Tree Automata for Schema-level Filtering of XML Associations

Vaibhav Gowadia and Csilla Farkas

Information Security Laboratory
and
Department of Computer Science and Engineering
University of South Carolina, Columbia, SC 29208

Abstract. In this paper we present query filtering techniques based on bottom-up tree automata for XML access control. In our authorization model (RXACL), RDF statements are used to represent security objects and to express the security policy. Our model allows to express and enforce access control on XML trees and their associations. We propose a query-filtering technique that evaluate XML queries to detect disclosure of association-level security objects. A query Q discloses a security object o iff the (tree) automata corresponding to o accepts Q. We show that our schema-level method detects all possible disclosures, i.e., it is complete.

1 Introduction

Several XML access control models have been developed recently [1–6]. They allow node-level security granularity by assigning access restrictions to the nodes and links of XML documents. However, none of these models provide access control for data *associations*. Intuitively, an association security object is an XML subtree that is disallowed to be accessed by a user, while all of its proper subtrees are permitted separately. Incorporating association in an access control model increases data availability while preserving confidentiality.

The RXACL architecture, introduced in [7], provides flexible access control granularity by allowing security classification of XML nodes and subtrees (simple security objects), and associations among nodes (association security objects). In [7] we proposed a technique to enforce association-based access control at data-level (i.e., check for security violation after query processing) and it is outside the scope of this paper. In this paper we extend RXACL architecture by presenting techniques for performing security check before the query is processed. Our work is similar to those proposed by Murata et al. [5] and Luo et al. [6]. However, their method supports node-level security objects only. The automata model, used by them is not sufficient to model association-level security objects. In this paper, we use bottom-up tree automata to represent security objects.

[0] This work was partially supported by National Science Foundation grant number IIS-0237782.

Gowadia V. and Farkas C. (2005).
Tree Automata for Schema-level Filtering of XML Associations.
In *Proceedings of the 3rd International Workshop on Security in Information Systems*, pages 136-145
DOI: 10.5220/0002575401360145

We propose a data-independent technique to recognize disclosure of association level security objects by XML queries. Results of our analysis can be (1) association objects are disclosed, (2) association objects are not disclosed, or (3) association objects may be disclosed. Options 1 and 2 indicate that the query should be rejected or accepted, respectively. If the third option is reached, data-level analysis is required to evaluate whether a security violation is present or not.

We present a two-layered association filtering method. First we detect disclosure of association in a given query-pattern, i.e., information encoded in the XML query itself. Second, we *extend* query-pattern with document schema to represent all schema information that the query answer would reveal to a user. XML query-patterns are labeled-trees where node labels may be variables, constants, or the special symbol '//'(self-or-descendant axis [8]). We model association security objects with *pattern automatas* (Definition 8). A pattern automata takes (extended) query-patterns as input and reaches an accepting state if and only if the input discloses the security object represented by it. The main technical contributions of this paper are the development of pattern automatas for security objects and the notion of extended query-pattern. We present algorithms to construct query-pattern, pattern automata, and to detect disclosure of security objects.

The organization of the paper is as follows: next section presents an overview of RXACL architecture and query filtering mechanism. Section 3 introduces formal definitions of basic constructs used in this paper. Section 4 presents algorithms for constructing query-pattern, association pattern automata and to detect association disclosure. Section 5 introduces the notion of extended query-pattern and presents a schema-level security analysis of query. We conclude in section 6 and list future work.

2 RDF-based XML Access Control Architecture

Figure 1 shows the RXACL architecture. The architecture contains four main components: 1. Query filter 2. Query engine, 3. Data level access control, and 4. User history. The query filtering component performs schema-level analysis to determine whether answer to the input query : (1) violates access control policy (*violating*), (2) does not violate the access control policy (*safe*), or (3) requires a data-level security check to detect possible violations (*unsafe*). The XML query engine is responsible for generating responses to user's requests. RXACL uses an existing XML query engine, the development of such an engine is outside of the scope of this paper. The data-level access control component analyzes the query-answer based on the security policy and data previously released to the user [7]. The history component keeps track of answered query-patterns and data released to each user.[1]

