know the age and salary of each individual employee in a department, but can access all the age information of a department; (2) Users should not know the age and salary of each individual employee and any statistical information about the age of employees in a department, but can access all the age information in the scope of the university; (3) Users should not know the age and salary of each individual employee, but can obtain information about the salary distribution with respect to employee’s age, in the scope of a department; (4) Users should not know the age and salary of each individual employee, and should not know any connection between the age and*

salary of each individual employee, since they may infer someone’s salary indirectly.

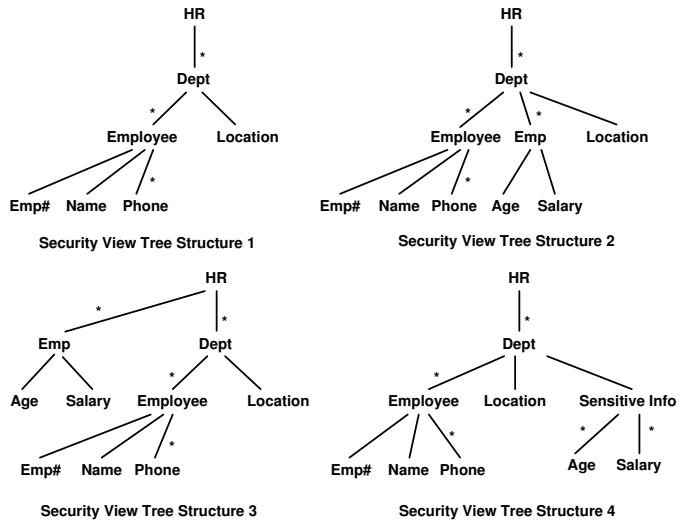


Figure 2: Example Tree Structures for the Security Views of an HR XML Database

Consider the rearrangement of the XML database, shown in the form of tree structures, in Figure 2. Tree₁ addresses security concern (0), but not any of the security concerns (1-4). Security concern (1) is addressed by Tree₂, and security concern (2) by Tree₃. Both Tree₂ and Tree₃ satisfy security concern (3). But none of the schema tree structures in Figure 2, except Tree₄, is strict enough for security concern (4).

All the security concerns listed in Example 1.2 are not uncommon, yet none of the existing techniques support them without actually generating or materializing the “views”. The security view specification language needs to be enriched for DBAs to address these concerns. Defining such a language is the first focus of this paper.

Furthermore all user queries should return only the values and the structures that are allowed to be revealed. One way of achieving this is to materialize each such security view and evaluate the user queries directly on the materialized data. Given that there may be multiple security levels, each requiring a set of materialized security views, besides the space overhead, view updating complexity is another problem that cannot be ignored, since the views have to be consistent with the base data on every query and update execution. Considering the large size of XML documents in use, materializing the security views every time a query is issued is not realistic. An alternative is to maintain the security views as virtual views, and rewrite each query q on a security view to q' against the base data. This is the approach taken by [6]. The enrichment of the security view specification we propose in this paper poses great challenges to the query rewriting process.

The main contributions of this paper can be summarized as follows:

- We propose an infrastructure for access control on XML documents - specifying security constraints and rewriting user queries, avoiding view materialization.
- We introduce an algebraic security view specification language SSX, which enables conditionally hiding and reorganizing XML elements/sub-trees.
- We propose SAS - an annotated schema to represent security constraints specified in SSX and a set of rules that rewrite user queries based on the SAS .
- We conduct extensive experimental evaluation on various benchmark databases and prove that our approach is both effective and efficient.

The rest of the paper is organized as follows: Background information about XML data, XPath and XQuery is introduced in Sec.2, along with the formal definition of the problem. In Sec.3, we introduce our security view specifications language SSX. The two primary components of our method, viz. the annotation process and the rewrite algorithms are discussed in Sec.4 and 5, respectively. We present the experimental results in Sec.6. Related work is discussed in Sec.7, followed by the conclusion and discussion of future work in Sec.8.

2 Preliminaries and Problem Definition

XML data is frequently represented as a rooted node-labeled tree structure, in which the nodes represent the objects (element (tags), element contents and attributes), and the edges represent the containment relationship among the objects. DTDs and XML schema are two popular languages for representing XML schema information. A typical XML structure can be represented as a tree, as shown in Figure 1. There is no substantial difference between a DTD and a XML schema in representing XML schema information, as far as security issues are concerned (XML schema-specific features such as data types are not required for our security view specifications). Hence, for the purpose of clarity, we shall use trees to represent schema information in the first half of this paper, and switch to XML schema when the implementation is discussed.

XPath [4] is a declarative query language on XML documents and is the core of other complex XML query languages, such as XQuery [3]. An XPath expression declares the query requirement by identifying the node of interest via the path from the root of the document to the elements which serve as the root of the sub-trees to be returned. Branching predicates (if any) enforce additional value/structural restriction on the elements along the path. To facilitate the discussion in this paper, we define a Simple Path Expression as follows:

Definition 1 A simple path expression (SPE) is an XPath expression of the form $p \doteq \epsilon \mid l \mid \star \mid p_1/p_2 \mid //p_1$ where p_1 and p_2 are simple path expressions.

In other words, an SPE is an XPath expression without branching predicates.

XQuery is a functional query language. The introduction of the FLWOR statement in XQuery enables variable declaration, variable binding, and result reconstruction, all of which are beyond the capability of XPath queries. We will take advantage of these features of XQuery in our query rewrite process.

Formally, we define the problem of XML security view specification and rewrite as:

Define a security view specification language L . Given an original XML schema S_0 and a sequence of primitives Sp_L in L , derive the view schema S_v that reflects the security constraints expressed by Sp_L . For each query q issued on S_v , evaluate q by rewriting it to q' on S_0 and executing q' on the base data(D_0), guaranteeing that $q'(D_0) \equiv q(Sp_L(D_0))$.

