A Framework for Access Control for XML

Sriram Mohan
and
Arijit Sengupta
and
Yuqing Wu

XML is gaining predominance as the standard for data representation and exchange. Access control for XML data is nontrivial as witnessed from the number of access control models presented in literature. Existing models provide the ability to extend access control to data as well as structure and enforce the specified access control via view materialization. However, view materialization based approaches suffer from update issues and maintaining such views may not be realistic. In this context, We introduce ACXESS(Access Control for XML with Enhanced Security Specifications), a framework for formalizing, presenting and enforcing security constraints for XML documents. Through ACXESS, we present SSX - an algebraic view specification language that provides an effective way to capture all the tree transformations achieved by existing access control models using a set of atomic primitives. We choose not to materialize the security views and introduce a notion of virtual security views that enforce the access constraints via query rewrites. A Security Annotated Schema(SAS) is proposed as the internal representation for virtual views expressed using SSX and the virtual security view exposed to the user can be automatically constructed from the SAS. Finally, we propose a rule based rewrite algorithm (SQR) that rewrites user XPath queries on the security view into equivalent XQuery expressions that can be evaluated against the original data, with the guarantee that the users only see information in the security view. Experimental evaluation and theoretical proofs demonstrate the capability of SSX in representing the access constraints expressible in existing access control models and also demonstrate the use of SAS and the SQR rewrite algorithm in enforcing the access constraints without view materialization.

Categories and Subject Descriptors: H.2.7 [**Information Systems**]: Database Management; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems

General Terms: Security, Design, Algorithms

Additional Key Words and Phrases: Access Control, Materialized Views, Query translation

1. INTRODUCTION

XML has become one of the most extensively used data representation and data exchange formats, since its introduction in the late 90's. 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 query XML data either as a part of a relational database or using native XML stores. However,

...

hiding sensitive data is as important as making the data efficiently available, as has been emphasized and studied for decades in relational databases.

1.1 Motivating Example

EXAMPLE 1. *Let's consider the problem of developing and conducting tests through a Course Management System such as Oncourse*[1]*. Online tests and quizzes are stored in XML in Oncourse using the IMS-QTI schema [IMS Global Learning Consortium]. A highly simplified version of this structure is shown in Figure 1(a).*

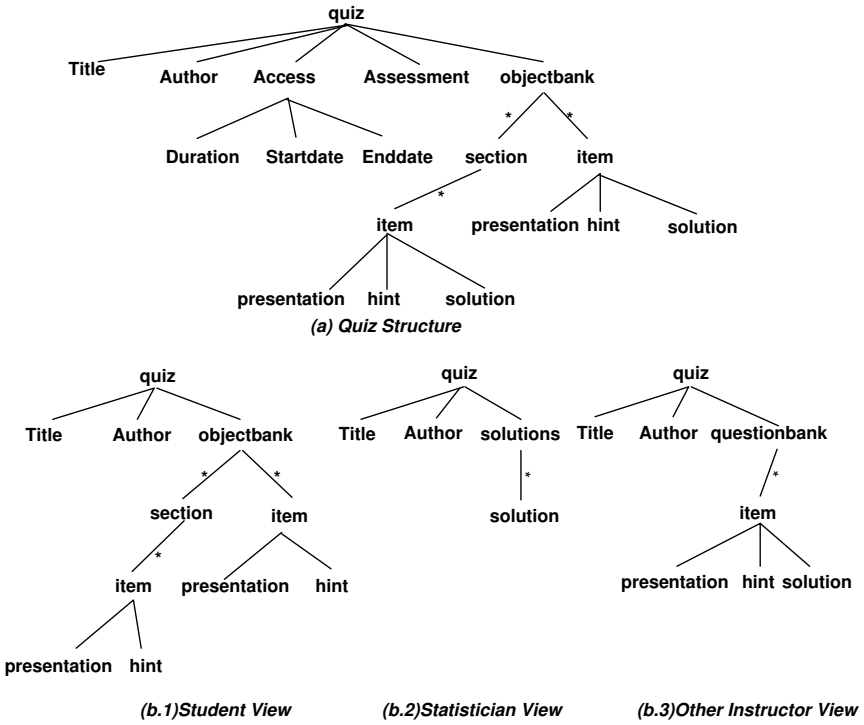


Fig. 1. Simplified tree structure for an online quiz and its security views

Several access constraints can be envisioned in the online quiz application. An author (instructor) should have access to all elements of quizzes he/she writes (full access). A student should only have access to current quizzes in courses that he/she is registered for, but not have access to the solutions (conditional and unconditional removal). Moreover, if quizzes are shared among instructors of different universities, instructors other than the authors should have access to the questions as well as to the solutions, but potentially without the course-specific structuring (conditional and unconditional restructuring). Finally, for summarizing purposes, a statistician

[1]Oncourse is a part of the Sakai project involving University of Michigan, Indiana, MIT, Stanford,OKI and the uPortal Consortium.

may have access to all the solutions. But even for someone who may have permission as both a student and as a statistician, he/she should not be able to figure out the solution to each individual question(removal of association).

In the currently Oncourse implementation, the definition of the access control layers (from a set of 10 predefined access control profiles) are specified in the "view" attribute of the elements in the XML document; hence a specific element can only have one unique access level. This method, even though straightforward, suffers from several problems. It is *literal*, the access constraints are specified directly in the document. It is *static* since there is no way to change the access to specific nodes without actually changing the data. It is also *rigid*, since different components of an element cannot have different access levels. Our work provides a solution to all of these drawbacks.

Materialized view based access control mechanism is another choice, If materialized views are used to implement the above role-based security policy, the views generated would look like those shown in Figure 1 (b).

All the security concerns listed in Example 1 are not uncommon, yet none of the existing techniques support them fully, without actually generating (or materializing) the "views".

1.2 Challenges in XML Access Control

At the very minimum, an access control engine should have two principle capabilities: constraint definition and constraint enforcement. At the constraint definition level, it should support a logical/declarative language with the ability to define the nature of the constraints that needs to be enforced. At the constraint enforcement level, the access control engine must support a mechanism to ensure that the user does not have access to the data that has been blocked. Compare to the access control methods studies in relational databases and object-oriented databases, new challenges needs to be conquered in the context of XML.

(1) *Data Versus Structure*
Given the semi-structured nature of XML data, valuable information is contained not just in the data values but also in the tree structure. Any access control language for XML should provide the DBA with the ability to control access to both data as well as to structure. They should consider the notion of structure and hierarchy of XML nodes and provide means to hide structural relationships between nodes and the possibility of forming new structural relationships.

(2) *Static Versus Dynamic Enforcement*
Access constraints may be specified on XML schema or XML data and the enforcement mechanism varies from static analysis, to data-annotated query rewrite, to materialized views, to virtual view. The static analysis approach is quite useful as long as the database need not be checked to determine access control. In other words, it's difficult to enforce value based access control and conditional access control. Furthermore even where static analysis can be performed, the time complexity of running them is high and checking each returned element for validity can be exponential.

The data-annotation approach is suitable for scenarios where the security levels are relatively stable. The materialized view approach works the best when the data is stable. While the virtual view approach trades off the time spent on maintaining the data annotation or views with the time spent on query rewrite.

1.3 Related Works

<== [[[we may rewrite this part with contribution from Jit]]] Several models of authorization have been designed for database systems supporting the hierarchical, network, relational and even object-oriented models of data. However these models are not adequate in meeting the demands of semi-structured data repositories. This section provides a brief survey of the varying notions of security proposed for relational and object-oriented databases and existing measures of access control for XML databases that have been reported in literature.

Access Control Strategies for Relational Databases The mechanism for specifying security policy for relational databases has generally involved a grant/revoke strategy. Subsets of a user's data, derived data, and other transformations of data are shared by defining a view and sharing that view. Privileges are granted to users by the object owners or the administrator using variations of the "grant" and "revoke" statements supported in SQL. Privileges on an object, once granted, may be withdrawn. Griffiths et.al [Griffiths and Wade 1976] defined the semantics of revocation within a shared database and presented a recursive algorithm which effects these semantics. Upon revocation of a privilege from a user, the algorithm revokes the user's grants of that privilege which were made before the oldest remaining receipt of the privilege. Bertino et. al. [Bertino et al. 1993] extended the revoke operation by introducing a non-cascading revoke.

Access Control Strategies for Object-Oriented Databases For object-oriented and object-relational database systems, enrichments of the existing access control authorization mechanisms to address the richer data models have been studied in [Bertino 1992; Rabitti et al. 1991; Rabitti et al. 1988; Thuraisingham 1989]. Rabitti et.al. [Rabitti et al. 1991] extended the grant/revoke strategy to work with the richer semantics of object-oriented databases. They provided a formal basis for an authorization mechanism for ORION, a prototype object-oriented database system. They introduced the notion of weak - authorizations to assist DBAs in defining constraints that can be overridden by other specifications. Jajodia and Kogan [Jajodia and Kogan 1990] proposed a security model that takes advantage of the encapsulation paradigm which is native to object-oriented databases. The model controls information flow by filtering the messages transmitted between objects. Every message exchanged between objects is intercepted by the message filter. The message filter decides how to handle the message based on the security levels of the sender and the receiver and the information encountered in a chain of method executions. Bertino [Bertino 1992] and Faatz et.al [Faatz and Spooner 1990] improved this idea by introducing an authorization model that exploits methods as a mechanism for access control-thereby allowing for the definition of arbitrarily complex access rules. The access control model applies object interfaces(object views) in order to restrict user interaction with the object.

Access Control Strategies for XML The problem of access control for XML has received comparatively little attention compared to the plethora of literature

available on access control for relational and object-oriented databases. XACL [Hada and Kudo] and XACML [OASIS] enforced a generic method of securing XML content by extending the grant/revoke mechanism. The policies are enforced via mechanisms that upon an action request either grant or deny access. Such mechanisms are too restrictive and in reality a 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. [Murata et al. 2003] 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. Issues of access control inheritance, granularity of access, access overriding and conflict resolution have been studied in detail by Bertino [Bertino and Ferrari 2002] and Damiani [Damiani et al. 2000]. Damiani et al.[Damiani et al. 2000] was one of the first works that utilized XPath as a means of specifying access constraints. They proposed an algorithm based on tree labeling and annotation to compute a security view of the data for an user group. But the work proposed that the view be materialized - a potentially complex and computationally expensive task. Miklau et al. [Miklau and Suciu 2003] have studied cryptographic techniques for securing XML content. 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. [Fan et al. 2004] is the most complete work on XML access control to date. The authors introduced a notion of specifying access 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 did not require materialization of individual views. However the paper limited the definition of views to either hiding a node directly or based on an XPath condition, and did not consider operations involving manipulation of the hierarchy and structural relationships.

1.4 Our Contributions

Summarizing the challenges in access control in XML and the pros and cons of the techniques available in the literature, our focus here is to provide a framework in the database engine to enforce such access constraints on query evaluation, without materializing the security views.

Formally, we define the problem of XML security view specification and rewrite as:
Given an original XML schema S_0 and a sequence of primitives Sp_L in a security view specification language L that define a role-based security view S_v, develop a framework F such that for each query q issued on S_v and each database D_0 conformed to S_0, $F(Sp_L, D_0, q) \equiv q(S_v)$.

