

Tree Automata for Schema-level Filtering of XML Associations

Vaibhav Gowadia and Csilla Farkas

Information Security Laboratory
Department of Computer Science and Engineering
University of South Carolina, Columbia, SC 29208
{gowadia, farkas}@cse.sc.edu

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. We present the concepts of a simple security object and an association security object. Our model allows us to express and enforce access control on XML trees and their associations. We propose a query-filtering technique that evaluates XML queries to detect disclosure of association-level security objects. We use tree automata to model security objects. Intuitively a query Q discloses a security object o if and only if the (tree) automata corresponding to o accepts Q . We show that our schema-level method detects all possible disclosures, i.e., it is complete.

Keywords: Access control, association object, flexible security granularity, XML security, tree automata.

ACM Classification: H.2.7 (Database Management – Database Administration – security, integrity and protection), D.4.6 (Operating Systems – Security and Protection – Access controls)

1. INTRODUCTION

Several XML access control models have been developed recently (Bertino *et al*, 2001, 1999, 2000; Damiani *et al*, 2000; Dridi and Neumann, 1998; Gabillon and Bruno, 2002; Kudo and Hada, 2000; Murata *et al*, 2003; Luo *et al*, 2004). They are based on traditional access control lists and provide extensions to XML syntax. Existing models 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 not allowed 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.

To illustrate the need of access control for data associations, we present an example in the medical domain. Assume that XML format is used for storing patient records. The DTD for patients' health records is shown in Figure 1. Alice, who is an intern at the hospital, needs limited access to the database. Her duties involve two main tasks: 1) Alice contacts patients to collect feedback about their treatments; thus, Alice is allowed to read <name> and <phone> elements, 2) Alice prepares statistical reports based on Age, Race and Diagnosis of the patients. These tasks require that Alice

Copyright© 2006, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 12 April 2005

Communicating Editor: Julio Cesar Hernandez

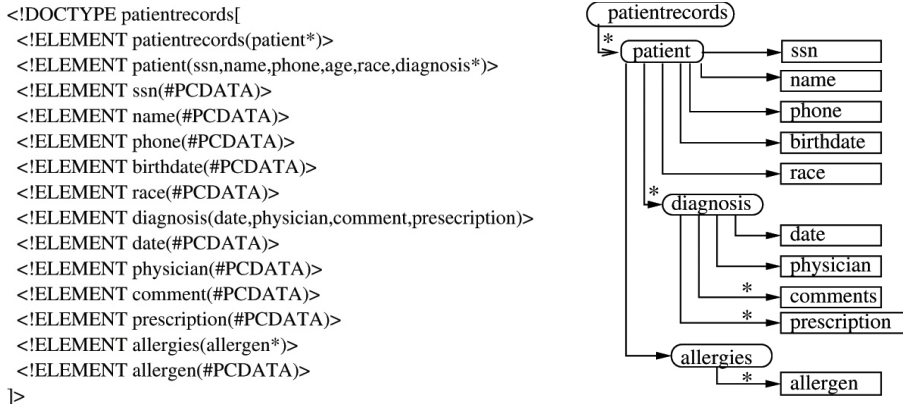


Figure 1: (a) An example of a DTD (b) Tree representation of DTD

is allowed to access both contact and diagnosis information for all patients. However, Alice is not authorized to access data about the name and diagnosis of the patients. Both, the functionality requirements of Alice’s work and the security restrictions cannot be satisfied using traditional access control list based methods.

The RXACL architecture, introduced in Gowadia and Farkas (2003), 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 Gowadia and Farkas (2003) we proposed a technique to enforce association-based access control at data-level (i.e., check for security violation after query processing). Data-level access control is outside the scope of this paper. In this paper we extend RXACL architecture by presenting techniques for performing a security check before the query is processed. Our work is similar to those proposed by Murata *et al* (2003) and Luo *et al* (2004). However, their methods support node-level security objects only. The automata model proposed by these authors is not sufficient to model association-level security objects. In this paper, we use bottom-up tree automata to represent security objects.

We propose query pre-processing techniques to recognize disclosure of association level security objects by XML queries. This analysis is data-independent and performed before the query is processed. The result of security evaluation can be either (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. For option 3, data-level analysis is required to evaluate whether a security violation occurs or not.