When a data request is submitted to a RXACL system, query filtering component first checks for disclosure of disallowed association-level security objects in the query (without utilizing the XML schema information). If a disallowed association-level security object is disclosed, the query is immediately rejected. Otherwise, the query-pattern is extended with schema information and query-patterns of previously answered queries to the user. Extended query-pattern are now checked for disallowed objects. If no disallowed association-level security object is disclosed in extended query-pattern the query

[1] Due to space limitation the handling of the history file is not presented in this manuscript.

answer is labeled *safe*. Otherwise, it is labeled *unsafe*. The query is submitted to XML query engine for further processing. The result of unsafe query's evaluation are further evaluated for possible data-level violations as described in [7]. Answer to safe queries is returned to user without further security analysis. For all queries that are answered, user history is updated with query-pattern and released data. The assurance of our query level filtering is based on the completeness property of the filtering.

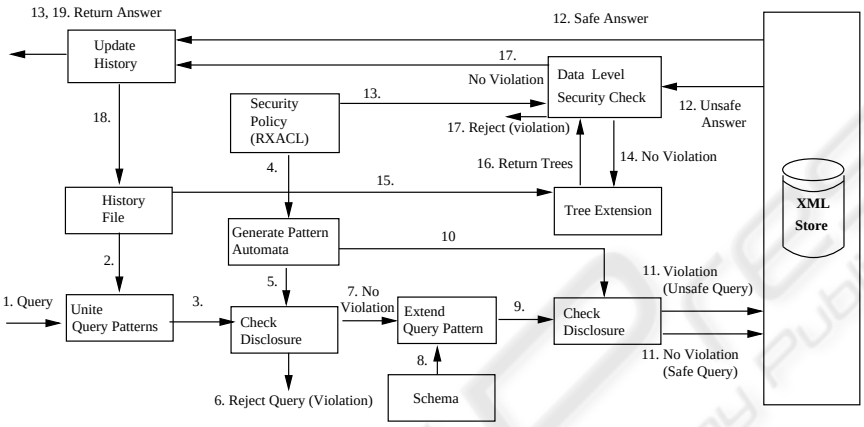


Fig. 1. RXACL architecture for enforcing XML access control

3 Definitions

This section describes definitions necessary to model XML queries , association objects, and XML schema.

Definition 1. (Labeled-tree)
A *labeled-tree*, or a tree, is defined recursively as follows: (1) the empty set $\{\}$ is a tree, called the empty tree, (2) a single *node* $\{n\}$ is a tree, and (3) if $t_1, t_2, \ldots, t_k$ are trees, then $\{n \rightarrow \{t_1, t_2, \ldots, t_k\}\}$ is a tree. In this case we say that $\{n \rightarrow \{t_1, t_2, \ldots, t_k\}\}$ represents the tree whose root n has outgoing edges to subtrees $t_1, t_2, \ldots, t_k$.

The nodes of the trees are labeled. Labels may be constants, node variables (corresponding to any node value), or path variables (corresponding to any path). Constants correspond to element, attribute and text values. Nodes labeled with text-values are called text nodes and are always leaf nodes. Attribute nodes can have only on child node, a text node. Also, any two attribute nodes of a given element cannot have same label. Element nodes can have zero or more child nodes that can be elements, attributes, or text nodes. We denote element nodes with n_i, attribute nodes with a_i, and text nodes with *pcdata*.

A labeled tree is called a *ground tree* if all of its nodes are labeled with constants. An XML instance T is a ground-tree.

Definition 2. (Path-expression)
Let $p = \{n, \{a_1, \ldots, a_j\}\}$ represent a single node n and its child nodes corresponding to attributes $a_1, \ldots, a_j$, where n is either a constant, or a variable. A *path-expression* is defined as: 1. p is a path expression, 2. $\{p_1 \rightarrow p_2 \rightarrow \ldots \rightarrow p_k\}$ is a path-expression where $p_i (i = 1, \ldots, k)$ are path-expressions, 3. Let // denote an arbitrary path-expression. Then the following are also path-expressions: $\{// \rightarrow p_1 \rightarrow \ldots \rightarrow p_m\}$, $\{p_1 \rightarrow // \rightarrow p_m\}$.