Anti-inference is not the focus of this paper and will be discussed briefly in future research directions.

We propose an infrastructure for access control on XML documents by specifying access constraints in the form of virtual security views and enforcing the access constraints via query rewrite. The main contributions of our work can be summarized as follows:

—We introduce an algebraic security view specification language SSX, by adopting and modifying a graph editing language to enable conditionally hiding and reorganizing XML elements/subtrees.

—We propose SAS - an annotated schema to represent the security constraints internally.

—We propose SQR, a rule-based security query rewrite algorithm for enforcing the access constraints, using only the information in the SAS.

—We conduct extensive experimental evaluation on various benchmark databases and prove that our technique is both effective and efficient.

The rest of the paper is organized as follows: we describe the infrastructure of ACXESS in Sec.2, followed by the definition of terms used in the rest of the paper; we then introduce the security view specification language SSX in Sec.3; the two primary components of our method, viz. the annotation process and the rewrite algorithm are discussed in Sec.4 and 5, respectively; the experimental results are presented in Sec.6, followed by the conclusion and future work in Sec.7.

2. ACXESS SYSTEM INFRASTRUCTURE AND PRELIMINARIES

We introduce ACXESS (Access Control for XML with Enhanced Security Specifications), a framework for specifying, presenting and enforcing access constraints on XML documents. The infrastructure of ACXESS is as shown in Figure 2. The system can be divided into two components: **security view construction** (on the left) and **security view based query answering** (on the right).

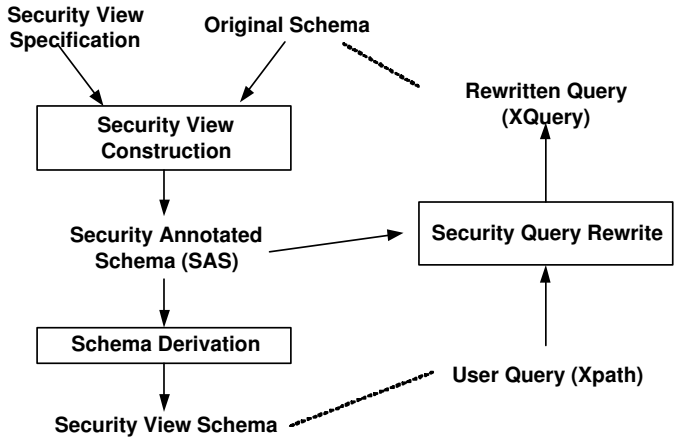


Fig. 2. The Infrastructure of ACXESS

In the **Security View Construction** process, a security view specification language SSX (Security Specification for XML) is used for specifying security views. A security view is represented internally in forms of a Security Annotated Schema(SAS). The view schema is visible to the user group to whom the security constraints are to be applied, and it is the only schema that is made available to the user group. The **Security View Based Query Answering** process rewrites the user queries (in XPath) into target queries (in XQuery) against the original schema

by using the information in the SAS. The resultant XQuery expressions reflect the access constraints imposed on the user group and are evaluated against on the base data.

2.1 Preliminaries

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. Formally we define an XML tree as follows

DEFINITION 1. *An XML document is a tree* $t < V_t, E_t, r_t >$ *over a finite alphabet* Σ *where*

—V_t *is the node set and* E_t *is the edge set.*

—$r_t \in V_t$ *is the root node of t.*

—*each node* $n \in V_t$ *has a label from* Σ *denoted as label(n).*

DEFINITION 2. *Given an XML tree* $t < V_t, E_t, r_t >$, $s < V_s, E_s, r_s >$ *is a subtree of t if* $V_s \subseteq V_t$ *and* $E_s = (V_s \times V_t) \cap E_t$.

DEFINITION 3. *Given an XML tree* $t < V_t, E_t, r_t >$ *and a node* $n \in V_t$, $s < V_s, E_s, r(s) >$ *is the subtree rooted at n and exactly containing all its descendants* [2] *if* $r(s) = n$ *and* $V_s \subseteq V_t$ *and* $\forall$ *descendant nodes A of n,* $A \in V_s$ *and* $E_s = (V_s \times V_t) \cap E_t$. *We call s a complete subtree of t at n, and represent it as* sub_t^n.

A typical XML structure can be represented as a tree, as shown in Figure 1. XML structure can be specified using either a DTD (Document Type Definition) or an XML schema. 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.

To facilitate the discussion in this paper, we define a Simple Path Expression as follows:

DEFINITION 4. *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 SPEs.*

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

For the purposes of definition of the security view specification language we adopt the definition of 'Pattern Trees' as defined by Jagadish et.al [Jagadish et al. 2001] to represent an XPath query on a XML document. A pattern tree p is represented as $p < V_p, E_p, r_p, ret_p >$ where V_p, E_p, r_p, ret_p represent the set of vertices, edges, the root node and the return node respectively. We also extend the notion of a 'Witness Tree' as defined by Jagadish et.al [Jagadish et al. 2001] to represent the evaluation of an XPath query on a XML document as follows:

DEFINITION 5. *Given an XML tree* $t < V_t, E_t, r_t >$ *and a pattern tree* $p < V_p, E_p, r_p, ret_p >$, *a witness tree* $wit(p,t) = w < V_w, E_w, r_w >$ *is defined by a total mapping* $h{:}p \rightarrowtail t$ *from the pattern tree p to the XML tree t such that:*

[2] A descendant node of any node A is any node below A in a tree model of a document, where "above" means "toward the root".

—$(v \in V_w \Leftrightarrow (v \in V_t \wedge \exists u \in V_p(v = h(u))$, *i.e n matches some node in the pattern tree p.*

—$(u, v) \in E_w \Leftrightarrow (u \in V_t \wedge v \in V_t \wedge \exists m_1, ...m_k \in V_t((u, m_1) \in E_t \wedge, (m_1, m_2) \in E_t \wedge, ..., (m_k, v) \in E_t) \wedge \exists (u', v') \in E_p \wedge u = h(u') \wedge v = h(v'))$ *. i.e. h preserves the structure of p.*

—$ret_w \in V_t \wedge ret_w = h(ret_p)$.

A witness tree contains nodes in the XML tree and satisfy the pattern. If a given pattern can be embedded in an XML tree in multiple ways, multiple witness trees are obtained, one for each such embedding.

DEFINITION 6. *Given an XML tree* $t < V_t, E_t, r_t >$*, and a pattern tree* $p < V_p, E_p, r_p, ret_p >$*, we define the result obtained by evaluating p on t as* $eval(p, t) = wit(p, t)$.

3. XML SECURITY VIEW SPECIFICATION

3.1 Security View Specification Language

Given access constraints such as those in Example 1 and other similar cases, we can summarize the basic requirements of a security view specification language as a language that should be able to achieve the following tree transformations:

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

Most of these are basic graph editing operations as proposed in [Atzeni and Mecca 1997]. We take the core of the graph editing language and introduce our Security Specification Language for XML (SSX) in the form of a set of primitives. [3]

SSX is a security view definition language. Each primitive takes an XML tree as input, and outputs an XML tree. The semantics of the primitives are defined as follows [4]:

create(destSPE, newName)

The **create** primitive creates a new element with tag 'newName', as a child of each element that matches the 'destSPE' in the input tree. Formally,

Given an input XML tree t, the resultant security view of the create operation is a tree Sec_v: for every $w \in eval(destSPE, t)$, there is a new node n with labor $newName$, such that $n \in V_{Sec_v} \wedge (ret_w, n) \in E_{Sec_v}$.

delete(destXPath)

[3] As a first step towards building the framework we assume that a schema is available for the XML document being secured and it is acyclic.

[4] parameters within square brackets are optional

The **delete** primitive removes the subtrees rooted at the elements that matches the 'destXPath' in the input tree. If the root of the input XML tree matches the 'destXPath' expression then the entire tree is removed. Formally,

Given an input XML tree t, the resultant security view of the delete operation is a tree Sec_v: for every $w \in eval(destSPE, t)$, $V_{Sec_v} = V_t - V_{sub(t,r_w)} \wedge E_{Sec_v} = (V_t \times (V_t - V_{sub(t,r_w)})) \cap E_t$.

copy(sourceXPath, destSPE, [newName],[scope], [preserve])

For each element that matches the scope, the **copy** primitive creates an identical copy of the subtrees rooted at the nodes that match the 'sourceXPath' in the original tree with respect to the 'scope', and makes them the children of the elements that match the 'destSPE' with respect to the 'scope' in the input tree. If a new name ('newName') is provided, a new element tag is assigned to the root element of the copied subtrees; otherwise, the original element tag is used. The default value for 'scope' is '/'. 'Preserve' is a flag that specifies if the copy primitive should preserve (the default) or break the document order between instances being copied.

[[[the original formal definition of copy looks too messy, have not fixed it yet. <==]]]

rename(destSPE, newName)

The **rename** primitive assigns a new name ('newName') to the elements that matches the 'destSPE' in the input tree. Formally

Given an input XML tree t, the resultant security view of the create operation is a tree Sec_v: for every $w \in eval(destSPE, t)$, ret_w's tag is replaced by $newName$ in Sec_v.

[[[another way to clarify the definition of the semantics is to give an example for each primitive.]]] <==

To facilitate the reading/writing of an SSX sequence, a set of secondary primitives can be derived from the primitives. For example, **Move(sourceXPath, destSPE, [newName], [scope], [preserve])**. The parameters of the **move** primitive are the same as that of the copy primitive. The **move** primitive is the concatenation of the **copy** primitive and the **delete** primitive: $Move(p, d, n, s, ps) = copy(p, d, n, s, ps) \bullet delete(p)$.

3.2 Specifying Security Views Using SSX

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 security view defined by the SSX sequence.

[[[we need to use data as example, not schema, since the view definition language really works on data and define view.]]] <==

EXAMPLE 2. *Let's reconsider the security concerns on the quiz structure in Example 1. Table I provides the security view specification sequences (written in SSX) that specify the security views whose schemas are shown in Figure 1 (b). Some of the specific capabilities of SSX can be noticed here: the conditional deletion in the "student" level removes access to all non-active quizzes; in order to create a "flat" version of the objectbank, the "other-instructor" level requires the creation of a node*

Security Level	Security View Definition
"other-instructor" level	delete(/quiz/Access) delete(/quiz/Assessment) create(/quiz,questionbank) copy(/quiz/objectbank//item,/quiz /questionbank,'', /quiz, false) delete(/quiz/objectbank)
"statistician" level	delete(/quiz/Access) delete{/quiz/Assesment} create(/quiz,solutions) copy(/quiz/objectbank//item/solution, /quiz/solutions, '', /quiz, false) delete(/quiz/objectbank)
"student" level	delete(/quiz/[not(Access/startdate< currdate and Access/enddate>currdate)]) delete(/quiz/Access) delete(/quiz/Assessment) delete(/quiz/objectbank//solution)

Table I. SSX Sequence Defining the Security Views for the Access Levels "Other-instructor", "Statistician" and "Student"

followed by copying desired nodes under the new node "questionbank"; and for the "statistician" level, the association between items and their solutions is broken by removing the solutions from the subtree rooted at item and copying them to a new location without preserving the order.