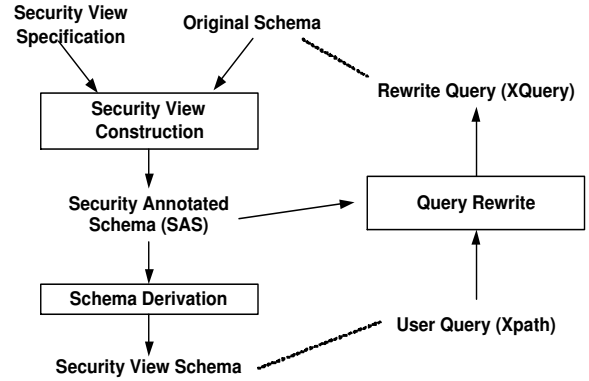


Figure 3: The Infrastructure of the Security View Based Query Answering System

The infrastructure of our security-view based query answering system is as shown in Figure 3. The system can be divided into two components: security view construction (on the left) and security view based query answering (on the right). Taking the original XML schema and security view specification (in the security view specification language SSX) as input, the **Security View Construction** process constructs an Security Annotated Schema(SAS). SAS is an internal representation in our system and it is straightforward to derive the schema of the security view from an SAS. The view schema is visible to the user group to whom the security constraints is to be applied, and it is the only schema that is made available to this user group. Note that it should be transparent to the users whether the view has been materialized or not. In our system the security views are not materialized by default. The **Security View based Query Answering** process rewrites the user queries (in XPath) into equivalent target queries (in XQuery) against the original schema by appropriately using the information in the SAS, and evaluates the target queries on the base data. We will focus on the security view specification in Sec. 3, the SAS generation in Sec. 4 and the rewrite algorithms in Sec. 5.

3 XML Security View Specification

In this section, we introduce our Security Specification Language for XML (SSX) in the form of a set of primitives and show how to specify security constraints using SSX.

3.1 Security View Specification Language

We assume that in the XML context, creation of the security view specification is the responsibility of DBAs, who have basic knowledge about databases and are familiar with the schema of the base data, and are aware of the security concerns with respect to each user group. We also assume that as a first step towards building the framework that a schema is available for the XML document being secured and it is acyclic.

Given security constraints such as those in Examples 1.1, 1.2 and other similar cases we can summarize the basic requirement of a security view specification language as a language that should be able to achieve the following schema transformations:

- R-1 (conditionally) eliminate element/sub-tree;
- R-2 (conditionally) break the association between the children of an element;
- R-3 (conditionally) copy/move elements/sub-trees to a higher level;
- R-4 (conditionally) copy/move a group of elements/sub-trees without breaking the association between them;
- R-5 (conditionally) break the ordering between instance nodes;
- R-6 rename elements/attributes;
- R-7 create new elements/attributes;

Having these in mind, we introduce a Security Specification Language for XML (SSX) in the form of a set of primitives. Each primitive takes an XML schema tree as input, and outputs an XML schema tree. The parameters and functions of the primitives are defined as follows (parameters within square brackets are optional):

create(destSPE, newName) : newName is a string. The **create** primitive creates a new element with tag 'newName', as a child of each element that matches the destSPE in the input schema.

delete(destXPath): The **delete** primitive removes the sub-trees rooted at the elements/attributes that matches the destXPath in the input schema.

copy(sourceXPath, destSPE, [newName], [scope], [preserve]) : newName – a string, scope – an SPE, and preserve – a boolean, are optional. For each element that matches the scope, the **copy** primitive creates an identical copy of the sub-trees rooted at the nodes that match the sourceXPath in the original schema with respect to the scope, and makes them the children of the elements that match the destSPE in the input schema. If a new name is provided, a new element tag is assigned to the root element of the copied sub-trees; otherwise, the original element tag is used.

The default value for scope is '/'. The fifth parameter 'preserve' is a flag that specifies if the copy primitive should preserve (the default) or break the document order between instances being copied.

rename(destSPE, newName): The **rename** primitive assigns a new name (newName) to the elements (attributes) that matches the destSPE in the input schema.

A security view specification is then written in the form of a sequence of these primitives. The primitives are applied sequentially, and they are not necessarily symmetric ($o_1 \circ o_2 \neq o_2 \circ o_1$). Each primitive takes the result of the sub-sequence in front of it as input. The final result is the secured schema for the security view defined by the SSX sequence. The secured schema is internally represented as a SAS (See Sec. 4).

To facilitate the reading/writing of an SSX sequence, we also introduce a set of secondary operations, which are derived from the primitives. An example is provided below.

Move(sourceXPath, destSPE, [newName], [scope], [preserve]): The parameters of a move primitive are the same as that of the copy primitive. The **move** primitive moves sub-trees that match the sourceXPath in the original schema and makes them children of the elements that match the destSPE in the input schema, within the scope, if specified. A new name, if specified, is assigned to the root of the moved sub-trees. The move operation is defined as the concatenation of the copy primitive and the delete primitive: $Move(p, d, n, s, ps) = copy(p, d, n, s, ps) \bullet delete(p)$.

Proposition 3.1 *The primitives (create, delete, copy, rename) can address all the schema transformations that are necessary to enforce XML access control according to [R-1] through [R-7].*

Proof sketch: The proof follows by construction:

- R-1 Blocking sub-trees rooted at a particular element can be achieved by a delete operation on that element.
- R-2 This can be achieved by a delete operation on each child that needs to be dissociated. In the case the dissociated children need to be retained, the delete operations should be preceded by a copy.
- R-3 This security concern involves copying children of an element to a higher level, and can be achieved by a copy operation to the destination node. In case of a move, the secondary primitive move can be used.
- R-4 This is achieved by performing a scoped copy (with the parent being the scope). In case of a move, the move primitive can be used.
- R-5 This security concern requires breaking the order between instance nodes, and can be achieved by using the unordered copy that breaks document order, and hence association between instances.

R-6.7 Renaming/Creating elements can be done by using the rename and the create primitives. ■

Example 3.1 *Let's reconsider the security concerns and security view tree structures in Examples 1.1 and 1.2. The following are the security view specification sequences, written in SSX that specify the security views shown in Figure 2:*

Security View Tree Structure 1:

```
delete(/HR/Dept/Employee/Age)
delete(/HR/Dept/Employee/Salary)
```

Security View Tree Structure 2:

```
copy(/HR/Dept/Employee, /HR/Dept, Emp, /HR/Dept)
delete(/HR/Dept/Employee/Age)
delete(/HR/Dept/Employee/Salary)
delete(/HR/Dept/Emp/Emp#)
delete(/HR/Dept/Emp/Name)
delete(/HR/Dept/Emp/Phone)
```

Security View Tree Structure 3:

```
copy(/HR/Dept/Employee, /HR, Emp, /HR)
delete(/HR/Dept/Employee/Age)
delete(/HR/Dept/Employee/Salary)
delete(/HR/Emp/Emp#)
delete(/HR/Emp/Name)
delete(/HR/Emp/Phone)
```

Security View Tree Structure 4:

```
create (/HR/Dept, 'Sensitive Info');
move(/HR/Dept/Employee/Age,
     /HR/Dept/Sensitive Info, /HR/Dept)
move(/HR/Dept/Employee/Salary,
     /HR/Dept/Sensitive Info, /HR/Dept)
```

Given the general definition of the primitives as stated above, one can come up with sequences that are counterintuitive or ambiguous as shown below in Example 3.2.