DTD [9] and W3C XML Schema [10] satisfy a constraint that all child nodes of any given node must have unique names. Trees satisfying this constraint are called *single-type* tree [11]. Path-expressions are single-type trees.
We consider XQuery syntax [12] of the following form:

Definition 3. (XML Query)
An XML query Q is of the following form:
FOR v_0 in P_0
LET $v_1 := P_1, \ldots, v_l := P_l$
RETURN $\{n \rightarrow \{\overline{v_k}, \ldots, \overline{v_j}\}\}$
WHERE ($\overline{v_i} == \overline{v_j}$ and $\ldots$ and $\overline{v_l} == \overline{v_m}$)
where, v_i $(i = 0, \ldots, l)$ are variables of query (we refer to them as *query-variables* in rest of this paper), $\overline{v_i}$ $(i = 0, \ldots, m)$ represent a path-expression $\{v_i \rightarrow p'\}$ $(i = 1, \ldots, l)$ and p' is a path-expression that does not contain any query-variables, P_i $(i = 0, \ldots, l)$ are path-expressions, and n is a constant.

Given a XML query Q, the first step in query filtering architecture is to build query-pattern of Q. Let $V = \{v_1, v_2, \ldots, v_l\}$ be the set of query-variables defined in Q, and $\overline{V} = \{\overline{v_1}, \overline{v_2}, \ldots, \overline{v_m}\}$ be the path-expressions in the RETURN or WHERE clause of the query. Intuitively, the query-pattern is constructed by *uniting* the path-expressions in $\overline{V}$. Since path-expressions may contain query-variables. We need a method to eliminate query variables. A formal definition of variable-substitution follows.

Definition 4. (Variable-Substitution)
Let $\$v_i = \{p_1 \rightarrow \ldots \rightarrow p_l\}$, and $\$v_j = \{\$v_i \rightarrow p'_1 \rightarrow \ldots p'_m\}$ be two assignments in the FOR or LET clause of the XML query. A *variable substitution* replaces $\$v_i$ in the second assignment with $\{p_1 \rightarrow \ldots \rightarrow p_l\}$.

Example 1. Consider the single-type tree $T = \{\$x \rightarrow \{a, d\}\}$, *where* $\$x = \{// \rightarrow \{r\}\}$ *is a query-variable.* Substituting $\$x$, *we get* $T = \{// \rightarrow \{r \rightarrow \{a, d\}\}\}$. □

Definition 5. (Single-type Tree-Merge)
Let $P_1 = \{n_1^1 \rightarrow n_2^1 \rightarrow \ldots n_k^1 \rightarrow n_{k+1}^1 \rightarrow \ldots n_l^1\}$ and $P_2 = \{n_1^2 \rightarrow n_2^2 \rightarrow \ldots n_k^2 \rightarrow n_{k+1}^2 \rightarrow \ldots n_m^2\}$ be two ground path-expressions over the same schema. We define *merge* of path expressions as follows:
if $n_1^1 = n_1^2, n_2^1 = n_2^2, \ldots, n_k^1 = n_k^2$, and $n_{k+1}^1 \neq n_{k+1}^2$, then
$P_1 \cup_S P_2 = \{n_1^1 \rightarrow n_2^1 \ldots n_k^1 \rightarrow \{\{n_{k+1}^1 \rightarrow \ldots \rightarrow n_l^1\}, \{n_{k+1}^2 \rightarrow \ldots \rightarrow n_m^2\}\}$.

We extend the notion of merging paths to merging single-type trees. Let $T_1 = \{n \rightarrow \{t_1, t_2, \ldots, t_k\}\}$ and $T_2 = \{n' \rightarrow \{t'_1, t'_2, \ldots, t'_l\}\}$ be two trees, then their merger $T_1 \cup_S T_2$ is defined as follows:

1. $T \cup_S \{\} \stackrel{def}{=} \{\} \cup_S T \stackrel{def}{=} T$
2. if $n \neq n'$, $T_1 \cup_S T_2 = \{T_1, T_2\}$,(trees cannot be merged).
3. if $n = n'$, then let $T = \{\}$. For all paths p originating from the root in T_1 and T_2, do $T = T \cup_S p$. $T_1 \cup_S T_2 = T$.

The query-pattern of an XML query Q is a labeled-tree representing all data disclosed by Q, i.e., all data returned to the user or accessed by Q.