<== [[[the following section that discuss the restriction gives pretty negative impression. we need to think about how to state these, or move them to appendix?]]]

3.3 SSX Restrictions

Based on the semantics of the graph editing language, a few extensions and restrictions are introduced in SSX, to make sure that the XML security views specified in SSX are meaningful. It should be noted that SSX is not a complete query language or transformation language, so these restrictions are only to ensure that the transformations are mainly for security specifications. These restrictions include the following:

(1) The **scope** parameter in the copy operation introduces a "grouping" effect. With a scoped copy, for example, we can group the test questions by section and grant access to the same to a user group. In addition, we restrict the scope of a **copy** operation to be an SPE that evaluates to a common ancestor of the elements that are identified by the 'sourceXPath' and the 'destSPE'.

(2) We restrict the destination path of **create, copy** and **rename** to be an SPE. This means the parent of a newly created element, the element to be renamed, and the destination of the copy has to be unconditionally identified. All security specifications that are restricted by this constraint can be expressed alternatively by a sequence of operations.

(3) We restrict the source path of the **copy** operation to be evaluated against the source tree, rather than the input tree to the operation. This means that, one cannot copy anything that has been modified by operations prior to the **copy** operation in the SSX sequence, to a new location. This constraint also specifies that the condition on the source path of the copy operation has to be evaluated against the source data. Although seemingly a limitation, this restriction is needed to ensure the tractability of the rewrite operations.

(4) We restrict the target node of a **rename** operation to be a node that has not been modified. This restriction does not limit the expressive 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** operations.

(5) Finally, there is an inherent chronological order among the operations, which has to be taken into consideration in the access control enforcement.

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

[[[the order dependency section was removed by Jit in the first draft, I think it is important. our later discussion about the ordering in SAS and in rewrite all refers to this.]]] <==

4. SECURITY ANNOTATED SCHEMA

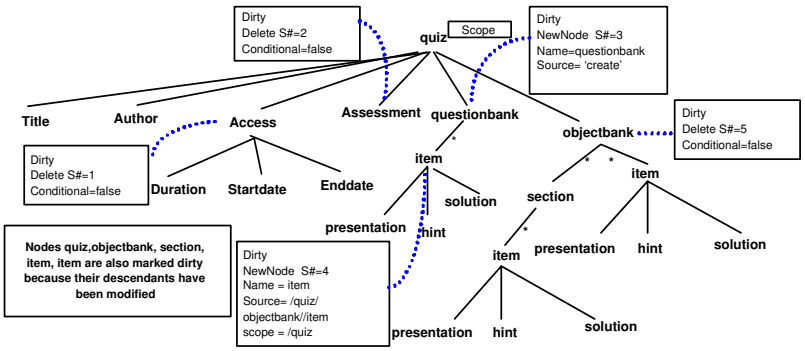
The Security Specification Language for XML (SSX) defined in the previous section assists the DBA in specifying access constraints.

In both relational and XML database models, multiple approaches have been proposed for enforcing access constraints. The ones that are most commonly adopted are: post-query filtering, materialized view and query rewrite. Materialized views have been successfully used in enforcing access control for XML [Bertino and Ferrari 2002; Damiani et al. 2000]. The main problem with this method is that it is computationally expensive to generate and maintain materialized views that implement a security specification, as shown by Fan et al [Fan et al. 2004]. In a real-time situation with multiple security levels and multiple views at each level, this can be non-realistic. Similarly, post-query filtering with static analysis to determine safe queries have been studied by Cho et.al [Cho et al. 2002]. The time taken to check each and every result node can be exponential and this method is realizable only because of the static analysis to determine 'safe' queries. But it is difficult to enforce value based access control and conditional access control. Further, even where static analysis can be performed, the time complexity of running such algorithms is very high. These disadvantages in post-query filtering and view materialization make query rewriting a promising choice.

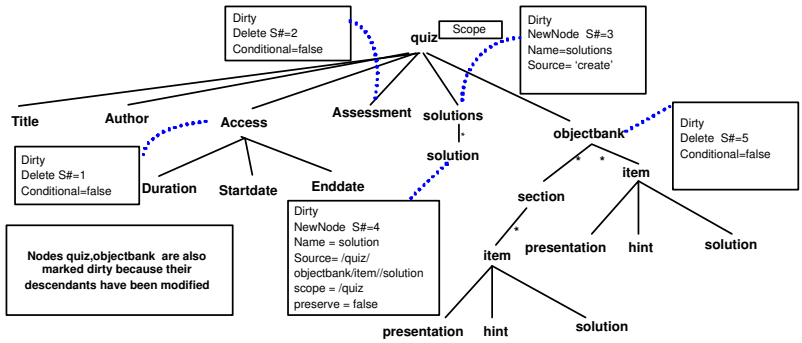
This necessitates a mechanism to represent the virtual view as part of the meta data. We propose the use of schema as the vehicle for such internal representation, to facilitate query answering and rewriting. We call such internal representation SAS (Security Annotated Schema). The rest of the section describes the various annotations that make up the SAS and an algorithm to construct the same. Access constraint enforcement based on query rewrites will be discussed in Section 5.

4.1 SAS

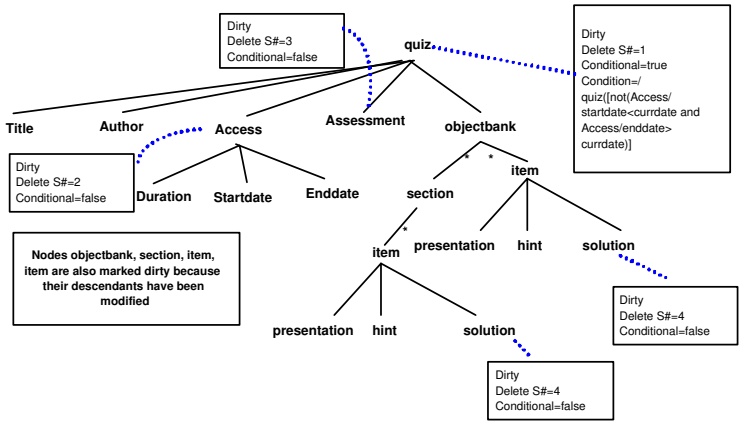
We introduce a set of schema annotations, to reflect the security view definition on the schema tree structure. An annotation is a special XML node with attributes to identify the primitive that causes the modification and other details of the changes performed by the primitive. All the annotations are associated with the element node that was modified. Also note that the SSX primitives are applied on a XML



Security Annotated Schema for the ` other-instructor' Security View shown in Figure 1



Security Annotated Schema for the ` statistician' Security View shown in Figure 1



Security Annotated Schema for the ` student'' Security View shown in Figure 1

Fig. 3. Security Annotated Schema of the Security Views Shown in Figure 1 (b)

data tree and are instance dependent. To precisely represent the effect of each SSX primitives, it is critical that the parameters of the primitives, such as condition in the destPath in the delete primitive, are recorded internally in SAS to facilitate the enforcement at a later stage.

The annotations in SAS are as follows:

—**Delete Annotation** identifies that the node has been removed, with additional parameters storing the condition in case the node was removed conditionally.

—**NewNode Annotation** identifies that the node has been newly constructed. A **NewNode** annotation with parameter 'name' can be the result of a create or a rename operation. A copy operation also results in a **NewNode** annotation, with additional parameters identifying the 'sourcePath', 'scope' and whether the ordering among subtrees is to be preserved.

—**Scope Stamp** A node N is stamped 'scope' if any of its descendants has the **NewNode** annotation resulting from a **copy** operation, and its scope parameter is a path that matches to N.

—**Dirty Stamp** A node is stamped 'dirty' if any of its descendants (including itself) is annotated.

—**Chronological Operation Sequence # (S#)** This is a chronological sequence# assigned to each operation in an SSX sequence. It reflects the potential logical dependency of an operation on preceding operations in the sequence.

EXAMPLE 3. *The SAS that derives the 'other-instructor' security view tree structure in Figure 1 (b.3) is shown in Figure 3. The annotation associated with the 'questionbank' node reveals that it is a newly created node. Similarly the annotation associated with the node 'item' indicates that it was created via the copy operation with the source '/quiz/objectbank//item' and scope '/quiz'. In addition, the scope stamp associated with the 'quiz' node reflects that it has been used as a scope. The Chronological Operation Sequence # (S#) associated with the NewNode annotation for 'item' identifies that it is the result of operation #4 in the SSX sequence. The delete annotations with the conditional value set to 'FALSE' indicate that the corresponding nodes have been deleted unconditionally. Note that a dirty stamp simply indicates that some nodes at or below that node has been modified. Figure 3 also shows the SAS associated with the 'statistician' and the 'student' user groups respectively. The annotation associated with the 'quiz' node in the security annotated schema for the 'student' view reveals that the node has been conditionally deleted with the condition '*$/quiz[not(Access/startdata < Access/currdate$ *and* $Access/enddate > Access/currdate)]$'.*

5. ENFORCING ACCESS CONSTRAINTS IN QUERY ANSWERING

The security view is defined by an SSX sequence and can be derived easily from the SAS that represents the SSX sequence and be exposed it to the end users. The next and the most intriguing problem is to enforce such security constraints when answering user queries.

In both relational and XML database models, multiple approaches have been proposed for enforcing access constraints. The ones that are most commonly adopted are: post-query filtering, materialized views and query rewriting. As analyzed above, the computational complexity and other disadvantages in post-query filtering and view materialization makes query rewriting a promising choice.

5.1 Security Query Rewrite (SQR)

We choose to rewrite the user queries against security view schema into queries against the source schema. The challenge is due to the tree structure of both the XML data and the XML query and the tree pattern matching nature of XML query evaluation as well as the following facts about the way security constrictions are specified and the manner in which the SAS is constructed:

—The uncertainty introduced by the wild card '⋆'and '//'.

—The impact of the annotations added to a specific element is on the subtree rooted at that element.

—There is a partial ordering among the annotations, *i.e.,* some annotations may depend on others, and during 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 instance.

As the first step, we assume that the user queries are written in XPath. The rewritten query cannot be specified in XPath, since XPath is not expressive enough to handle the grouping generated by the 'scope' (specified in a copy primitive), and does not have the ability to create new structures (specified in the copy, rename and create primitives). Hence we choose XQuery as the query language of the 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 subtrees.

5.2 Security Query Rewrite Algorithm

The security query rewrite process (SQR) is rule-based. Given an XML database with schema S_0, a security view V (identified by SAS S_v), and an XPath expression $p = t_1[c_1]/t_2[c_2]\ldots /t_n[c_n]$ against V, where t_is are tags (element tags, attribute name or wildcard) and c_is are XPath expressions serving as branching predicates, SQR rewrites p to q such that $q \equiv p$.