Example 3.2 *Using the XML schema tree shown in Figure 1 as the original schema, let's examine the following security view specification.*

```
copy(/HR/Dept/Employee/salary, /HR, 'sal', /HR/Dept, false)
```

This expression copies the salary information of all the employees in a dept (specified by the scope /HR/Dept) and makes them children of each HR element. This operation is counterintuitive and is not an legal security view specification in SSX under the restrictions described below.

We propose the following restrictions to avoid such counterintuitive primitives.

- **Restrict the destPath of create, copy and rename to be an SPE**. This means that the parent of a newly created element, the element/attribute to be renamed, and the destination of the copy have to be unconditionally identified in the input schema. All security specifications that are restricted by this constraint can be expressed alternatively.
- **Restrict the sourcePath of copy to the original schema/data**. We restrict the sourcePath of the copy primitive to be evaluated against the source schema, rather than the input schema to the primitive. This

means, one cannot copy anything that has been modified by primitives prior to the copy operation in the SSX sequence, to a new location. This constraint also specifies that the condition on the sourcePath of the copy operation has to be evaluated against the source data. Since security specifications are written by the DBAs, who are seasoned database professionals and are familiar with the schema and transformation processes, most meaningful security specification can still be specified using SSX with this constraint. Although seemingly a limitation, this restriction is needed to ensure the tractability of the rewrite operations.

- **Restrict the scope in the copy primitive**. We restrict the scope of a copy primitive to be an SPE that evaluates to a common ancestor of the elements that are identified by the sourceXPath and the destSPE. This restriction is necessary as the semantics of a scoped copy is similar to that of a 'grouping' operation. This constraint will only eliminate some counter-intuitive specifications.
- **Restrict the target node in the rename primitive**. We restrict the target node of a rename primitive to be a node that has not been modified. This restriction does not limit the power of the operations as any newly constructed node can be set to the correct name by directly using the newName argument of the create and the copy primitives.

The above restrictions do not limit the expressiveness of SSX, as it restricts only ambiguous security constraints. Meaningful constraints can still be expressed as shown in Example 3.3. The reader should note that the goal of this research is not to mimic all possible transformations of XML trees, but only the potential security constraints.

Example 3.3 *The operation rename(/HR/Dept/Employee[salary>100,000], 'Rank-Emp') is a meaningful specification. It cannot be done with the restriction that the destination path be an SPE. However, it can be accomplished by the sequence*

```
copy(/HR/Dept/Employee[salary>100000], /HR/Dept, 'Rank-Emp',
     /HR/Dept)
delete(/HR/Dept/Employee[salary>100000])
```

Given that security view specifications are defined as a sequence of primitives with each primitive accepting the result of the previous primitive as the input, there is a dependency amongst the primitives in the sequence, based on their position in the sequence as shown below.

Example 3.4 *Let's revisit the SSX sequences in Example 3.1. The two delete operations in the sequence that construct the security view tree structure 1 are independent of each other. Switching the two operations in the sequence results in the same security view. Such is not the case, however, for the sequence that constructs the security view tree structure 2. The two delete operations that remove the Age and Salary nodes from Employee can be placed anywhere in the sequence. However, the three delete operations that*

remove *Emp#*, *Name* and *Phone* have to follow the copy operation. Consider two other SSX sequences:

- *sequence1*:
`delete(/HR/Dept/Employee[age>50]/salary)`
`delete(/HR/Dept/Employee[age[.>55]])`
- *sequence2*:
`delete(/HR/Dept/Employee[age[.>55]])`
`delete(/HR/Dept/Employee[age>50]/salary)`

Sequence 1 hides the salary of all employees over 50 years old and the age information of all employees older than 55. *Sequence 2*, by switching the two delete operations, ends up hiding the age information of all employees that are over 55, but only the salary of the ones between 50 and 55. Hence there is clearly an dependency between the two operations.

Definition 2 An operation op_1 logically depends on an operation op_2 if the XPath expressions specified as parameters of op_1 depends on the elements that match the destination path of op_2 for its evaluation.

With this definition, given an SSX sequence, there is a chronological order among the operations. Given that the dependency is not linear, switching operations could possibly change the semantics of the security view being constructed. Furthermore, SSX sequences that specify the same security view can be substantially different.

4 Security Annotated Schema

To facilitate query answering and rewriting, we propose an internal representation - Security Annotated Schema (SAS) for the schema transformation specified by an SSX sequence. This section introduces the SAS and proposes an algorithm to construct it.

4.1 Schema Annotation

We introduce a set of schema annotations, to reflect the primitives that modify the schema tree structure. All the annotations are associated with the element/attribute node that was modified.

- **Delete Annotation** identifies that the node has been removed, with additional parameters identifying whether the node was removed conditionally. In case of a conditional delete the condition used to delete the node is also stored.
- **NewNode Annotation** identifies that the node has been newly constructed. A **NewNode** annotation with parameter ‘newname’ can be the result of a create or a rename primitive. A copy primitive also results in a **NewNode** annotation, with additional parameters identifying the sourcePath, scope and whether the ordering among sub-trees are to be preserved.
- **Scope Stamp** A node N is stamped ‘Scope’ if any of its descendants has the NewNode annotation, and its scope parameter is a path that matches to N .

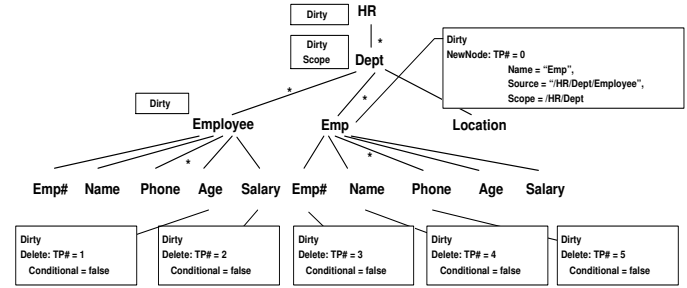


Figure 4: Security Annotated Schema for the Schema Tree Structure 2 in Figure 2

- **Dirty Stamp** A node is stamped ‘Dirty’ if any of its descendants (including itself) is annotated.
- **Chronological Operation Sequence #** This is a sequence # assigned to each operation in an SSX sequence. It reflects the potential dependency of a primitive on preceding primitives in the sequence.