We present a two-layered association filtering method. First we detect disclosure of association in a given query-pattern, i.e., in 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 (Clark and DeRose, 1999)). We model association security objects with *pattern automata* (Definition 8). A pattern automaton takes (extended) query-patterns as input and accepts them if and only if the input discloses the security object represented by the pattern automata. The main technical contributions of this paper are the development of pattern automata 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 constructions used in this paper. Section 4 presents algorithms for constructing query-patterns, association pattern automata and to detect association disclosures. Section 5 introduces the notion of extended query-pattern and presents a schema-level security analysis of query. Our conclusions are presented in section 6 along with recommendations for future work.

2. RDF-BASED XML ACCESS CONTROL ARCHITECTURE

Figure 2 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 an 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 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 (Gowadia and Farkas, 2003). The history component keeps track of answered query-patterns and data released to each user.

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-patterns are now checked for disallowed objects. If no disallowed association-level security object is disclosed in extended query-patterns the query answer is labeled *safe*. Otherwise, it is labeled *unsafe*. The query is submitted to XML query engine for further processing. The results of unsafe query's evaluation are further evaluated for possible data-level violations as described in Gowadia and Farkas (2003). 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.

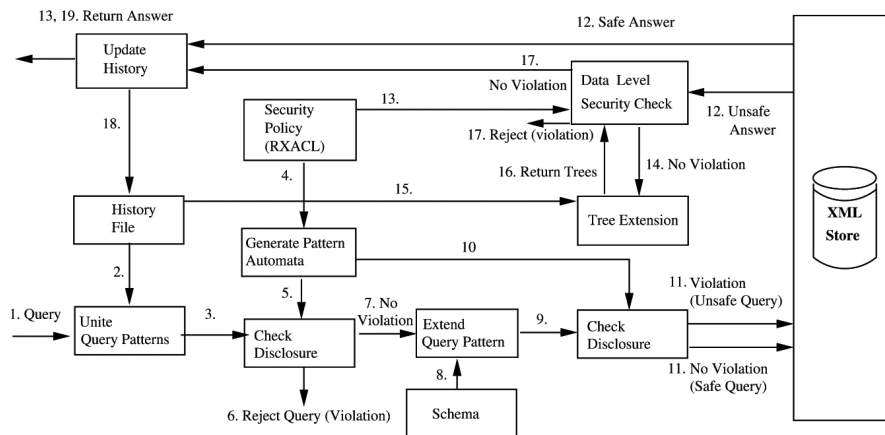


Figure 2: RXACL architecture for enforcing XML access control

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.
3. If t_1, t_2, \dots, t_k are trees, then $\{n \rightarrow \{t_1, t_2, \dots, t_k\}\}$ is a tree. In this case we say that $\{n \rightarrow \{t_1, t_2, \dots, t_k\}\}$ represents the tree whose root n has outgoing edges to subtrees t_1, t_2, \dots, 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 one child node, a text node. Also, any two attribute nodes of a given element cannot have the 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 $pdata$. A labeled tree is called a *ground tree* if all of its nodes are labeled with constants.

Definition 2. (Path-expression)

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

DTD (Bray *et al*, 2000) and W3C XML Schema (Thompson *et al*, 2001) require that all child nodes of any given node must have unique names. Trees satisfying this constraint are called *single-type trees* (See Murata *et al*, 2005) for discussion on single-type tree grammars).

We consider XQuery syntax (Fernández *et al*, 2003) of the following form:

Definition 3. (XML Query)

An XML query Q is of the following form:

```
FOR  $v_0$  in  $P_0$ 
LET  $v_1 := P_1, \dots, v_l := P_l$ 
RETURN  $\{n \rightarrow \{\overline{v_k}, \dots, \overline{v_j}\}\}$ 
WHERE  $(\overline{v_i} == \overline{v_j}$  and ... and  $\overline{v_l} == \overline{v_m})$ 
```

where, $v_i (i = 0, \dots, l)$ are variables of query (we refer to them as *query-variables* in the rest of this paper), $\overline{v_i} (i = 0, \dots, m)$ represent path-expressions $\{v_i \rightarrow p'\}$ ($i = 1, \dots, l$) where p' is a path-expression that does not contain any query-variables, $P_i (i=0, \dots, l)$ 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 $V = \{v_1, v_2, \dots, v_l\}$ be the set of query-variables defined in Q , and $\overline{V} = \{\overline{v_1}, \overline{v_2}, \dots, \overline{v_m}\}$ 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 $\$v_i = \{p_1 \rightarrow \dots \rightarrow p_i\}$, and $\$v_j = \{p'_1 \rightarrow \dots \rightarrow p'_m\}$ be two assignments in the FOR or LET clause of the XML query. A *variable substitution* replaces $\$v_i$ in the second assignment with $\{p_1 \rightarrow \dots \rightarrow p_i\}$.