Definition 6. (Query-Pattern)
Let Q be the given XML query and $P_1, \ldots, P_n$ are path-expressions that occur in the RETURN or the WHERE clause of Q. If $P_i == P_j$ is a condition in the WHERE clause, we add a new leaf node labeled with a data-variable v to P_i and P_j. Substitute all query-variables in $P_1, \ldots, P_n$. Query pattern P is the labeled-tree produced by merging paths $P_1, \ldots, P_n$. Algorithm 1 shows the construction of the query-pattern.

Example 2. Consider the following XML query Q_2: FOR $x in //r LET $y := $x/d, $z := $x/a RETURN <answer> {$z/c} </answer> WHERE { $z/b == $y }. Let T_r be the tree in the return statement of Query Q_2. T_r specifies structure of query answer being returned to the user. To evaluate the query-answer $\$z/b$ and $\$y$ must be accessed. Query-pattern constructed from query Q_2 is shown in Fig. 2(c). □

Algorithm 1: Algorithm to construct query-pattern

input : Query Q
output: Query-Pattern Tree T

Let V = $\{v_1, \ldots, v_k\}$ be the set of variables defined in Q.
Let P = $\{p_1, \ldots, p_m\}$ be the set of path-expressions in RETURN and WHERE clause of Q.
$i \leftarrow 1$
$list \leftarrow \{\}$ /* List of sets, where each set contains path-expressions in WHERE clause of Q, such that their values are transitively equal*/
/* Extend the path-expressions with a data-variable, such that path-expressions equated in WHERE clause have same data-variable.*/
foreach *expression* ($p_l == p_n$) *in WHERE clause of* Q **do**
  if $p_l \in S$ *and* S *is a set in list* **then**
    Append leaf node of p_l to p_n
    Add p_n to S
  else if $p_n \in S$ *and* S *is a set in list* **then**
    Append leaf node of p_n to p_l
    Add p_l to S
  else
    Create a new data-variable v_i
    Append v_i to p_l and p_n
    Create a set $S = \{p_l, p_n\}$
    Add S to list
    $i \leftarrow i + 1$
/* Removing query-variables from path-expressions*/
for $i := 1$ *to* m **do**
  Let $T_i \leftarrow p_i$, where $p_i \in$ P
  Let $r \leftarrow$ root node of T_i
  repeat
    Substitute r in T_i, with its assigned value (by $:=$ or in operator) in Q
    $r \leftarrow$ root node of T_i
  until r *is a constant or '//'*
/* Uniting path-expressions to obtain query-pattern */
Initialize $T \leftarrow \{\}$
for $i := 1$ *to* m **do**
  $T \leftarrow T \cup_S T_i$
return T

Definition 7. (Protection Object)
A *simple security object* o is a node-labeled tree, where all distinct subtrees $t_1, t_2, \ldots, t_k$ of o have the same access permission as o. That is, for every proper subtree $t_i \in o$, $\lambda(o) = \lambda(t_i)$, where $\lambda(o)$ and $\lambda(t_i)$ denote the security classification of o and t_i respectively. Simple security objects are equivalent to node-level security classification. An *association security object* o is a node-labeled tree where every proper subtree $t_i \in o$, $\lambda(o) > \lambda(t_i)$ $(i = 1, \ldots, n)$.

We construct *Pattern Automatas* (PA) (similar to the tree-automatas in [13, 14]) to represent security objects.

Definition 8. (Pattern Automata)
Let E be a set of node-labels for elements, A be a set of node-labels for attributes, and let the label *pcdata* represent all text nodes. A Pattern Automata is defined as $\mathcal{X} = \{\Sigma, Q, q_0, Q_f, \delta\}$, where $Q = \{q_0, \ldots, q_n\}$ is a finite set of automaton states, $\Sigma = E \cup A \cup \{pcdata, //\}$ is automata alphabet, '//' is a symbol for self-or-descendant axis, q_0 is start state, $Q_f \subset Q$, $(q_0 \notin Q_f)$ is set of accepting final states, and δ is set of state transition rules.