Example 4.1 The SAS that derives the security view tree structure 2 in Figure 2 is shown in Figure 4.

The annotation associated with *Emp* reflects that it is a new element created via a copy primitive. The source is */HR/Dept/Employee* within the scope */HR/Dept*. In addition, the scope stamp associated with the *Dept* node reflects that it has been used as a scope. The Chronological Operation Sequence # (TP#) associated with the *NewNode* annotation identifies that it is the result of operation #1 in the SSX sequence. The delete annotations with the conditional flag set to false indicate that the corresponding nodes have been deleted unconditionally. Note that a dirty stamp simply indicates that either the node or its descendants has been modified.

4.2 Security Annotated Schema Derivation Algorithm

The annotation algorithm takes a schema and a SSX sequence as input (dealing with one operator at a time), and creates an SAS which can be used for rewriting user queries. The annotation algorithm is presented in Figure 5. Associated helper functions are presented in Figure 6.

The algorithm iterates through the sequence of primitives using the output schema from the previous primitive as input. For each primitive, the destination XPath is evaluated on its input schema and a node is created, deleted, renamed or copied depending on the operation. The newly modified node is suitably annotated by making use of the input arguments of the operation along with the current primitive’s sequence #. The current primitive’s sequence # is used to establish the chronological order. The copy operation alone utilizes both the original schema as well as the output schema from the previous operation. The nodes to be copied are obtained from the source schema and the modification is performed on the output schema obtained from the previous operation.

```

CreateAnnotation(sch: Schema DOM,
                op: Sequence of operations)
{
  outputschema = sch;
  currentop = 0;
  for each o in op do {
    switch o {
      case o = delete operation arguments(p1):
        nlist = evalxpath(outputschema, p1);
        for each node t in nlist {
          addAnnotation(t, p1, 'delete', currentop);
          setdirty(t);
        }
      case o = create operation arguments(p1, name) :
        nlist = evalxpath(outputschema, p1);
        for each node t in nlist {
          if (t is leaf)
            convert t to complextyp;
            create new node n1(name) as child of t;
            addAnnotation(n2, p1, 'newnode', 'create',
              currentop);
            setdirty(n1);
          }
        }
      case o = rename operation arguments(p1, newname) :
        nlist = evalxpath(outputschema, p1);
        for each node t in nlist {
          change name of t to newname;
          addAnnotation(n2, p1, 'newnode', 'rename',
            currentop);
          setdirty(t);
        }
      case o = copy operation arguments(p1, p2,
        newname, scope, preserve):
        sourcen1 = evalxpath(sch, p1);
        destn1 = evalxpath(outputschema, p2);
        for each node s in sourcen1 {
          n2 = copydom(s)
          if (newname != null)
            root tag of n2 = newname;
          for each node t in destn1 {
            if (t is leaf)
              convert t to complextyp;
              add n2 as child of t;
              addAnnotation(n2, p1, 'newnode', 'copy',
                currentop, scope, preserve);
              setdirty(n2);
              if scope != '/'
                setscope(scope);
            }
          } end for destn1
        }
      } // end switch
    currentop++;
  } // end for
  return outputschema;
} // end CreateAnnotation

```

Figure 5: SAS Construction Algorithm

5 Security View Based Query Rewrite

Once the SAS has been constructed, it is trivial to derive the security view schema and expose it to the end users. The next and the most intriguing problem is to answer user queries, guaranteeing all security restrictions, without materializing the views. This section introduces a set of rules to rewrite user queries and describes an algorithm to implement the rules.

We chose to rewrite the user queries (in XPath) to queries (in XQuery) against the source schema. This is non-trivial, due to the following facts about the way security restrictions are specified and the manner in which the SAS is constructed:

- The impact of the annotations added to a specific element is on the sub-tree rooted at that element.
- There is a partial ordering among the annotations, *i.e.*, some annotations may depend on others, and during

```

evalxpath(outputschema, p1) evaluates the XPath expression
p1 on the outputschema and returns the list of all the
nodes that match p1;
setdirty(node) recursively attaches dirty stamps
at the node and all its ancestors;
setscope(node) Set scope flag for the node;
addAnnotation(...) adds annotation attributes to a
node. In case of a newnode annotation a flag is set,
indicating the actual operation that was performed;
copydom(node) makes a duplicate copy of the entire
DOM tree of the schema rooted at node and returns
the root node of the duplicate version.

```

Figure 6: Helper Functions for the SAS Construction Algorithm

query rewriting, this order cannot be violated, even when the annotations are on different nodes.

- Even though the annotations are on the schema, conditions (if any) have to be evaluated on the data.
- The uncertainty introduced by the wild card and '//'.

We assume that the user queries are written in XPath. The rewritten query cannot be specified in XPath as well, since XPath is not expressive enough to handle the grouping generated by the 'scope' (specified in a copy primitive), and XPath does not have the ability to create new structures (specified in the copy, rename and create primitives). Hence we chose XQuery as the query language of the resulting rewritten query. The `unordered` feature in XQuery also allows us to break the document order (and hence prevent the possible inference of relationships among siblings and sub-trees).

The query rewrite process is rule-based. Given an XML database D containing instances conforming to schema S_0 , a security view V , with SAS S_v , and an XPath expression $p = t_1[c_1]/t_2[c_2].../t_n[c_n]$ against V , where t_i s are tags (element tags, attribute name or wildcard) and c_i s are XPath expressions serving as branching predicates, we develop a set of rewrite rules to transform p to q where $q = Rw_{S_v}(p)$, such that $p(V) \equiv q(D)$. To facilitate the discussion of the rewrite rules and the algorithm, we define a recursive procedure as follows:

$q = Rw_{S_v}(p, vb, vb')$:: The function Rw translates an XPath query p to an XQuery expression q , referencing the annotations in a SAS S_v and under a specific environment, represented by a set of variable-bindings vb . vb' represents the new variable-bindings in q . A conceptual presentation of the rules used by $Rewrite$ is provided below.