Class	**Rule**	**Short Description**
Path Rules	Rule 1	handle branching conditions
	Rule 2	handle sub-query in clean subtrees
	Rule 3	handle "//"
	Rule 4	handle dirty node on linear XPath
	Rule 5	handle dirty node on the tail of XPath
Operation Rules	Rule 6	handle unconditionally deleted node
	Rule 7	handle conditionally deleted node
	Rule 8	handle subtrees from *copy* operation
	Rule 9	handle new node from *create* operation
	Rule 10	handle new node from *rename* operation

Table II. The Rewrite Rules in SQR

To facilitate the discussion of the rewrite rules and the algorithm, we define a recursive procedure as follows:

$q = SQR_{S_v}(p, [vb], [vb'])$:: The function SQR translates an XPath query p to an XQuery expression q, referencing the annotations in an SAS S_v and under a specific environment (if apply), represented by a set of variable-bindings vb. vb' represents the new variable-bindings in q.

We develop a set of rewrite rules to transform p to q. The rewrite rules can be summarized, based on their functions in the SQR process, into path rules and operation rules, as shown in Table II.

The skeleton of the rewrite algorithm is presented in Figure 4 [5]. The rules and the algorithm only deals with elements, however, they can be easily adapted to handle attributes.

```
SQR(p:Parsed XPath, vb:  Input Environment, vb':  Output Environment) {
  nexttoken = Obtain next token from input XPath;
  if (nexttoken = ``//'') Apply RULE 3
  else if (currtoken is XPath condition c) Apply RULE 1 ;
  // the nexttoken is a string now.
  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 2 for HCE;
    else if (annotation A in curnode)
      switch A {
        case A = 'uncond delete':  Apply RULE 6;
        case A = 'cond delete':  Apply RULE 7;
        case A = 'copy':  Apply RULE 8;
        case A = 'create':  Apply RULE 9;
        case A = 'rename':  Apply RULE 10;
        }
    else if (Dirty node with no annotation) Apply RULE 4;
    if (current element is last Token) Apply RULE 5
  }
  return q;
} // end SQR
```

Fig. 4. The SQR Algorithm

The algorithm accepts as input an XPath query p that needs to be rewritten. It iterates through p and walks the SAS tree based on the tokens found in p. Annotations, if found during the tree walk, are appropriately handled to generate the rewritten XQuery expression q. In the 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 an 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.

[[[the following is implementation details, should be shorting into a sentence or two if space is needed]]] Following the rules, the algorithm needs to be provided with an environment that includes information such as <==

[5] applying a rewrite rule consumes (part of) p and the input environment vb, generates rewritten query expression q and new environment vb'.

—*currnode:* The current node of the schema that is being processed has to be provided to keep track of the tree walk.

—*current variable count (vcount):* This is necessary for keeping track of variable bindings and to generate the appropriate variable binding for each element in the XQuery expression.

—*tailexpr:* The tail expression keeps track of the variable binding of the element, with respect to which, the variable binding for the current element has to be generated.

—*indelete:* The 'indelete' flag determines whether processing is within a delete annotation (since deletes, especially conditional deletes have to be handled differently than the other operations. See Rule 7).

In addition we keep track of scope paths and variables associated with them. We refer to the environment parameters as env.*paramname*, e.g., `env.curnode`, etc in the algorithm. To process a user XPath query, rewrite needs to be bootstrapped as follows:

```
set env{curnode = node(xs:schema), vcount=1,
    tailexp='', 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 and exhaustive experimentation indicates the algorithm to be a complete representation of the rewrite rules. The correctness of the algorithm can be proved inductively over the rewrite rules, and is presented later in this section.

A prototype implementation of the security view definition, schema annotation and the SQR algorithm (compatible with XPath1.0) [Mohan et al. 2005], has been developed using Java, JAXP and Galax [J.Simeon and M.Fernandez] and is available for download at [Mohan et al.].

5.3 Rewrite Rules

The core of the security query rewrite process is the rewrite rules. A conceptual presentation of the rules used by *SQR* is provided below.

5.3.1 *Path Rules.* The *Path Rules* deals with (sub) XPaths in the XPath query to be rewritten, and further simplifies the XPath expression by breaking it into parts, or keeping it as is, or rewriting it into XQuery expressions, based on the access constraints to be enforced.

<== [[[there is some inconsistency on the presentation, which is w.r.t. *vb.* in the above *vb* is the whole environment, including things like curnode, current variable count, etc..... in the rules, *vb* is defined as if it is only variable binding, since we use notions like $vb_5 \cup \{i, j\}$. I have not thought a very good solution yet.]]]

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 predicate, taking the form of p_3[op p_4] (please note that 'op p_4' is optional), the XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

```
for $i in SQR_{S_v}(p_1, vb, vb_1)
for $j in SQR_{S_v}(t_1, vb_1 ∪ {i}, vb_5)
where SQR_{S_v}(p_3, vb_5 ∪ {i, j}, vb_3) op
      SQR_{S_v}(p_4, vb_5 ∪ {i, j}, vb_4)
return SQR_{S_v}(p_2, vb_5 ∪ {i, j}, vb_2)
```

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. 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.

EXAMPLE 4. *Let's consider the query '/quiz[Author='Charles']/Title' as applied on the 'other-instructor' security view. Applying Rule 1 the condition "Author= 'Charles' " must be treated as a separate XPath Expression whose equivalent rewritten XQuery Expression appears in the 'WHERE' clause of the final rewritten expression. This results in the generation of the following query*

```
for $i in doc(''data/quizzes.xml'')/quiz
where SQR_{S_v}(Author, vb_0 ∪ {i}, vb_1) = 'Charles'
return SQR_{S_v}(Title, vb_0 ∪ {i}, vb_2)
```

What $SQR_{S_v}(Author, vb_0 \cup \{i\}, vb_1)$ and $SQR_{S_v}(Title, vb_0 \cup \{i\}, vb_2)$ turn out to be depends on the S_v.

A 'dirty' stamp associated with a node in the SAS identifies that either the node itself or its descendant(s) has been modified by the SSX sequence. Before presenting the rewrite rule that responds to 'dirty' stamps, we first define the following important notion:

DEFINITION 7. *Given an SAS S_v and an SPE expression $p = t_1/t_2 \ldots /t_n$, if there exists t_k, such that t_{k-1} has a 'dirty' stamp and none of t_k, t_{k+1}, ..., t_n has a 'dirty' stamp, we call t_k the* highest clean element (HCE) *of p on S_v, and $p_1 = t_1/t_2 \ldots /t_{k-1}$ the* prefix path *w.r.t. S_v and $p_2 = t_k/t_{k+1} \ldots /t_n$ the* suffix path *w.r.t. S_v.*

EXAMPLE 5. *Let's consider the query '/quiz/Author' on the 'student' security view. The node 'quiz' is marked 'dirty' while the node 'Author' is not. Hence the node 'Author' is the HCE for the above query in the 'student' view. Similarly for the query '/quiz/objectbank/item/hint' the node 'hint' is the HCE.*

RULE 2. *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 $q = SQR_{S_v}(p, vb, vb')$ of p as follows:*

```
for $i in SQR_{S_v}(p_1, vb, vb_0)
return $i/p_2
```

The notion of HCE *enables us to leave the suffix path "*as is*" in the rewrite. Since the nodes below the HCE are not modified they can be directly evaluated as a subexpression in the resultant XQuery expression without any rewrites.*

EXAMPLE 6. *Since both 'Author' and 'Title' are not 'dirty', the intermediate results in Example 4 can be further rewritten using Rule 2. The final rewritten XQuery expression for /quiz[Author='Charles']/Title is*

```
for $i in doc(''data/quizzes.xml'')/quiz
where $i/Author = 'Charles'
return  $i/Title
```

RULE 3. *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 $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

```
for $i in SQR_{S_v}(p_1, vb, vb_0)
return
    {SQR_{S_v}(t1/p2, vb_0 ∪ {i}, vb_1)},
        ⋮
    {SQR_{S_v}(tn/p2, vb_0 ∪ {i}, vb_n)}
```

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

<== [[[the rule above may not be 100% correct, at least, the rewritten query generated by this rule turns out to be illegal queries. need a fix.]]]

Uncertainty due to '//' can be handled by walking the tree and searching all possible paths for the node of interest. Given that as a first step, we have assumed that the schema is acyclic, we have optimized the rewrite algorithm by ensuring that given a pair of nodes 'A' and 'B', all possible paths from 'A' to 'B' are precomputed and stored in the SAS.[6] Hence when the expression 'A//B' is encountered during a rewrite the uncertainty associated with the '//' expression is resolved by obtaining the requisite paths from the SAS. The final rewrite is obtained as a union of the rewrite of the various stored paths between 'A' and 'B'. Similarly, Rule 3 can be used to handle the wild card character '$\star$'.

EXAMPLE 7. *Let's consider the query '//item' as applied on the 'student' security view. Looking at the SAS we can find that we can reach the 'item' node via the paths '/quiz/objectbank/item' and '/quiz/objectbank/section/item'. Based on the above rule the XPath query is rewritten as*

```
return
{SQR_{S_v}(/quiz/objectbank/item, {}, vb_1)},
{SQR_{S_v}(/quiz/objectbank/section/item, {}, vb_2)}
```

The node 'quiz' has been conditionally deleted and hence is rewritten based on the rule for 'conditional delete'(Rule 7). The final rewritten XQuery expression for '//item' is

[6]This has been efficiently carried out by recursively walking the tree at the end of the SAS generation process.

```
return
{
    for $q in doc(‘‘data/quizzes.xml’’)/quiz
    where $q/Access/Startdate < = ‘currdate’
        and $q/Access/Enddate >= ‘currdate’
    return for $q2 in $q/objectbank/item
    return <item> {$q2/text} {$q2/hint} </item>
},
{
    for $q3 in doc(‘‘data/quizzes.xml’’)/quiz
    where $q3/Access/Startdate < = ‘currdate’
        and $q3/Access/Enddate >= ‘currdate’
    return for $q5 in $q3/objectbank/item
    return <item> {$q5/text} {$q5/hint} </item>
}
```

The ‘item’ node is reconstructed based on the rule for result reconstruction that is described below.

RULE 4. *Given an SAS S_v, and an SPE $p = t_1/t_2/\ldots/t_n$ 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. The XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

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

EXAMPLE 8. *Let’s consider the expression ‘/quiz/Author’ on the ‘other-instructor’ security view. Since the node ‘quiz’ has been marked ‘dirty’, but otherwise has no annotations on it, we can directly proceed to the next element.*

```
for $q in doc(‘‘data/quizzes.xml’’)/quiz
return  SQR_{S_v}(Author, {$q}, vb_1)
```

The resultant XQuery Expression is

```
for $q in doc(‘‘data/quizzes.xml’’)/quiz
return  $q/Author
```

RULE 5. *Given an SAS S_v and an SPE $p = t_1/t_2\ldots/t_n$ under environment vb, if t_n is ‘dirty’ and has children $cld_1,\ldots cld_m$ in S_v, the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as follows:*

$$\texttt{for \$i in } SQR_{S_v}(t_1/t_2\ldots/t_{n-1}, vb, vb_0)$$
$$\texttt{return } < t_n >$$
$$SQR_{S_v}(cld_1, vb_0 \cup \{i\}, vb_1)$$
$$\vdots$$
$$SQR_{S_v}(cld_m, vb_0 \cup \{i\}, vb_m)$$
$$< /t_n >$$

When the tail of the SPE has a ‘dirty’ stamp, rather than stopping at the node and returning the whole subtree rooted at the node, 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. This is achieved by recursively rewriting each and every child node of the node being constructed.

EXAMPLE 9. *Let's consider the query '/quiz/objectbank/item' on the 'student' security view . The node 'quiz' has been conditionally deleted and hence is rewritten based on the rule for 'conditional delete'(Rule No 7-described below). The part '/quiz/objectbank' is translated as*