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 $P_1 = \{n_1^1 \rightarrow n_2^1 \rightarrow \dots \rightarrow n_k^1 \rightarrow n_{k+1}^1 \rightarrow \dots \rightarrow n_l^1\}$ and $P_2 = \{n_1^2 \rightarrow n_2^2 \rightarrow \dots \rightarrow n_k^2 \rightarrow n_{k+1}^2 \rightarrow \dots \rightarrow n_m^2\}$ be two ground path-expressions over the same schema. We define *merge* of path expressions as follows: if $n_1^1 = n_1^2, n_2^1 = n_2^2, \dots, n_k^1 = n_k^2$, and $n_{k+1}^1 \neq n_{k+1}^2$, then

$$P_1 \cup_s P_2 = \{n_1^1 \rightarrow n_2^1 \rightarrow \dots \rightarrow n_k^1 \rightarrow \{\{n_{k+1}^1 \rightarrow \dots \rightarrow n_l^1\}, \{n_{k+1}^2 \rightarrow \dots \rightarrow n_m^2\}\}$$

We extend the notion of merging paths to merging single-type trees. Let

$T_1 = \{n \rightarrow \{t_1, t_2, \dots, t_k\}\}$ and $T_2 = \{n' \rightarrow \{t'_1, t'_2, \dots, t'_l\}\}$ 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 . That is, all data returned to the user or accessed to evaluate the query.

Definition 6. (Query-Pattern)

Let Q be the given XML query and P_1, \dots, 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, \dots, P_n . Query pattern P is the labeled-tree produced by merging paths P_1, \dots, P_n . Algorithm 2 shows the construction of the query-pattern.

Example 2. Consider the following XML query Q :

```
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 Q . 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 is shown in Figure 3(c).

Definition 7. (Protection Object)

A *simple security object* o is a node-labeled tree, where all distinct subtrees t_1, t_2, \dots, 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

Algorithm 1: Algorithm to construct query-pattern