RULE 1 Given an SAS S_v , and an XPath expression $p = p_1/t_1[c_1]/p_2$ under environment vb , where p_1 and p_2 are sub-XPath expressions, t_1 is an element tag and c_1 is a branching condition, taking the form of $p_3[op p_4]$ (please note that $op p_4$ is optional), the XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as follows:

```

for $i in RwSv(p1, vb, vb1)
for $j in $i/t1
where RwSv(p3, vb1 ∪ {i, j}, vb3) op
RwSv(p4, vb1 ∪ {i, j}, vb4)
return RwSv(p2, vb1 ∪ {i, j}, vb2)

```

The basic idea behind this rule is that every branching predicate in an XPath expression can be treated as an XPath expression, rewritten into an XQuery expression and be a part of the WHERE clause in the final rewritten XQuery expression. The comparison operators and any literals found in conditions can be directly used in the XQuery expression without any translation. Therefore, we focus our discussion on rewriting an SPE from now on.

The rewrite algorithm works under the assumption that given nodes ‘A’ and a descendant ‘B’, all possible paths from ‘A’ to ‘B’ are precomputed and stored in the SAS.¹

RULE 2 Given an SAS S_v , and an SPE $p = p_1/t/p_2$ under environment vb , where p_1 and p_2 are SPEs, the XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as follows:

```

for $i in RwSv(p1, vb, vb0)
return
  {RwSv(t1/p2, vb0 ∪ {i}, vb1)},
  ⋮
  {RwSv(tn/p2, vb0 ∪ {i}, vbn)}

```

where $\{t_1, \dots, t_n\}$ are the set of paths that lead to t from p_1 .

A ‘Dirty’ stamp associated with a node in the SAS identifies that either the node itself or its descendant(s) has been modified by an SSX sequence. Before presenting the rewrite rules that respond to ‘Dirty’ stamps, we first define the following important notions:

Definition 3 Given an SAS S_v and an SPE $p = t_1/t_2\dots/t_n$, if there exists t_k , such that t_{k-1} has a ‘dirty’ stamp and none of t_k, t_{k+1}, \dots, t_n has a ‘dirty’ stamp, we call t_k the highest clean element (HCE) of p on the S_v , and $p_1 = t_1/t_2\dots/t_{k-1}$ the prefix path w.r.t. S_v and $p_2 = t_k/t_{k+1}\dots/t_n$ the suffix path w.r.t. S_v .

The notion of HCE enables us to leave the suffix path as is in the rewrite.

RULE 3 Given an SAS S_v , and an SPE p under environment vb , if there exists an HCE on S_v such that $p = p_1/p_2$ (p_1 is the prefix path w.r.t. S_v , and p_2 is the suffix path w.r.t. S_v), we define the equivalent XQuery expression $q = Rw_{S_v}(p, vb, vb')$ of p as follows:

```

for $i in RwSv(p1, vb, vb0)
return $i/p2

```

On the contrary, when the tail of the SPE has a ‘Dirty’ stamp, rather than stopping at the node and returning the whole sub-tree rooted at the node, as an XPath query usually does, the children of the tail node have to be constructed individually as the ‘Dirty’ stamp indicates that at least one descendant node has been modified.

¹This has been efficiently carried out by recursively walking the tree at the end of the view generation process.

RULE 4 Given an SAS S_v and an SPE $p = t_1/t_2\dots/t_n$ under environment vb , if t_n is dirty and has children cld_1, \dots, cld_m in S_v , the equivalent XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as follows:

```

for $i in RwSv(t1/t2.../tn-1, vb, vb0)
return
  <tn>
    RwSv(cld1, vb0 ∪ {i}, vb1)
    ⋮
    RwSv(cldm, vb0 ∪ {i}, vbm)
  </tn>

```

Each primitive in SSX results in its own annotation in the SAS. These need to be treated differently in the rewrite procedure. The rest of the rules explain how the various primitives are handled.

Unconditional delete removes not only an element, but the whole sub-tree rooted at the element. Therefore, if any node on an SPE has an unconditional delete annotation, the SPE evaluates to the empty sequence (). Conditional delete removes element/attributes based not only on the schema, but also on the data instance. Conditional delete on a node being queried is rewritten as a condition in the resultant query.

RULE 5 Given an SAS S_v and an SPE $p = t_1/t_2\dots/t_n$, if t_i ($0 \leq i \leq n$ is an unconditional delete annotation in S_v , the equivalent XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as:

```

return ()

```

RULE 6 Given an SAS S_v , and an SPE $p = t_1/t_2\dots/t_n$ under environment vb , if node t_k ($k < n$) is annotated by conditional delete operation, with condition $cond$, the equivalent XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as:

```

for $i in RwSv(t1/t2.../tk, vb, vb0)
where count(RwSv(cond, vb, vb1)) = 0
return RwSv(tk+1/.../tn, vb0 ∪ {i}, vb2)

```

Here, S'_v is the prefix of S_v , up to the operation before the conditional delete in question.

The copy operation constructs new sub-trees in the schema tree structure. It is annotated on the new sub-tree root, with all the required information for the rewrite: source path, scope, preserve, etc. When a node in the copied structure is queried, data is retrieved from the source, along the source path. The retrieved data is compliant to any annotations on the descendant sub tree.

RULE 7 Given an SAS S_v , and an SPE $p = t_1/t_2\dots/t_n$ under environment vb , if node t_y ($y < n$) is annotated as a ‘new node’ generated by the copy operation $copy(t_1/t_2/\dots/t_k/\dots/t_x, t_1/t_2/t_k/\dots/t_{y-1}, t'_x, /t_1/t_2/\dots/t_k, ps)$, where $(t_x = t_y \wedge t'_x = \phi) \vee t'_x = t_y$, the equivalent XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as:

```

for $i in  $Rw_{S_v}(t_1/t_2.../t_k, vb, vb_0)$ 
for $j in  $Rw_{S_v}(t_{k+1}.../t_{y-1}, vb_0 \cup \{i\}, vb_1)$ 
for $k in  $\$i/t_{k+1}/t_{k+2}/.../t_x$ 
return  $Rw_{S_v}(t_{y+1}/.../t_n, vb_0 \cup \{i, k\}, vb_2)$ 

```

Both the rename and the create operations generate new tags that do not exist in the original schema, at the specified location. If such a node appears in the middle of an XPath expression, the rewrite process simply matches it and moves on to the sub-trees rooted at the node in question.