Let $\sigma \in \Sigma$ is label of scanned node N of a query pattern and therefore the next input symbol for the automata, and $Q_c \subseteq Q$ is set of states associated with child nodes of N. A valid transition is of the form, $\sigma(q_i, \ldots, q_j) \rightarrow q_k$, where $\{q_i, \ldots, q_j\} \subseteq Q_c$, and q_k is state associated with N after scanning. For simplicity, we will often write transition rule in the form $\sigma(Q_t) \rightarrow q_k$, where $Q_t = \{q_i, \ldots, q_j\}$ is set of states required for transition. To distinguish data values from labels of elements and attributes, we write data values inside [ ]. If δ does not contain a valid transition rule, by default the state associated with the scanned node is q_0.

4 Security Analysis of Query Pattern

RXACL performs security analysis by evaluating query-pattern with the pattern automatas corresponding to protection objects. An accepting state is reached if the protection object is disclosed by the input pattern. These automatas can also be used for recognizing possible disclosure of security objects by query-patterns extended with document schema as discussed later in section 5.

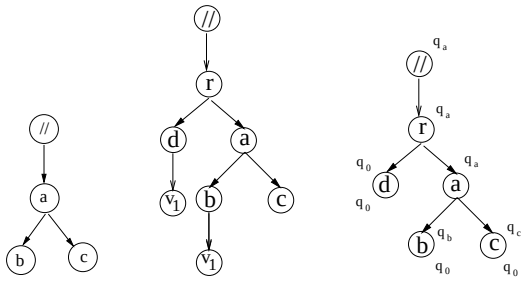
A pattern automaton $\mathcal{X}$ accepts a query-pattern $\mathcal{P}$ iff there is at least one accepting path of transitions that reads complete $\mathcal{P}$. For clarity, in this paper we allow use of wildcard symbol $(*)$ to represent *any* alphabet symbol. Let us now consider an example.

Example 3. The following automaton $\mathcal{X}_3 = \{\Sigma, Q, q_0, Q_f, \delta\}$ is a XML Pattern Automata that accepts query patterns disclosing association A_2 (see Fig. 2(b)). An accepting run of this automaton on query Q_2 is shown in Fig. 2(d). It means that answers of Q_2 disclose A_2. □

We now present an algorithm (See Alg. 2 and Proc. AddRules) to generate pattern automata for associations. [2] Given a query-pattern P a pattern automaton X is generated, such that on input P', X accepts iff P is contained in P'. Algorithm 2 performs

[2] Full version of our security analysis algorithms is given in technical report available at http://www.cse.sc.edu/research/isl/

$Q = \{q_0, q_a, q_b, q_c\}$,
$\Sigma = \{a, b, c, //\}$,
$q_0 = q_0$,
$Q_f = \{q_a\}$,
$\delta = \{\ b() \rightarrow q_b,$
$\quad c() \rightarrow q_c,$
$\quad a(q_b, q_c) \rightarrow q_a,$
$\quad *(q_a) \rightarrow q_a\}$



(a) **(b)** **(c)** **(d)**

Fig. 2. (a) Pattern automata example $\mathcal{X}_3$ (b) Example association A_2 (c) Query-pattern of Q_2 (d) States of $\mathcal{X}_3$ (q_i) on query-pattern of Q_2 as input.

a bottom up traversal of the association security object (a labeled-tree). At each step of traversal the label of current node is read. If the label is read for first time, it is added to pattern automaton's alphabet and a new state is also created. If the label denotes a self-or-descendant edge in the query-pattern then a transition rule with a wildcard (*) for read symbol is added to the pattern automata. Otherwise transition rule with symbol read at the current node is added.

Algorithm 2: Algorithm to generate pattern automata

input : Association pattern P
output: Pattern Automata X = $\{\Sigma, Q, q_0, Q_f, \delta\}$

Q $\leftarrow \{q_0\}$
$\Sigma \leftarrow \{//, pcdata\}$
$Q_f \leftarrow \{\}$
$\delta \leftarrow \{\}$
X $\leftarrow \{\Sigma, Q, q_0, Q_f, \delta\}$ Let S be a global stack
$S \leftarrow \emptyset$ /* S is a global stack used to remember states of child nodes during bottom-up traversal of P */
X $\leftarrow$ AddRules (P, X)
$Q_f \leftarrow$ pop(S)
return X

Theorem 1. Let Q be an XML query, P the query-pattern generated from Q (Def. 6), O an association object and AO the association-automata representing O. The association-automata AO accepts a input query-pattern P iff there exists an XML instance I such that the answer to Q over I discloses O.