```

input : Query Q
output: Query-Pattern Tree T
Let  $V = \{v_1, \dots, v_k\}$  be the set of variables defined in Q.
Let  $P = \{p_1, \dots, p_m\}$  be the set of path-expressions in RETURN or WHERE clause of Q.
i ← 1
list ← {} /* 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 ← 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 ← root node of  $T_i$ 
    | repeat
    | | Substitute r in  $T_i$ , with its assigned value (by := or in operator) in Q
    | | r ← root node of  $T_i$ 
    | until r is a constant or '//'
/* Uniting path-expressions to obtain query-pattern */
Initialize T ← {}
for i := 1 to m do
    |  $T \leftarrow T \cup S T_i$ 
return T

```

$\lambda(t_i)$ 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 $t_i \in o$, $\lambda(o) > \lambda(t_i)$ ($i = 1, \dots, n$).

We construct *Pattern Automata* (PA) to represent security objects. Due to the page limitation, we only show the representation of association security objects, which is more difficult than the representation of the simple security object.

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 $X = \{\Sigma, Q, q_0, Q_f, \delta\}$, where $Q = \{q_0, \dots, q_n\}$ 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 $\sigma \in \Sigma$ be the label of a scanned node N in the given 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, $\sigma (q_i, \dots, q_j) \rightarrow q_k$, where $\{q_i, \dots, q_j\} \subseteq Q_c$, 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 $Q_t = \{q_i, \dots, q_j\}$ 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 automata corresponding to protection objects. An accepting state is reached if the protection object is disclosed by the input pattern. These automata 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 X accepts a query-pattern P iff there is at least one accepting path of transitions that reads complete 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 $X = \{\Sigma, Q, q_0, Q_f, \delta\}$ is a XML Pattern Automaton that accepts query patterns disclosing association A (Figure 3(b)). An accepting run of this automaton on query Q' is shown in Figure 3(d). It means that answers of Q' disclose A.

We now present Algorithm 2 and Procedure AddRules to generate pattern automata for associations. 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 3 performs 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.

Next we present Algorithm 3 and Procedure EvaluateQueryPattern that runs a given pattern automata on a given query-pattern. The query pattern is accepted if the association object represented by the pattern automata is disclosed by the query pattern. A query-pattern may contain variables

$$Q = \{q_0, q_a, q_b, q_c\}$$

$$\Sigma = \{a, b, c, //\}$$

$$q_0 = q_0,$$

$$Q_f = \{q_a\},$$

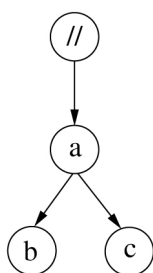
$$\delta = \{ b() \rightarrow q_b,$$

$$c() \rightarrow q_c,$$

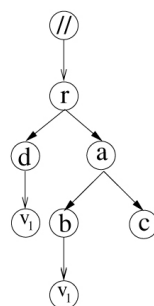
$$a(q_b, q_c) \rightarrow q_a,$$

$$*(q_a) \rightarrow q_a\}$$

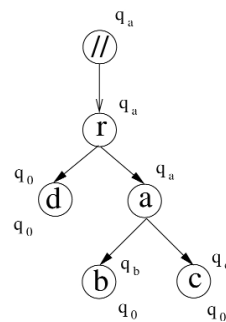
(a)



(b)



(c)



(d)

Figure 3: (a) Pattern Automaton example (b) Example association A (c) Query-pattern of Q' (d) States of X (q_i) on query-pattern of Q' as input

Algorithm 2: Algorithm to generate pattern automata

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

$Q \leftarrow \{q_0\}$
 $\Sigma \leftarrow \{/, pcd\}$
 $Q_f \leftarrow \{\}$
 $\delta \leftarrow \{\}$
 $X \leftarrow \{\Sigma, Q, q_0, Q_f, \delta\}$ 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 */
 $X \leftarrow \text{AddRules}(P, X)$
 $Q_f \leftarrow \text{pop}(S)$
return X

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

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

root \leftarrow root node of P
 $Q_c \leftarrow \{\}$
list \leftarrow child nodes of root
foreach *node* in list **do**
 $X \leftarrow \text{AddRules}(\text{node}, X)$ /* Perform bottom-up tree traversal */
if list $\neq \emptyset$ **then**
 $n \leftarrow$ Length of list
 while $n > 0$ **do**
 $Q_c \leftarrow Q_c \cup \text{pop}(S)$ /*Retrieve automata states after scanning child nodes */
 $n \leftarrow n - 1$
label $\leftarrow \text{LabelOf}(\text{root})$
if label = '/' **then**
 foreach *state* $q \in Q_c$ **do**
 $\delta \leftarrow \delta \cup \{*(q) \rightarrow q\}$
else
 Find set of transition rules R , of the form $\{\text{label}(Q') \rightarrow q\}$ in δ
 if R is empty **then**
 Create a new state q_{new} /* label(Q') is read for the first time*/
 $Q \leftarrow Q \cup \{q_{new}\}$
 $\delta \leftarrow \delta \cup \{\text{label}(Q_c) \rightarrow q_{new}\}$
 push($S, \{q_{new}\}$)
 else if $Q_c \neq Q'$ for all rules in R **then**
 Create a new state q_{new} /* Transitions exist for label Q' but are not applicable*/
 $Q \leftarrow Q \cup \{q_{new}\}$
 $\delta \leftarrow \delta \cup \{\text{label}(Q_c) \rightarrow q_{new}\}$
 push($S, \{q_{new}\}$)
 else
 push($S, \{q\}$) /* An existing transition leading to state q is applicable*/
return X

(called data-variables) at the leaves, due to the equalities in the WHERE clause. Data-variables of a query may correspond to *pcdata* constants of a pattern automaton, thus indicating potential disclosure. The procedure EvaluateQueryPattern is data independent. Algorithm 3 analyzes the input query pattern by analyzing all possible transitions of the pattern automata in parallel. If there is at least one accepting run of the automata then the algorithm returns true, otherwise false.