RULE 8 Given an SAS S_v , and an SPE $p = t_1/t_2.../t_n$ under environment vb , if node t_y ($y < n$) is annotated as a “new node”, generated by the rename operation $rename(/t_1/t_2/...t_{y-1}/t_x, t_y)$, the equivalent XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as:

```

for $i in  $Rw_{S_v}(t_1/t_2.../t_{y-1}/t_x, vb, vb_0)$ 
return  $Rw_{S_v}(t_{y+1}/.../t_n, vb_0 \cup \{i\}, vb_1)$ 

```

RULE 9 Given an SAS S_v , and an SPE $p = t_1/t_2.../t_n$ under environment vb , if node t_y ($y < n$) is annotated as a “new node”, generated by the create operation $create(/t_1/t_2/...t_{y-1}, t_y)$ or, the equivalent XQuery expression $q = Rw_{S_v}(p, vb, vb')$ is defined as:

```

for $i in  $Rw_{S_v}(t_1/t_2.../t_{y-1}, vb, vb_0)$ 
return  $Rw_{S_v}(t_{y+1}/.../t_n, vb_0 \cup \{i\}, vb_1)$ 

```

A dirty node without annotation does not need to be specially processed, and the processing can continue along the path if the node is not the tail node of the input XPath query.

RULE 10 Given an SAS S_v , and an SPE $p = t_1/t_2/.../t_y/..t_n$ under environment vb , if node t_y ($y < n$) is dirty without any annotations then we can directly proceed to the next element in the path.

```

for $i in  $Rw_{S_v}(t_1/t_2.../t_{y-1}, vb, vb_0)/t_y$ 
return  $Rw_{S_v}(t_{y+1}/.../t_n, vb_0 \cup \{i\}, vb_1)$ 

```

Example 5.1 Following the rewrite rules described above, let’s take a look at a few queries on the security view whose SAS is as shown in Figure 4.

```

XPath Query  $p_1$ : /HR/Dept/Employee/Name
Rewrite Query  $q_1$ :
  for $i in doc("hr.xml")/HR/Dept/Employee/Name
  return $i

```

```

XPath Query  $p_2$ : /HR/Dept/Employee/Salary
Rewrite Query  $q_2$ :
  return ()

```

```

XPath Query  $p_3$ : /HR/Dept/Employee
Rewrite Query  $q_3$ :
  for $i in doc("hr.xml")/HR/Dept/Employee
  return
    <Employee>
      {$i/Emp#}
      {$i/Name}
      {$i/Phone}

```

```
</Employee>
```

```

XPath Query  $p_4$ : /HR/Dept/Emp[Salary > 10000]
Rewrite Query  $q_4$ :
  for $i in doc("hr.xml")/HR/Dept
  for $j in $i/Employee
  Where ($j/Salary > 10000)
  return
    <Emp>
      {$j/Age}
      {$j/Salary}
    </Emp>

```

5.1 Rewrite Algorithm

The rewrite algorithm skeleton is presented in Figure 7. The rules discussed above as well as the presentation of the algorithm only deal with elements. However, the rules as well as the algorithm can be easily adapted to handle attributes. For the presentation of the algorithm, we are assuming a schema which has been annotated as described in Section 4. The algorithm accepts as input an XPath query (q) that needs to be rewritten. It iterates through the input XPath query and walks the schema tree based on the tokens found in the input XPath Query. Annotations, if found during the tree walk, are appropriately handled to generate the final XQuery expression. In case of a condition in the input XPath, the condition expression is treated as an XPath expression and the procedure is recursively called to generate a XQuery expression for the condition. Operators, if encountered in a condition, are substituted with equivalent XQuery operators while literals are carried over to the XQuery expression without any changes.

Following the rules, the algorithm needs to be provided with an environment that includes information such as *current node* (*currnode*) of the schema that is being processed, the *current variable count* (*vcount*) for keeping track of variable bindings, the variable binding to be used for the current element (*tailexpr*), keep track of scope paths and variables associated with them and a *flag* (*indelete*) that determines whether processing is within a delete annotation (since deletes, especially conditional deletes have to be handled differently than the other operations. See Rule 6). We refer to the environment parameters as *env.paramname*, e.g., *env.currnode*, etc in the algorithm. To process a user XPath query, rewrite needs to be bootstrapped as follows:

```

set env{currnode = node(xs:schema), vcount=1,
      tailexpr='', indelete = false}) Rewrite(q)

```

Note that all the special cases have not been included in the algorithm because of space limitations, however, the algorithm is true to the rules described above and exhaustive experimentation indicates the algorithm to be a complete representation of the rewrite rules. A proof of concept prototype of the view generation and the query rewrite process has been developed using Java, JAXP and Galax [8] and is available for download at [15].

```

Rewrite(q:Parsed XPath) {
  nexttoken = Obtain next token from input XPath
  if (nexttoken = '//')
    // Apply Rule 2
  else if (nexttoken = '/')
    // Continue loop with next token
  else if (currtoken is XPath condition c)
    // APPLY RULE 1
  else if (nexttoken is an element) {
    if (env.curnode has no children on the desired path)
      output ``return ()``
    if (env.curnode is not dirty)
      // APPLY RULE 3 for HCE
    else if (annotation A in curnode)
      switch A {
        case A = 'udelete':
          // APPLY RULE 5
        case A = 'create':
          // APPLY RULE 9
        case A = 'copy':
          // APPLY RULE 7
        case A = 'rename':
          // APPLY RULE 8
        case A = 'Cond del':
          // APPLY RULE 6
      }
    else if (Dirty node with no annotation)
      // APPLY RULE 10
    if (current element is last Token)
      // APPLY RULE 4
  }
} // end Rewrite

```

Figure 7: Rewrite Algorithm

5.2 Soundness and Completeness Properties

The rewrite algorithm is sound (all security operations are properly handled) and complete (no more than what is specified is blocked). Full proofs of these properties are beyond the scope of this paper, but short proof sketches are presented below.