```
for $q in doc(''data/quizzes.xml'')/quiz
where $q/Access/Startdate < = 'currdate'
    and $q/Access/Enddate >= 'currdate'
return  for $q1 in $q/objectbank
return {SQR_{S_v}(item, {$q, $q1}, vb_1)},
```

Now the node 'item' has been marked 'dirty' and hence the children have to be individually reconstructed. This ensures that the child node 'solution' which has been unconditionally deleted is not returned as part of the result. Further since the node is being manually reconstructed the printing of the open and closing tags of the 'item' node should also be enforced by the resultant XQuery expression. The final rewritten query for the expression '/quiz/objectbank/item' is

```
for $q in doc(''data/quizzes.xml'')/quiz
where $q/Access/Startdate < = 'currdate'
    and $q/Access/Enddate >= 'currdate'
return for $q1 in $q/objectbank
for $q2 in $q1/item
return <item> {$q2/text} {$q2/hint} </item>
```

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

RULE 6. **Unconditional Delete Rule:** *Given an SAS S_v and an SPE $p = t_1/t_2 \ldots /t_n$, if t_i ($0 \leq i \leq n$) is an unconditional delete annotation in S_v, the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

```
return ()
```

If a node has an unconditional delete annotation any query that retrieves the node or its descendants should result in an empty set. This is achieved by rewriting the XPath expression using the XQuery empty sequence ().

EXAMPLE 10. *Let's consider the query '/quiz/Assessment' on the 'statistician' view. The node 'Assessment' has been unconditionally deleted and hence the resultant XQuery expression is*

```
return ()
```

RULE 7. **Conditional Delete Rule:** *Given an SAS S_v, and an SPE $p = t_1/t_2 \ldots /t_n$ under environment vb, if node t_k ($k < n$) is annotated by a conditional delete operation, with condition cond (an XPath expression), the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

```
for $i in SQR_{S_v}(t_1/t_2 .../t_k, vb, vb_0)
where count(SQR_{S'_v}(cond, vb, vb_1)) = 0
return SQR_{S_v}(t_{k+1}/.../t_n, vb_0 ∪ {i}, vb_2)
```

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

The condition used to delete a node is a valid XPath expression and is rewritten as a condition in the resultant XQuery expression. Since a given node has been conditionally deleted, instances of the node which satisfy the condition should not be a part of the final result set. This is achieved by rewriting the condition to calculate the complement of the condition used to delete the nodes (Implemented using the count operator). Further care should be taken to ensure that the rewrite process for the condition takes into effect only the changes performed by the primitives before the conditional delete in question. This is automatically handled during the rewrite process by making use of the chronological sequence #.

EXAMPLE 11. *Consider the query '/quiz/Author' on the 'student' security view. The node 'quiz' has been conditionally deleted by using the condition 'Access/startdate <= currdate and Access/enddate >= currdate'. Only those 'quiz' instances for which the complement of the above condition is satisfied should be returned. Instead of using the 'count' operation to compute the number of nodes returned by the rewrite of the above condition and returning only those quiz nodes for which the count is equal to zero the rewrite has been optimized as the delete condition has the negation operator 'not'. This is identified during the rewrite process and the negation is removed to get the complement. The final rewritten XQuery expression for the above query is*

```
for $q in doc(''data/quizzes.xml'')/quiz
where $q/Access/Startdate < = 'currdate'
    and $q/Access/Enddate >= 'currdate'
return  $q/Author
```

RULE 8. **Copy Rule:** *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 operation copy($t_1/t_2/.../t_k/.../t_x$, $t_1/t_2/t_k/.../t_{y-1}$, t'_x, $/t_1/t_2/.../t_k$, ps), where $(t_x = t_y \wedge t'_x = \phi) \vee t'_x = t_y$, the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

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

The copy operation constructs new subtrees in the schema tree structure. It is annotated on the new subtree root, with all the required information for the rewrite: source path, scope, preserve, etc. When a query retrieves information from a copied node or its descendants, data is retrieved from the source, along the source path. The retrieved data is compliant to any annotations on the descendant subtree.

EXAMPLE 12. *Consider the query '/quiz/solutions/solution' on the 'statistician' security view. The node 'quiz' has been marked 'dirty' but has no annotation and hence the rewrite algorithm can directly proceed to the next element in the XPath expression. Further the node 'solutions' was newly created and as per Rule 9 (Create Rule) described below, the rewrite algorithm can directly proceed to the next element 'solution'. The element 'solution' was obtained by copying the solution nodes along the paths (/quiz/objectbank/item/ solution) and (/quiz/objectbank/section/item/solution). Hence the rewritten XQuery expression directly retrieves these nodes in the final result. The final rewritten XQuery expression is*

```
for $q in doc(''data/quizzes.xml'')/quiz
return  {for $q1 in $q/objectbank/item/solution
             return $q1},
        {for $q2 in $q/objectbank/sectionitem/solution
            return $q2 }
```

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

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

RULE 10. **Rename Rule:** *Given an SAS S_v, and an SPE $p = t_1/t_2 \ldots /t_n$ under environment vb, if node t_y ($y < n$) is annotated as a "new node", generated by operation rename($/t_1/t_2/\ldots t_{y-1}/t_x$, t_y), the equivalent XQuery expression $q = SQR_{S_v}(p, vb, vb')$ is defined as:*

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

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

EXAMPLE 13. *Consider the translation for the query '/quiz/questionbank//item[hint]' for the 'other-instructor' view. The node 'questionbank' is a newly created node and is bypassed in the rewrite process which moves from the rewrite for the node 'quiz' directly to handling the rewrite for the path to the 'item' node stored in the SAS. The final rewritten XQuery expression is*

```
for $q in doc(''data/quizzes.xml'')/quiz
for $q1 in $q/objectbank
for $q2 in $q1/item | $q1/section/item
where count(for $q3 in $q2/hint return $q3) > 0
return $q2
```

Level	Rewritten query

"other-instructor" level	`for $q in doc("data/quizzes.xml")/quiz`
	`return`
	`{`
	` for $q1 in $q/objectbank`
	` for $q2 in $q1/item`
	`  where count(for $q3 in $q2/hint return $q3)>0`
	` return $q2`
	`},`
	`{`
	` for $q4 in $q/objectbank`
	` for $q5 in $q4/section/item`
	`  where count(for $q6 in $q5/hint return $q6)>0`
	` return $q5`
	`}`

"statistician" level	`return ()`

"student" level	`for $q in doc("data/quizzes.xml")/quiz`
	`where $q/Access/Startdate <= currdate`
	`and $q/Access/Enddate >= currdate`
	`return`
	`{`
	`  for $q1 in $q/objectbank/item`
	`  where count(for $q2 in $q1/hint return $q2) > 0`
	`  return <item> {$q1/text} {$q1/hint} </item>`
	`},`
	`{`
	`  for $q3 in $q/objectbank/section/item`
	`  where count(for $q4 in $q3/hint return $q4) > 0`
	`  return <item> {$q3/text} {$q3/hint} </item>`
	`}`

Table III. Rewritten queries for query $/quiz//item[hint]$ on different access levels

The condition 'hint' is translated into an XQuery expression that evaluates the number of matches for the given condition and returns true only when there is at least one node matching the condition.

Let's consider the rules together and examine a query on all three security views.

EXAMPLE 14. *Following the rewrite rules described above, let's take a look at the query '/quiz//item[hint]', which finds all items in the quizzes with a hint. When users in different user groups issue this query, it will be rewritten differently, based on the security constraints specified for the user group. The rewritten queries for the user groups(see Table I) are as shown in Table III. For the 'student' group, we find that there are two different paths to 'item' nodes and both are reconstructed. The results are reconstructed to hide the 'solutions', and the conditional delete operation in the access control definition is taken care of by a "where" clause. For the 'statistician' group, when SQR detects that there are not paths to the node 'item' for the user group, it is obvious that the result is empty and the rewrite query reflects the same. The rewrite for the 'other-instructor' group ensures that all the different paths to the 'item' nodes are reconstructed.*

Note how the partially hidden structure for 'item' (without the solution shown for the 'student' group) is reconstructed to return only the accessible elements.

5.4 Soundness and Completeness Properties

We now show that the rewrite algorithm is sound and complete, proving that running an XPath query through the SQR algorithm has exactly the same effect as materializing the view and evaluating the XPath query against the materialized view. Informally, soundness ensures that a node in the original document but not in the materialized view cannot be accessed through SQR. Completeness ensures

that every node in the materialized view can in fact be accessed the same way using SQR.

Before we prove the results, we introduce the notation of a security view $V_{(D,O)}$ on a database D with a known schema induced by a sequence of SSX operations O as the XML tree generated by applying all the operations in O in proper order.

LEMMA 1. **XPath simplification** *Any XPath query involving conditions can be transformed into an XQuery expression involving only simple path expressions (SPE).*

Justification An XPath query of the form $p_1[p_2\ op\ p_3]/p4$, where p_1, p_2, p_3, p_4 are XPath expressions, can be rewritten in XQuery as follows:

```
for $i in p1
where   $i/p2 op $i/p3
return  $i/p4
```

This translation simply follows XQuery semantics [Chamberlin et al. 2001]. This process is used in Rule 1 for ensuring that the other rewrite rules only need to handle simple path expressions (SPE).

LEMMA 2. **Removal of '//'** *Any XPath query involving '//' can be rewritten using only '/' in the presence of a known acyclic schema.*

Justification Since the schema is known and has no cycles, all possible expansions of the XPath query can be computed, and can be combined using XQuery. This property also follows XQuery semantics. Rule 3 in SQR uses this property to simplify the query. Note that this property applies to XPath wildcards as well.

Given the notation for a security view, we can now formally state our soundness property as

THEOREM 1. **Soundness**. *Given a document D conforming to a known schema $\mathcal{S}$, and a sequence of security view operations O inducing a security view $V_{(D,O)}$, any node x in D but not in $V_{(D,O)}$ cannot be accessed through SQR using any XPath query p. Formally,*

$$x \in D, x \notin V_{(D,O)} \Rightarrow \neg\exists p \quad x \in SQR_{S_v}(p)(D)$$

Given the two lemmas, we can now restate the soundness property without loss of generality to involve XPath queries without conditions, // and wildcard. So we can now assume that the XPath query p in the soundness theorem is of the form: $p = t_1/t_2/\cdots/t_k$, where $t_1 \ldots t_k$ are in the vocabulary of the schema $\mathcal{S}$ of D.

Proof:. The soundness property can now be proved by contradiction. Assume that there is an XPath query $p = t_1/t_2/\cdots/t_k$ that can be used to access a node x in D which is not in $V_{(D,O)}$. This implies that x is either a node labeled t_k, or is a descendant of t_k. Given this observation, two cases may arise depending on the highest clean element (HCE) of p with respect to $V_{(D,O)}$:

. **Case 1**: t_k is the HCE(p) or a descendant of HCE(p)
$\Rightarrow t_k$ has no operations defined on it
$\Rightarrow t_k \in V_{(D,O)}$

Hence, if x is labeled t_k, $x \in V_{(D,O)}$, which contradicts our assumption. We can similarly show that if x is a descendant of t_k, $x \in V_{(D,O)}$, which contradicts our assumption.

. **Case 2**: t_k is dirty
$\Rightarrow t_1/t_2 \ldots /t_k$ are all dirty in the SAS for $V_{(D,0)}$

Given this, t_k must either have an annotation or have at least one annotated descendant. Since all nodes with annotations are rewritten by SQR, we examine the different annotations individually:

—**newnode**: The newnode annotation is created by a copy, create, or rename operation. As observed by Rules 8, 9, 10 in the SQR algorithm, in each of these cases, the nodes are created using new information. Hence x, accessed by the path p, is not a node in the original document (*i.e.*, $x \notin D$). This leads to a contradiction to our original assumption $x \in D$.

—**delete**: The delete annotation is assigned using a conditional or unconditional delete operation. We examine both of these cases:

—*Unconditional delete*: Rule 6 demonstrates that when any node with an unconditional delete is processed, SQR returns an empty nodelist. Hence either the target node x or one of its descendants is modified to return an empty nodelist, thereby contradicting the assumption that $\exists p \;\; x \in SQR_{S_v}(p)(D) \;\; for \; x \in D, x \notin V_{(D,O)}$.

—*Conditional delete*: By Rule 7, the only possible way a node marked with a conditional delete is returned is if the node does not match the condition. If x is the node with the annotation, x does not match the delete condition, and hence must be in the view $V_{(D,O)}$. If x is the ancestor of a node with the annotation, x is still part of the view since through rule 7, the conditionally deleted nodes will not be part of the subtree under x. Hence, $x \in V_{(D,O)}$, once again contradicting our assumption.

Thus, we conclude that

$$x \in D, x \notin V_{(D,O)} \Rightarrow \neg\exists p \;\; x \in SQR_{S_v}(p)(D)$$

∎

THEOREM 2. **Completeness**. *Given a document D conforming to a known schema $\mathcal{S}$, and a sequence of security view operations O inducing a security view $V_{(D,O)}$, every node in $V_{(D,O)}$ can be accessed by some XPath query p. Formally,*

$$x \in V_{(D,O)} \Rightarrow \exists p \;\; x \in SQR_{S_v}(p)(D)$$

Proof: We can prove completeness by induction on the length(# of operations) of O.
Base case length(O)=0.

For 0 operations, all nodes in the SAS are clean, so HCE(p) for any path p is the root element. Given Rule 2 in SQR such an XPath query will essentially be handled as is, and so any node in $V_{(D,O)}$ can be accessed using the same XPath expression used to access it on D.
Hypothesis.

Assume that for length(O) between 0 and $m-1$, the completeness property holds. In the induction step, we add one more operation and show how the completeness property still holds.

Induction step length(O)=m.

According to the hypothesis, the completeness holds for $O_1 = op_1; O_2 = op_1, op_2;\ldots$ and $O_{m-1} = op_1, op_2 \ldots op_{m-1}$. Now, to prove completeness for an SSX sequence O of length m we examine the m^{th} operator as follows:

—**create(destSPE, newName)**: Three cases arise depending on the location of x relative to $newName$:

. *x is newName*:

By the definition of the **create** primitive the parent node of x can be accessed by $p = destSPE$ under O_{m-1}. Based on Rule 9 $p^1 = destSPE/newName$ can be used to access x under O_m.

. *x is outside the subtree rooted at newName*: By induction hypothesis, there must exist an XPath expression $p = p_1/p_2/\ldots/p_i/\ldots/p_j$ that can be used to access x under O_{m-1}. Given that x is outside the subtree rooted at $newName$, x must be either within $destSPE$, or in an area of the tree not affected by the **create** operation. Hence the XPath expression $p = p_1/p_2/\ldots/p_i/\ldots/p_j$ can still be used to access x under O_m.

. *x is a descendant of newName*: A descendant of $newName$ must be created through one more operation (likely a **copy** or a **create** operation). Since the **create** is the the m^{th} operation,this case will not arise.

—**delete(destXPath)**: Let t_{eval}^{dest} represent the result tree obtained by the evaluation of $destXPath$ in $V_{(D,O_{m-1})}$. Two cases arise depending on the location of x relative to $destXPath$:

. *x is outside the subtree(s) represented by* t_{eval}^{dest}: By induction hypothesis, there must exist an XPath expression $p = p_1/p_2/\ldots/p_i/\ldots/p_j$ that can be used to access x under O_{m-1}. Given that x is outside the subtree(s)represented by t_{eval}^{dest} that is affected by the **delete** operation, the XPath expression p=$p_1/p_2/\ldots/p_i/\ldots/p_j$ can still be used to access x under O_m.

. *x is at or below a node rooted by* t_{eval}^{dest}: By induction hypothesis there exists an XPath expression $p = destXPath$ that can be used to access the root of t_{eval}^{dest} under O_{m-1}. If x is at or below a node that is unconditionally deleted (or part of the nodes conditionally deleted), then this case is not applicable as x will not be a part of $V_{(D,O)}$. On the other hand if x is at or below a node that has been conditionally deleted,$x \in V_{(D,O)} \Rightarrow x$ does not satisfy the condition, then as per Rule 7 of SQR all nodes in the complement of the condition including x can be returned by the XPath expression $p = destXPath$ under O_m.

—**rename(destSPE, newName)**: Three cases arise depending on the location of x relative to $newName$:

. *x is newName*: By induction hypothesis, the node to be renamed can be accessed by $p = destSPE$ under O_{m-1}. Assuming the tail tag of $p = destSPE$ is y, we have $p = destSPE = p1/y$. Based on Rule 10 the XPath expression $p^1 = p1/newName$ can be used to access x under O_m.

. *x is outside the subtree rooted at newName*: By induction hypothesis, there must exist an XPath expression p that can be used to access x under O_{m-1}.

Given that x is outside the subtree rooted at $newName$, x must be either within $destSPE$, or in an area of the tree not affected by the **rename** operation. Hence the same XPath expression p can still be used to access x under O_m.
. *x is a descendant of newName*: By induction hypothesis, there must exist an XPath expression $p = p_1/p_2/\ldots/p_i/\ldots/p_j$ (where p_i is the tag that is replaced by $newName$)that can be used to access x under O_{m-1}. Since **rename** affects only the node renamed, $p^1 = p_1/p_2/\ldots/newName/\ldots/p_j$ can be used to access x under O_m.

—**copy(sourceXPath, destSPE, [newName], [scope], [preserve])**: Let t^{dest}_{eval} represent the result tree obtained by the evaluation of $destSPE$. Lets consider the various cases depending on the location of x.
. *x is outside the subtree(s) represented by* t^{dest}_{eval}: By induction hypothesis, there must exist an XPath expression p that can be used to access x under O_{m-1}. Since **copy** is the m^{th} operation and since x is not affected by it, the XPath expression p can still be used to access x under O_m.
. *x is in the subtree which is the result of the copy operation*: By induction hypothesis there exists an XPath expression $p = destSPE$ that can be used to access the root of t^{dest}_{eval} under O_{m-1}. Since x is part of the subtree(rooted with the tag $oldName$) that has been copied, it will be retrieved from the original subtree by Rule 8. Since **copy** is the m^{th} operation,these nodes are not affected by a **delete** operation following the **copy** operation. Hence the XPath expression $p^1 = destSPE/oldName$ can be used to access x under O_m.
. *x is newName*: By induction hypothesis there exists an XPath expression $p = destSPE$ under O_{m-1} that can be used to access the new parent of x. Now based on Rule 8 the XPath expression $p^1 = destSPE/newName$ can be used to access x under O_m.

The proof follows via total induction on the length of O. ∎

6. EXPERIMENTAL EVALUATION

To demonstrate the effectiveness and efficiency of the techniques proposed in this paper, we conducted evaluations of ACXESS using datasets generated by publicly available XML benchmarks. We compared our security query rewrite algorithm SQR against an alternative enforcement technique viz 'Materialized Views' and a previously available method for XML access control [Fan et al. 2004]. Our experimental results clearly revealed that our approach was superior in terms of effectiveness and performance when compared to materialized views. In the presence of uncertainty (wild card '$\star$' and '//') and deeply nested paths the performance improvement is substantial. In other cases our approach exhibited better or comparable performance to that of materialized views. We also compared our approach against the most recent access control technique for XML based on query reformulation by Fan et al. [Fan et al. 2004]. Our approach proved to be superior providing better functionality and better or comparable performance on common functionality.

Experimental Setup Our experiments were conducted on a 2.8 GHz Intel Pentium IV, 512 MB Linux desktop. The times reported in this section were obtained

as an average over five runs[7] using the Galax [J.Simeon and M.Fernandez] XQuery engine.

DataSet The data sets reported in this section model a catalog of books generated using the XBench benchmark [Yao et al. 2002]. Datasets produced by the XBench test suite were modified to generate three different document sizes - 10, 50 and 75 MB.