Proof Sketch: ($\Rightarrow$) The pattern-automata performs bottom-up traversal of P, i.e., states of child nodes are evaluated before evaluating state for root node. Let n be a node in P scanned to detect disclosure of O. If n is a leaf node in O, there must exist a valid transition of form $\{n() \rightarrow q\} \in \delta$, where δ is the transition function of pattern-automata AO created by Algorithm 2. If n is an internal node with child nodes $\{n_1, \ldots, n_k\}$, Algorithm 2 generates a transition rule of the form $\{n(q_1, \ldots, q_k) \rightarrow q\}$, where $q_1, \ldots, q_k$ are states associated with $n_1, \ldots, n_k$ respectively. Clearly there exists an accepting path

Procedure `AddRules(`*Root, Pattern Automata*`)`

```
input : Pattern tree P, Pattern Automata X = { Σ, Q, q0, Qf, δ }
output: Modified Pattern Automata X
root ← root node of P
Qc ← { }
list ← child nodes of root
foreach node in list do
    X ← AddRules( node, X ) /* Perform bottom-up tree traversal */
if list ≠ ∅ then
    n ← Length of list
    while n > 0 do
        Qc ← Qc ∪ pop(S) /*Retrieve automata states after scanning child nodes */
        n ← n − 1
label ← LabelOf(root)
if label = '//' then
    foreach state q ∈ Qc do
        δ ← δ ∪ {*(q) → q}}
else
    Find set of transition rules R, of the form {label (Q') → q} in δ
    if R is empty then
        Create a new state q_new /* label(Q') is read for the first time*/
        Q ← Q ∪ {q_new}
        δ ← δ ∪ {label(Qc) → q_new}
        push(S,{q_new})
    else if Qc ≠ Q' for all rules in R then
        Create a new state q_new /* Transitions exist for labelQ' but are not applicable*/
        Q ← Q ∪ {q_new}
        δ ← δ ∪ {label(Qc) → q_new}
        push(S,{q_new})
    else
        push(S,{q}) /* An existing transition leading to state q is applicable*/
return X
```

of automata evaluation if the association pattern is traversed. Thus, pattern-automata finds the accepting path if it exists.

($\Leftarrow$) For this, we show how to construct instance I such that the answer to Q over I must contain O. Let ζ be a mapping from P to O with following properties: ζ maps (1) a constant to the same constant, (2) variable to $pcdata$, and (3) a arbitrary path p to //.

If there exists a ζ such that the pattern P' created from P by replacing all variables of P with $\zeta(v)$ and p with //, and O is a subtree of P' then we generate I as follows: (1) replace all mapped variables $v \in P$ with $\zeta(v)$, (2) replace all non-mapped variables in P with pcdata c, and (3) replace // with the empty path, i.e., remove //.

5 Security Analysis of Extended Query-Pattern

In addition to the structural information contained in the RETURN and the WHERE clauses of the query, a query answer also contains subtrees of the original XML document, where each returned subtree originates from one of the path-expressions in the RETURN clause. To incorporate this knowledge in our model, we define the notion of extended query-pattern.

Definition 9. (Extended Query-Pattern)
Let P denote a query-pattern and S the schema (ground-tree) of the XML document that Q is posed on. The extended-query-pattern (EQP) is defined as a set of trees $\{T_1, \ldots, T_m\}$, where $T_i (i = 1, \ldots, m)$ are constructed as follows: Let ν denote a symbol mapping from the symbols of P to the symbols of S such that: (1) for constants ν is an identity mapping, (2) ν maps the data-variables to the empty node $\emptyset$,

and (3) ν maps '//' to any ground path in S. We extend ν to map paths of P, such that given a path $p = \{n_1 \rightarrow n_2 \rightarrow \ldots \rightarrow n_l\}$, its mapping $\nu(p) = \{\nu(n_1) \rightarrow \nu(n_2) \rightarrow \ldots \rightarrow \nu(n_{l-1}) \rightarrow t_l\}$, where t_l is a tree rooted at $\nu(n_l)$ such that $\nu(p) \in S$. Finally, given paths $p_1, \ldots, p_k$ of of all leaf nodes in P we construct $T_i \in EQP$ as $T_i = \nu(p_1) \cup_S \nu(p_2) \cup_S \ldots \cup_S \nu(p_k)$ and $T_i \in S$ for all possible symbol mapping ν.