Theorem 5.1 Soundness. *Given a document D conforming to a schema \mathcal{S} , and a sequence of security view operations \mathcal{O} that blocks access to a secure path p in D , then there is no user XPath query q such that $\exists x \in D.p \wedge x \in R_{w_{\mathcal{S}_v}}(q)(D)$.*

Proof sketch. This theorem can be proved by contradiction. Assume that there is a query q such that for some node $x \in D.p$, running $R_{w_{\mathcal{S}_v}}(q)$ on D exposes x . The proof follows a case-by-case analysis of each of the rules, ensuring that the data returned by the rules do not include any suppressed node. Note that Rules 1, 2, 4, 6, 7, 8, 9 and 10 all call $R_{w_{\mathcal{S}_v}}$ recursively, and only Rules 3 and 5 return actual nodes of the tree. Rule 5 definitively blocks any suppressed element by returning nothing, and Rule 3 clearly only returns clean elements. Analysis of the for statements of the other rules demonstrate that no information is passed in the recursive calls that can expose any blocked node. So x could not have been exposed, and is a contradiction. Hence the proof. ■

Theorem 5.2 Completeness. *Given a document D conforming to schema \mathcal{S} , and a sequence of security view operations \mathcal{O} , if there exists a node x in D which is not blocked by any operation $op \in \mathcal{O}$, then there exists a user XPath query q that can retrieve x .*

Proof sketch. The proof of this statement is more involved since it requires an analysis of how the annotations are done. Regardless, the intuition behind this proof comes from the fact that only non-dirty nodes are exposed verbatim from the source in the generated XQuery expression. Any dirty node is recreated in our query rewrite algorithm, blocking any suppressed nodes and exposing only non-suppressed nodes. Hence, for sub-trees that are not stamped dirty, no blocking operations are performed, and hence can always be reached by user XPath queries. As shown in Figure 4, nodes that users have access to are not marked dirty (e.g., nodes referred to by paths like `/HR/Dept/Emp/Age`), and through a case by case analysis it can be shown that every such “clean” node will have a corresponding XPath. Hence the proof. ■

6 Experimental Evaluation

To demonstrate the effectiveness and efficiency of the techniques proposed in this paper, we conducted evaluations of our system using datasets generated by publicly available XML Benchmarks. Our experimental results clearly revealed that our approach was superior in terms of effectiveness and performance when compared to materialized views and other previously available methods for securing XML documents. Our approach on average had a performance which is quite similar to that of materialized views, but in case of uncertainty (wild card and `//`) and in the presence of deeply nested XPaths the improvement is substantial. Our approach also proved to be superior to the recent work by Fan et al. [6] providing better functionality and better or comparable performance on common functionality. Our experiments were conducted on a 2.8 GHz Intel Pentium IV machine with 512 MB of memory running Linux. The times reported in the graphs were obtained as an average over 5 runs.

6.1 Experimental Setup

DataSet The data sets that are reported in this section model a catalog of books (a typical real world scenario) generated using the Xbench [16] Benchmark. Datasets produced by the Xbench test suite were modified to generate 3 different document sizes - 10, 50 and 75 MB.

Security View Generation Our efforts were targeted at testing the rewrite and the evaluation performance of the various primitives, both together as well as independently. In accordance, we created 7 different security models, with the first four testing a specific primitive thoroughly while the last three implemented a good and viable mixture of the four primitives put together. For each of the above models, SAS(referred to as the SSX virtual views(vv) in the graphs) and materialized views (mv) were constructed. The materialized views were constructed using a specially written XSL script. The time taken to generate the SAS and the materialized views for the 10, 50 and 75 MB files are shown in Figure 8. As can be seen our security view generation process is independent of the size of the data and

therefore the generation is done in constant time while the time for view materialization grows linearly with the size of the data. The SAS view is only dependent on the sequence of security primitives. Further materialized views suffer from data maintenance as any update to the data has to be cascaded to all the materialized views.

For the comparisons against Fan et al. [6] we just considered the security models that implemented the delete primitive (the only primitive supported in [6]). Virtual views used by [6] were not generated as a part of the experiment.

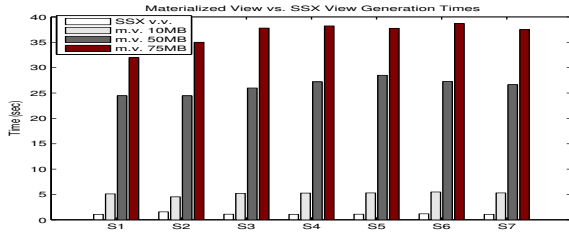


Figure 8: Materialized vs SSX View Generation Times

Query Profiles We chose 4 different types of queries to run on the 7 security models. Type 1 attempted to retrieve the nodes that were modified by the security primitives while Type 2 varied from deep paths (for example length > 7) to small paths (length > 3), Type 3 attempted to retrieve paths that were not modified and Type 4 attempted to retrieve the entire document. We also tested paths with “//” to test the performance in the presence of uncertainty. Queries for the Fan et al. technique [6] were generated by hand based on the algorithm described in [6]. Figure 9 presents the results from 7 sample queries against the materialized views while Figure 10 presents the results from 5 sample queries (distinct from the previous test set) against [6]. The generated queries were tested using Galax [8]. Complete details of the queries and the timing results are not presented because of space limitations but can be obtained from [15].

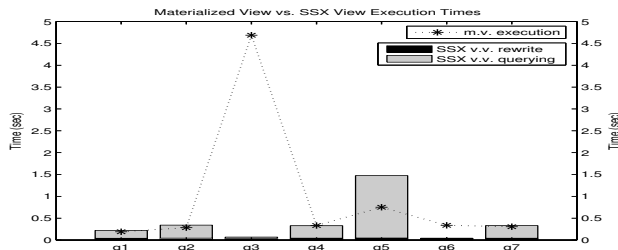


Figure 9: Materialized vs SSX View Querying Times

6.2 Experimental Results and Analysis

Based on the comparison against materialized views and the method proposed by Fan et al. [6] we can arrive at the following conclusions:

SSX Rewrite Versus Materialized View

- For queries with uncertainty (the presence of a // or a wild card) the rewrite approach is much faster than the

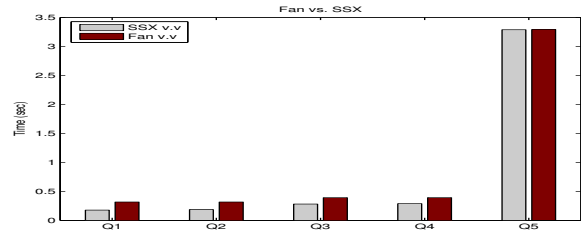


Figure 10: SSX View vs Fan et al. View Querying Times

materialized views (q3 in Figure 9). This is because the reachability of each node has been computed before hand.