Security Models We created seven different security models on the XBench Catalog Schema [XBench](See Appendix C.1 for a visual representation). Models were generated to evaluate the effect of each individual primitive in SSX on the SAS construction and access constraint enforcement phase. Models were also created to test the cumulative effect of all the primitives. Accordingly four models were created to test a specific primitive thoroughly while three models implemented a good and viable mixture of the four primitives. Appendix C.2 provides detailed information about the SSX sequences used to generate the seven security models.

Query Set We summarize the queries we tested with into five classes, to facilitate the discussion and analysis of the results. The main aim in deciding the query classes was to check the scenarios that are directly affected by the presence of annotations in the SAS. Accordingly we generated queries to retrieve the nodes that were modified by a SSX sequence(entire document,close to the root, close to the leaf nodes). The presence of a '//' increases the complexity of evaluation of an XPath query. Hence query classes were generated to test queries with uncertainty. The description of each query class and a few example queries in each class are shown in Table IV.

6.1 Comparison against Materialized Views

Security View Generation For each of the seven security models, an annotated schema (SAS virtual view (vv)) was constructed using our framework. For comparison, a materialized view (mv) was also constructed, using a specially written XSLT script. Our experiments showed that the time taken for the SAS virtual view generation process was dependent only on the length of the SSX sequence while the time for view materialization grew linearly with the size of the data. The time taken to generate the SAS and the materialized views for the 10, 50 and 75MB files are shown in Figure 5.

Access Control Enforcement To study the impact of our framework in access control enforcement, on the XBench dataset, we ran a set of queries in 105 different environments (7 security models $\times$ 5 query classes $\times$ 3 data sizes). The time taken to execute the 7 sample queries (as shown in Table IV) on the 50 MB XBench document based on Security Model No.5 (as shown in Appendix C.2) is shown in Figure 6. In our approach the query answering time is the sum of the rewrite time and the execution time of the rewritten query.

Based on the comparison against materialized views we can draw the following conclusions:

[7]Each query was run five times and the best and the worst time were not considered and the time taken to run the query was evaluated as the average of the remaining three times.

Query Type	Description	Sample Query
T_1	Queries with uncertainty - The presence of a // or a wild card character	q3:/catalog/item//contact_information
T_2	Retrieves nodes modified by a SSX sequence with deep paths (length $\geq$ 6)	q1:/catalog/item/publisher/contact_information/mailing_address/state q6:/catalog/item/authors/author/contact_information/mailing_address/state
T_3	Retrieves modified nodes with short paths(length $\geq$ 3)	q2:/catalog/item/attributes q4:/catalog/item/contacts
T_4	Retrieves entire document	q5:/catalog
T_5	Retrieves unmodified nodes	q7:/catalog/item/subject

Table IV. Test Query Characteristics for Comparison against Materialized Views

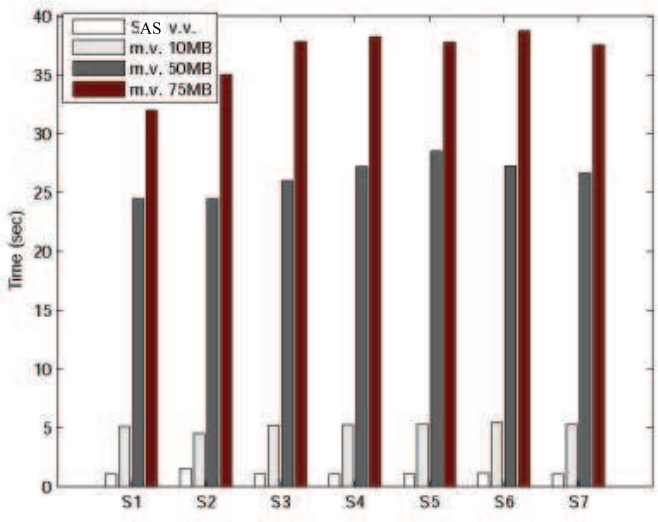


Fig. 5. Materialized View vs. SAS Generation Times

—For Type 1 queries the SQR approach is much faster than the materialized views approach, because the reachability of each node is computed before hand and stored in the SAS.

—For Type 2 queries the SQR mechanism exhibits better or comparable performance over that of the materialized views approach. The performance improve-

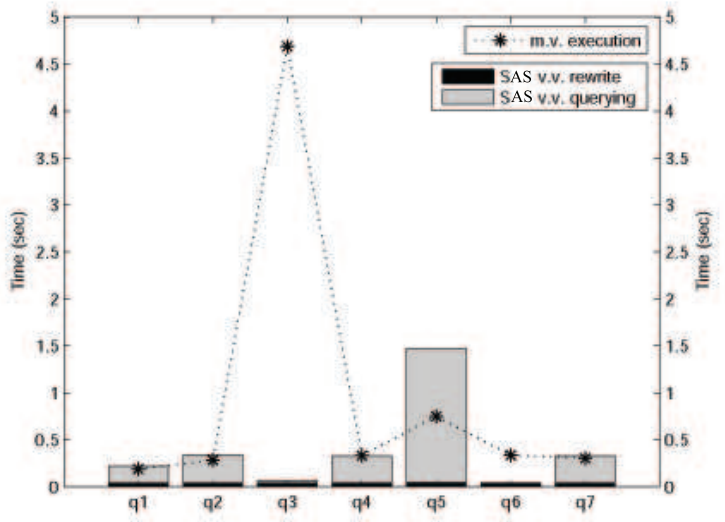


Fig. 6. Materialized View Approach vs. SQR Query Answering Approach

ment is mainly due to the level of nesting and the efficiency introduced by the rewrite rules.

—For Type 3 and 5 queries, the two approaches have comparable performance. Type 2 queries are really small while Type 5 queries return unmodified nodes. Hence the overhead due to the rewrite is negligible resulting in comparable performance.

—For Type 4 queries, the materialized view approach tends to perform better. This is because the return data has to be re-constructed (if nodes have been modified) in case of the SQR method .

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 and multiple security levels on the same document makes our approach more attractive when compared to the materialized view approach.

6.2 Comparison against the Fan Approach

One of the most recent works on XML access control was done by Fan et al.[Fan et al. 2004]. The techniques proposed in [Fan et al. 2004] provided limited functionality that is comparable to the "delete" primitive in SSX and hence is comparable to only one of the seven tested security models. Our experiments here focused on the performance for the specific security model that was supported in [Fan et al. 2004].Table V provides the profile information about the queries that were used in the comparison.

Figure 7 presents the query answering time of five sample queries (distinct from

Query Type	Description	Sample Query
T_1	Queries with uncertainty - The presence of a // or a wild card character	p1:/catalog//contact_information
T_2	Retrieves nodes modified by a SSX sequence with deep paths (length $\geq$ 6)	p2:/catalog/item/authors/author/contact_information/mailing_address/state
T_3	Retrieves modified nodes with short paths(length $\geq$ 3)	p3:/catalog/item/attributes
T_4	Retrieves entire document	p5:/catalog
T_5	Retrieves unmodified nodes	p4:/catalog/item/subject

Table V. Test Query Characteristics for Comparison against Fan et.al

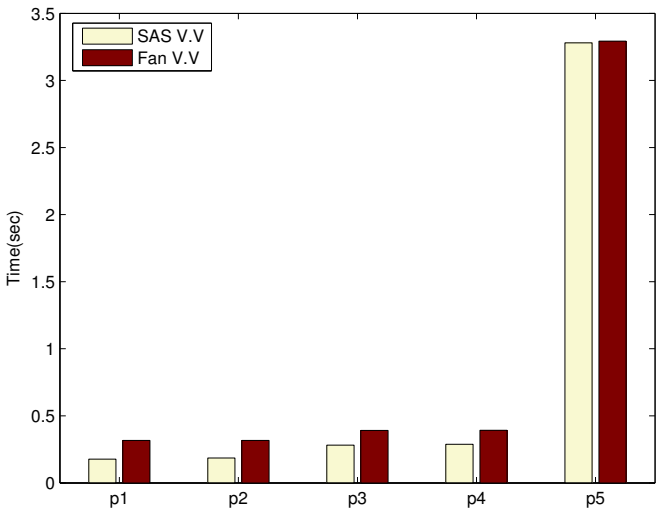


Fig. 7. SQR Approach vs. Fan et al. Approach

the test set used in the section above) for both the techniques.[8] The results indicate that our approach provides enhanced functionality with better or comparable

[8]The limited functionality offered by Fan et.al technique demands that we use a different security model. Hence the queries that are reported are different from the ones used to compare against the materialized views approach.

performance to the technique implemented by Fan et al.[Fan et al. 2004]. Complete details of the timing results can be obtained from [Mohan et al.].

7. CONCLUSION AND FUTURE WORK

With the growing popularity of XML and XML database systems, role-based access control is critical in XML data management. Comparing XML to the relational world, the challenge in XML access control lies in the semi-structured nature of XML documents and tree pattern matching nature of XML queries. In an XML context, not only can values of elements/attributes be sensitive, but also the structural relationship among elements/attributes. We propose a framework for specifying and enforcing extended access constraints on XML documents, comprising of SSX, an XML security view specification language for DBAs to specify the access constraints; SAS, an annotated schema which acts as the internal representation of the access constraints and the foundation for security constraint enforcement; and SQR, a rule-based algorithm that enforces the access constraints on query evaluation via query rewrite, without the cost of materializing and maintaining the security views. Our experiments demonstrate the effectiveness and efficiency of our approach.

The techniques proposed in this paper assume that every XML database has an acyclic schema. We are looking forward to extending our work to handle recursions in 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 may be beneficial. 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. We are also working on a declarative language for defining access control for XML. The expressive power of the declarative language will be equivalent to SSX, but it will enable the DBAs to just specify the constraints without specifying when and how the access constraint will be implemented or enforced. Furthermore, there will not be a logical dependency between the supported primitives. Yet another direction of research is anti-inference, the study of information leakage and techniques to prevent their occurrence.

8. ACKNOWLEDGMENTS

We thank the members of the Database Lab at Indiana University for their valuable comments in improving the content and the style of this paper. We also thank Jonathan Klinginsmith for helping us in testing the system and for discussions to refine the rewrite rules.

REFERENCES

Atzeni, P. and Mecca, G. 1997. Cut & paste. In *ACM SIGACT-SIGMOD-SIGART*. ACM Press.

Bertino, E. 1992. Data hiding and security in object-oriented databases. In *International Conference on Data Engineering*.

Bertino, E. and Ferrari, E. 2002. Secure and selective dissemination of XML documents. *ACM Trans. Inf. Syst. Secur. 5,* 3.

Bertino, E., Samarati, P., and Jajodia, S. 1993. Authorizations in relational database management systems. In *ACM Computer and Communications Security*.

Chamberlin, D., Clark, J., Florescu, D., Robie, J., Simeon, J., and Stefanescu, M. 2001. XQuery 1.0: An XML query language. Available at http://www.w3.org/TR/xquery.

Cho, S., Yahia, S., Lakshmanan, L., and Srivastava, D. 2002. Optimizing the secure evaluation of twig queries. In *International Conference on Very Large Databases*.

Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P. 2000. Securing XML documents. In *International Conference on Extending Database Technology*.

Faatz, D. B. and Spooner, D. L. 1990. Discretionary access control in object-oriented engineering database systems. In *Database Security*. 73–84.

Fan, W., Chan, C.-Y., and Garofalakis, M. 2004. Secure XML querying with security views. In *ACM Special Interest Group on the Management of Data(SIGMOD)*.

Griffiths, P. P. and Wade, B. W. 1976. An authorization mechanism for a relational database system. *in ACM Transactions on Database Systems*.

Hada, S. and Kudo, M. XML access control language: Provisional authorization for XML documents, Available at http://www.trl.ibm.com/projects/xml/xacl/xacl-spc.html.

IMS Global Learning Consortium. IMS question and test interoperability information model, version 2.0 final specification. Available at http://www.imsglobal.org.

Jagadish, H. V., Lakshmanan, L. V. S., Srivastava, D., and Thompson, K. 2001. TAX: A tree algebra for XML. In *Database Programming Languages*. 149–164.

Jajodia, S. and Kogan, B. 1990. Integrating an object-oriented data model with multilevel security. In *IEEE Symposium on Security and Privacy*. 76–85.

J.Simeon and M.Fernandez. The XQuery implementation available at http://www.galaxquery.org.

Miklau, G. and Suciu, D. 2003. Controlling access to published data using cryptography. In *International Conference on Very Large Databases*.

Mohan, S., Sengupta, A., and Wu, Y. 2005. Access control for XML - a dynamic query rewriting approach. In *ACM Conference on Information and Knowledge Management*.

Mohan, S., Sengupta, A., Wu, Y., and Klinginsmith, J. XML access control, available at http://www.cs.indiana.edu/∼acxess/.

Murata, M., Tozawa, A., Kudo, M., and Hada, S. 2003. XML access control using static analysis. In *ACM conference on Computer and communications security*.

OASIS. Project available at http://www.oasis-open.org/committees/xcaml.

Rabitti, F., Bertino, E., Kim, W., and Woelk, D. 1991. A model of authorization for next-generation database systems. *ACM Trans. Database Syst.*.

Rabitti, F., Woelk, D., and Kim, W. 1988. A model of authorization for object-oriented and semantic databases. In *International Conference on Extending Database Technology(EDBT)*.

Thuraisingham, B. M. 1989. Mandatory security in object-oriented database systems. In *OOPSLA*.

XBench. The catalog benchmark schema data set generator available at http://db.uwaterloo.ca/ ddbms/projects/xbench/schemas.html.

Yao, B. B., Ozsu, M. T., and Keenleyside, J. 2002. XBench - a family of benchmarks for XML DBMSs. In *Efficiency and Effectivness of XML Tools and Techniques*.

XML File