Theorem 2. Let Q be an XML query, S be the schema of XML document, EQP be the query-pattern extended with S, O an association object, and AO be the association-automaton representing O. If AO does not accept the extended query-pattern EQP, then the query is safe to answer for any XML document that satisfies S. That is for all XML instances over S the query Q will not disclose O.

Proof Sketch: Lets assume by contradiction that the query Q discloses an association object AO and the pattern-automata generated from AO does not accept the extended query-pattern. But then, either the specifying query itself discloses O, i.e., the union of the paths $p_1, \ldots, p_k$ in the FOR, LET, RETURN, and WHERE clause of Q disclose O, or the answer generated from any XML instance conforming to S together with $p_1 \cup_S p_2 \cup_S \ldots \cup_S p_k$ disclose O. But this is exactly the information used to generate the extended query-pattern. Using Theorem 1 this implies that the tree-automata must accept the extended query-pattern, which is a contradiction.

6 Conclusions

In this paper we have presented a bottom-up tree automata (pattern-automata) based technique for filtering XML association before query evaluation. We have also given algorithms for constructing query-pattern, pattern automata, detect disclosure of association security object in a query-pattern itself and query-pattern is extended with schema information. We have also shown that our security-analysis is complete, i.e., our method detects all possible disclosures.

We have considered only simple XQueries in this work. In future, we plan to extend our analysis to incorporate nested queries. At present our schema-level analysis requires the schema to be a single-type tree language (DTD or W3C XML schema). We also plan to extend our schema-level security analysis to incorporate regular tree languages, like RELAX NG.

References

1. Bertino, E., Castano, S., Ferrari, E.: Securing XML Documents with Author-X. IEEE Internet Computing **3** (2001)
2. Bertino, E., Castano, S., Ferrari, E., M.Mesiti: Specifying and Enforcing Access Control Policies for XML Document Sources. In: World Wide Web Journal. Volume 3. Baltzer Science Publishers (2000)
3. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: XML Access Control Systems: A Component-Based Approach. In: IFIP WG11.3 Working Conference on Database Security, The Netherlands (2000)

4. Kudo, M., Hada, S.: XML Document Security based on Provisional Authorizations. In: Proc. of the 7th ACM conference on Computer and Communications Security, Athens, Greece (2000)
5. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML Access Control using Static Analysis. In: CCS '03: Proceedings of the 10th ACM conference on Computer and communications security, ACM Press (2003) 73–84
6. Luo, B., Lee, D., Lee, W.C., Liu, P.: QFilter: Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting. In: Proc. of ACM Conference on Information and Knowledge Management (CIKM). (2004)
7. Gowadia, V., Farkas, C.: RDF metadata for XML Access Control. In: Proceedings of the 2003 ACM workshop on XML security, ACM Press (2003) 39–48
8. W3C Recommendation: XML Path Language (XPath) Version 1.0. (1999)
9. Bray, T., Paoli, J., Sperberg-McQueen, C.M.: Extensible Markup Language Language 1.0 specification. W3C Recommendation. (2000)
10. Thompson, H.S., Beech, D., Maloney, M., Mendelsohn, N.: XML Schema Part 1: Structures. Technical report, W3C Consortium (2001)
11. Murata, M., Lee, D., Mani, M., Kawaguchi, K.: Taxonomy of XML Schema Languages using Formal Language Theory. ACM Trans. on Internet Technology (2005)
12. Fernández, M., Malhotra, A., Marsh, J., Nagy, M., Walsh, N.: XQuery 1.0 and XPath 2.0 Data Model. Technical report, W3C (2003) W3C Working Draft 12 November 2003.
13. Chidlovskii, B.: Using Regular Tree Automata as XML Schemas. In: ADL '00: Proceedings of the IEEE Advances in Digital Libraries 2000, IEEE Computer Society (2000) 89
14. Neven, F.: Automata theory for XML researchers. SIGMOD Rec. **31** (2002) 39–46