- For queries with deep paths (Types 1,2) (q1,q6 in Figure 9) the rewrite mechanism exhibits better or comparable performance with that of the materialized views. For queries with small paths the performance is comparable (Types 1,2) (q2, q4 in Figure 9).
- For queries returning the entire document or a node high in the XML tree (Type 4), the materialized view tends to perform better. This is because the return data has to be constructed (if nodes have been modified) in case of the rewrite method (q5 in Figure 9).
- For paths which have not been modified (Type 3), both methods exhibit comparable performances (q7 in Figure 9).

Overall in most cases the extra overhead due to the query rewrite does not affect the performance and in some cases reduces the average querying time. Further view update complexities and data storage issues as illustrated in Figure 8 only get worse with multiple security levels on the same document. This makes our approach more attractive when compared to materialization.

SSX Rewrite Versus Fan et al. Rewrite

From Figure 10 we can see that our approach provides enhanced functionality with better or comparable performance to the approach introduced in [6].

7 Literature Review

Several models of access control have been proposed in literature for specifying and enforcing access control in databases. Access control for relational databases using the notion of views has been presented earlier by [10, 2, 13]. Halevy [7] provided a comprehensive discussion on answering queries over views by rewriting them over the base data. The problem of access control for XML has received comparatively little attention. This section introduces a few papers that study mechanisms to secure XML documents.

XACL [14] and XACML [12] enforced a generic method of securing XML content and are not just limited to data hiding from queries. The policies are enforced via mechanisms that upon an action request either grant or deny access. Such mechanisms are too restrictive and in reality the query must return portions of the result that the

user is authorized to access and not just deny the entire request. An alternative approach was taken by Murata et al. [11] wherein the queries were executed and each element of the result was checked for security and returned only if the user had the requisite authorization to view the element. They also propose the use of static algorithms to check and see if a query could potentially return only accessible elements (safe queries). But the time-complexity of running such algorithms statically is very high and checking each returned element for validity can be exponential.

Issues of access-control inheritance, granularity of access, access overriding and conflict resolution have been studied in detail [1, 5]. Damiani et al.[5] was one of the first works that utilized XPath as a means of specifying security constraints. The paper proposed an algorithm based on tree labeling and annotation to compute a security user view of the data. But the work proposed that the view be materialized - a potentially complex and computationally expensive task. Cryptographic techniques for securing XML content have been studied by Miklau et al. [9]. They introduced the notion of keys to ensure that published data is visible to everybody but only understood by authorized users(achieved by key enciphering).

Fan et al. [6] is the most complete work on XML access control to date. The authors introduced a notion of specifying security constraints by annotating a DTD which is then automatically converted to produce the actual security model with separate security views for each user group. The policies were enforced by rewriting XPath queries and does not require materialization of individual views. However the paper limited the definition of views to either hiding a node directly or based on a XPath condition. It does not consider the notion of structure and hierarchy of XML nodes and does not provide means to hide structural relationships between nodes and also does not talk about the possibility of forming new structural relationships.

8 Conclusion and Future Work

With the growing popularity of XML and XML database systems, the ability to hide data from a group of users is as important as making the data available to users in an efficient and friendly manner. Comparing XML to the relational world, the challenge in XML access control lies in the semi-structured nature of XML documents. In an XML context, not only can values of elements/attributes be sensitive, but also the structural relationship among elements/attributes. We propose an XML security view specification language SSX for DBAs to specify the security constraints. Given an SSX sequence and an XML schema, our system produces a Security Annotated Schema(SAS) from which a view schema can be derived easily. The proposed rewrite algorithm rewrites user XPath queries (against the view schema) to an XQuery expression against the original schema. This technique guarantees the enforcement of security constraints, without the cost of materializing and maintaining the security views. Our experiments demonstrate the security view annotation algorithm

and rewrite algorithm to be effective and efficient.

The techniques proposed in this paper assume that every XML database has a schema and that the schema is acyclic. We are looking forward to extending our work to handle recursions in XML schema and to tackle the more generic scenario where partial or no schema is available. We also propose to study several security models to determine situations where view materialization will be helpful and factor in that analysis to come up with a system which can automatically decide whether a given security constraint has to be modeled as a virtual view or as a materialized view. We plan to study the proposed primitives from a formal perspective to determine useful properties and also perform algorithmic analysis to calculate bounds for the rewrite algorithm.

9 Acknowledgments

We thank the members of the Database Lab at Indiana University for their comments in improving the content and the style of this paper.

References

- [1] E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Trans. Inf. Syst. Secur.*, 5(3), 2002.
- [2] E. Bertino, S. Jajodia, and P. Samarati. A flexible authorization mechanism for relational data management systems. *ACM Trans. Inf. Syst.*, 17(2), 1999.
- [3] D. Chamberlin, J. Clark, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. <http://www.w3.org/TR/xquery>, June 2001. W3C Working Draft.
- [4] J. Clark and S. DeRose. XPath version 1.0. <http://www.w3.org/TR/xpath>, 1999. W3C Working Draft.
- [5] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Extending Database Technology*, 2000.
- [6] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *ACM SIGMOD*, 2004.
- [7] A. Y. Halevy. Answering queries using views: A survey. *The VLDB Journal*, 10(4), 2001.
- [8] J.Simeon and M.Fernandez. The XQuery implementation at www.galaxquery.org.
- [9] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *VLDB*, September 2003.
- [10] A. Motro. An access authorization model for relational databases based on algebraic manipulation of view definitions. In *ICDE*. IEEE Computer Society, 1989.
- [11] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. In *ACM conference on Computer and communications security*, 2003.
- [12] Oasis. Project at <http://www.oasis-open.org/committees/xcaml>.
- [13] A. Rosenthal, E. Sciore, and V. Doshi. Security administration for federations, warehouses, and other derived data. In *IFIP WG 11.3*, 2000.
- [14] S.Hada and M.Kudo. XML access control language: Provisional authorization for XML documents.
- [15] S.Mohan, A.Sengupta, Y.Wu, and J.Klinginsmith. XML access control, at <http://www.cs.indiana.edu/~jklingin/ac/index.html>.
- [16] B. B. Yao, M. T. Ozsu, and J. Keenleyside. Xbench - a family of benchmarks for xml dbms. In *VLDB 2002 Workshop EEXTT*. Springer-Verlag, 2003.