```
<operand>
 <delete source="/quiz/Access"></delete>
 <delete source="/quiz/Assessment"></delete>
 <create source="/quiz" newNode="solutions"></create>
 <copy source="/quiz/objectbank//item" scope="/quiz"
  destination="/quiz/questionbank",preserve="false"></copy>
 <delete source="/quiz/objectbank"></delete>
</operand>
```

Table VI. Input XML file representing the SSX sequence used to generate "other-instructor" security view

A. SECURITY ANNOTATED SCHEMA DERIVATION ALGORITHM

The SAS construction 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 SAS construction algorithm is presented in Figure 8. Associated helper functions are presented in Figure 9.The SAS construction algorithm has been implemented using Java and JAXP. All annotations are effected on the nodes that are modified and are represented by introducing a new child with the name space "view:annotation". The specific details of the annotation and the type of annotation and any other desired arguments are stored as attributes of the "view:annotation" element. Dirty and Scope annotations are indicated by introducing a new child with the name space "view:dirty" and "view:scope" respectively.

The sequence of SSX operations are provided to the algorithm via an XML file. Each and every element in the XML file represents an operation in the SSX sequence, with the attributes of the element representing arguments of the SSX operation. A sample XML file representing an SSX sequence is shown in Table VI. The algorithm parses the XML file one element at a time and iterates through the sequence of operations thus obtained,using the output schema from the previous operation as input. For each operation, 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 operation's chronological sequence # (S#).

A.1 SAS derivation Algorithm - Helper Functions

This section describes briefly the various helper functions that are needed for the successful construction of the SAS including:

—**XPath Translation:** As pointed out in Section 4, the node of interest is specified by either an XPath or a Simple Path Expression. As can be seen from Table VI the XPath expression used to identify the nodes is specified on the XML document while it has to be evaluated against the input XML schema. The XML schema introduces additional tags to identify the properties of each and every XML element and this necessitates that the input XPath expression be automatically translated to handle XML schema.
This translation is carried out by the "*translateExpression*" function which parses the input XPath expression into tokens. Each token is analyzed to determine

whether it is an attribute, a simple element, a complex element or a root element and is then translated accordingly into an XPath expression that can be evaluated against the XML Schema.

—**XPath Evaluation:** The XPath expression is automatically evaluated by using the predefined XPath Factory library in JAXP. The evaluation is carried out by the *evalxpath*(inputschema,p1) function, which evaluates the XPath expression *p1* on the *inputschema* and returns the list of all the nodes that match *p1*.

—**Add Annotation:** This helper function is used to add a "view:annotation" element to the node that is being modified. This is carried out by using the *addAnnotation*(...) function that sets the annotation attributes depending on the type of modification that is being carried out on the node. For example in case of a **delete** operation, attributes indicate if the delete was conditional and if conditional, the condition that was used to delete the node. Similarly in case of a 'NewNode' annotation, a flag is set, indicating the actual operation that was performed and any other attributes that need to be stored. The function also calls the "set dirty" helper function to ensure that the node and all its ancestors are marked as dirty. In case of a scoped copy the "set scope" helper function is also called.

—**Set Dirty:** A dirty node indicates that either the node or one of its descendants has been modified by an SSX operation. This is indicated on the schema by using the dirty attribute of the "view:dirty" element. This is carried out by the *setdirty*(node) function which recursively attaches dirty stamps ("view:dirty" elements) at the *node* and all its ancestors. The implementation of the function has been optimized to ensure that recursive trace stops if an ancestor node 'n' has already been marked dirty as all the ancestors of 'n' would have already been marked dirty.

—**Set Scope:** A node annotated with the scope flag indicates that the node has been used as a scope argument for the **copy** operation. This is indicated on the schema by using the scope attribute of the "view:scope" element. This is carried out by the *setscope*(node) function that adds a "view:scope" element for the *node*.

—**Nested Copy** The copy primitive entails that the node to be copied is copied along with all its descendant nodes. This is carried out by using the *copydom*(node) function which makes a duplicate copy of the entire DOM tree of the schema rooted at *node* and returns the root node of the duplicate version.

—**Chronological Sequence Number Generator:** Due to the dependency between various operations, its important that the annotation reflect the chronological sequence of changes that were carried out on the schema. This is done by ensuring that each and every operation in the sequence is associated with a number directly related to their appearance in the SSX sequence. When the input SSX XML file is parsed and a valid operation obtained, a unique number is obtained from an increasing monotonic series, and used as the chronological sequence number.

—**Check Primitive Validity:** Every SSX operation obtained from the input SSX XML file is passed to this function to ensure that the operations are valid under the SSX restrictions and that the arguments to the function are also valid.

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

Fig. 8. SAS Construction Algorithm

B. SECURITY ANNOTATED SCHEMA CONSTRUCTION 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 8. Associated helper functions are presented in Figure 9. A detailed description is available in Appendix

evalxpath(outputschema,p1) evaluates the XPath expression *p1* on the *outputschema* and returns the list of all the nodes that match *p1*;
evalxpath_scope(outputschema,p1,scope) evaluates the XPath expression *p1* on the *outputschema* and returns the list of all the nodes that match *p1* with respect to the *scope*;
setdirty(node) recursively attaches dirty stamps to the *node* and all it's 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.

Fig. 9. Helper Functions for the SAS Construction Algorithm

A. The annotation algorithm has been implemented using Java and JAXP. All annotations are effected on the node that is being modified.

Each operation uses the output schema from the previous operation as input. 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. The rest of the section describes briefly the changes effected on the input schema as a result of a particular SSX primitive.Each of these functions make use of the helper functions to perform the desired changes. Note that we use XML Schema as the schema language for discussion below.

—**Create New Node:** The create operation creates a new node using the argument provided by the DBA. Since the changes are being effected on the schema this function creates a new node of the type 'xs:element' and registers the newly created element with the correct name space. The function should also effect changes if the parent element is a leaf node. The leaf node has to be converted to a complex type element in the XML Schema. The function also calls the appropriate helper functions to ensure that annotations are added to reflect the newly created node.

—**Rename Node:** The rename operation is the most trivial of all the tree modification primitives. It is carried out by changing the attribute 'name' of the element to the new name. The function also calls the appropriate helper functions to ensure that annotations are added to reflect the rename operation.

—**Copy Node:** The copy operation relies on the "copydom" helper function to obtain a deep copy (including the descendants) of the nodes being copied. The duplicate nodes thus obtained are added as children at the destination, ensuring that if the destination node is a leaf element it's converted into an complex type element as mentioned in the create new node function.
If multiple nodes are being copied(the result of a // at the source) the copy function has to ensure that all the nodes are copied to the destination and the annotation reflects accurately the source from which each node was obtained. If the copy is being performed conditionally, the condition is typically ignored during the SAS construction, but is remembered using the annotations for enforcement at a later stage.

—**Delete Node:** The delete operation does not make any changes to the existing tree structure. It just adds new annotations to reflect the delete operation. Just like the conditional copy, if the delete is being performed conditionally, the condition is typically ignored during the SAS construction, but is remembered using the annotations for enforcement at a later stage.

C. EXPERIMENTAL EVALUATION DETAILS

To demonstrate the effectiveness and efficiency of the techniques proposed in this paper, we conducted evaluations of ACXESS using datasets generated by publicly available XML benchmarks. We compared our security query rewrite algorithm SQR against an alternative enforcement technique viz 'Materialized Views' and a previously available method for XML access control [Fan et al. 2004]. Our experimental results clearly revealed that our approach was superior in terms of effectiveness and performance when compared to materialized views. In the presence of uncertainty (wild card '$\star$' and '//') and deeply nested paths the performance improvement is substantial. In other cases our approach exhibited better or comparable performance to that of materialized views. We also compared our approach against the most recent access control technique for XML based on query reformulation by Fan et al. [Fan et al. 2004]. Our approach proved to be superior providing better functionality and better or comparable performance on common functionality.

C.1 Test Schema

Figure 10 shows the original Schema of the XBench benchmark dataset.

C.2 Test SSX Sequences

Table VII shows the SSX sequences that was used to derive the seven security models used in the experimental evaluation.

Fig. 10. The Catalog Schema

Model	Security View Definition
Security Model 1	delete(/catalog/item/authors/author/contact_information) delete(/catalog/item/publisher/contact_information) delete(/catalog/item/pricing/cost) delete(/catalog/item/authors/author/date_of_birth) delete(/catalog/item/attributes)
Security Model 2	rename(/catalog/item/publisher/contact_information/ mailing_address/name_of_city, city) rename(/catalog/item/publisher/contact_information/ mailing_address/name_of_state, state) rename(/catalog/item/authors/author/contact_information/ mailing_address/name_of_city, city) rename(/catalog/item/authors/author/contact_information/ mailing_address/name_of_state, state) rename(/catalog/item/subject,person) rename(/catalog/item, catalogitem)
Security Model 3	copy(/catalog/item/publisher/contact_information,/catalog, contact_publisher,/,true) copy(/catalog/item/authors/author/contact_information,/catalog, contact_author,/,true) copy(/catalog/item/publisher/contact_information,/catalog/item, all_publishers,/catalog/publisher/item,true) copy(/catalog/item/authors/author/contact_information, /catalog/item,all_authors,/catalog/authors/author/item,true)
Security Model 4	create(/catalog,contacts) create(/catalog/contacts,author_contact) create(/catalog/contacts,publisher_contact) create(/catalog/item/attributes,bookType) create(/catalog/item//contact_information,cont)
Security Model 5	rename(//contact_information/name_of_state,state) delete(//contact_information/country/exchange_rate) delete(//contact_information/country/currency) copy(/catalog/item/publisher/contact_information,/catalog/item, contacts,/,true) delete(/catalog/item/attributes/number_of_pages) rename(/catalog/item/attributes/book_type,booktype)
Security Model 6	create(/catalog, contacts) copy(/catalog/item/authors/author/contact_information, /catalog/contacts, contact_information, /,true) copy(/catalog/item/authors/publisher/contact_information, /catalog/contacts, contact_information, /,true) delete(/catalog/item/authors) delete(/catalog/item/publisher)
Security Model 7	copy(/catalog/item/title, /catalog/item/attributes, book_title, /,true) delete(/catalog/item/title) copy(/catalog/item/date_of_release, /catalog/item/attributes, release_date, /,truee) delete(/catalog/item/date_of_release) copy(/catalog/item/subject, /catalog/item/attributes, book_subject, /,true) delete(/catalog/item/subject) copy(/catalog/item/description, /catalog/item/attributes, desc, /,true) delete(/catalog/item/description) rename(/catalog/item, catalog_item)

Table VII. SSX Sequences used to Define the Security Views for the Seven Security Models used in the Experimental Evaluation