Algorithm 3: Algorithm to decide whether a pattern-automaton accepts input query-pattern

input : Pattern Automaton $X = \{\Sigma, Q, q_0, Q_f, \delta\}$, Query-pattern tree T
output: True if X accepts T , otherwise false

Let S be a global stack
 /* S is used to remember states of child nodes */
 Let $root$ be the root node of T
 $returnvalue \leftarrow EvaluateQueryPattern(root, X)$ /* Store reachable states on S */
 $list \leftarrow pop(S)$
foreach set *in* $list$ **do**
 if $(set \cap Q_f) \neq \{\}$ **then**
 /* Final state after reading T can be an accepting state */
 return true
 /* Final state after reading T cannot be an accepting state */
return false

Procedure $EvaluateQueryPattern(Node, X)$

input : Node $node$, Pattern Automata X
output: Compute set of reachable states, when X reads the input tree rooted at $node$.
 The result is stored on global stack S .

/* Evaluate mapping with a bottom-up traversal of T */
 $ListOfStateSets \leftarrow ()$ /* List of state sets that X may enter in upon processing $node$ */
if $node$ *is a leaf* **then**
 if $LabelOf(node)$ *is a data-variable in* T **then**
 Add $\{q_0\}$ to $ListOfStateSets$ /* data variables can always have safe mappings */
 else if $rule \{LabelOf(node)() \rightarrow q\} \in \delta$ **then**
 Add $\{q\}$ to $ListOfStateSets$ /* Constants are mapped to the same constant */
 else
 Add $\{q_0\}$ to $ListOfStateSets$ /* Default transition on no mapping */
 push($S, ListOfStateSets$)
 return
 $list \leftarrow$ child nodes of $node$
 $ListOfChildStateSets \leftarrow ()$ /* List of sets, where each set Q_c contains possible states for child nodes
foreach $childnode$ *in* $list$ **do**
 EvaluateQueryPattern($childnode, X$)
 $ListOfChildStateSets \leftarrow ListOfChildStateSets \times pop(S)$
 $\sigma \leftarrow LabelOf(node)$
 Compute R , a set of valid transitions (Def. 3.8) on σ and Q_c , where $Q_c \in ListOfChildStateSets$
if R *is empty* **then**
 /* Default transition on no mapping */
 $ListOfStateSets \leftarrow (\{q_0\})$
 else
 /* Follow all possible transitions */
 $ListOfStateSets \leftarrow (\{q_1\}, \dots, \{q_k\})$, where $(\sigma(Q_i) \rightarrow q_i) \in R$ ($i = 1, \dots, k$)
 push($S, ListOfStateSets$)
return

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 perform 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 $\{n_1, \dots, n_k\}$, Algorithm 2 generates a transition rule of the form $\{n(q_1, \dots, q_k) \rightarrow q\}$, where q_1, \dots, q_k are states associated with n_1, \dots, n_k respectively. Clearly there exists an accepting path 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 O 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) an 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 $\{T_1, \dots, T_m\}$, where $T_i (i = 1, \dots, m)$ are constructed as follows: Let v denote a symbol mapping from the symbols of P to the symbols of S such that:

- for constants v is an identity mapping.
- v maps the data-variables to the empty node \emptyset .
- v maps $'//'$ to any ground path in S .

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

Algorithm 4 and Procedure EvaluateExtendedQueryPattern decide whether a pattern automaton accepts the extended query pattern. Unlike Procedure EvaluateQueryPattern, Procedure EvaluateExtendedQueryPattern evaluates data variables too. We now extend our formalism to be able to evaluate data variables in the query pattern.

Definition 10. (Self-or-ancestor state set)

Let $X = \{\Sigma, Q, q_0, Q_f, \delta\}$ a XML Pattern Automata, and T be an arbitrary XML pattern given as input. We say that *self-or-ancestor state set* $Q_{aN} = \{q_1, q_2, \dots, q_k\}$ is a set of states, such that $Q_{aN} \subseteq Q$, and X may enter a state q_i by scanning a node labeled N in T or any ancestor node of N , iff $q_i \in Q_{aN}$.

Algorithm 4: Algorithm to decide whether a pattern-automaton accepts an extended query-pattern

input : Pattern Automaton $X = \{\Sigma, Q, q_0, Q_f, \delta\}$, Extended query-pattern tree T
output: True if X accepts T , otherwise false

if Σ contains pcdata values **then**
 Construct self-or-ancestor state set Q_{aN} for each pcdata value in Σ
 Construct data-characteristic state set Q_d
 Construct variable-characteristic state set Q_{v_i} for each data variable v_i in T

Let S be a global stack
/* S is used to remember states of child nodes */
Let $root$ be the root node of T
EvaluateExtendedQueryPattern($root, X$)
:
:
:

Procedure EvaluateExtendedQueryPattern($Node, X$)

input : Node $node$, Pattern Automata X
output: Compute set of reachable states, when X reads the input tree rooted at $node$.
 The result is stored on global stack S .

/* Evaluate mapping with a bottom-up traversal of T */
Let $ListOfStateSets$ be the list of state sets that X may enter in upon processing $node$

if $node$ is a leaf **then**
 if $LabelOf(node)$ is a data-variable in T **then**
 foreach rule of form $\{pcdata() \rightarrow q\} \in \delta$ **do**
 Add $\{q\}$ to $ListOfStateSets$ /* Map data variable to all pcdta values*/
 :
 :
 return

:
:
/* Validate matching of data-variables*/
Compute $Q_{required} \leftarrow Q_1 \cap Q_2 \dots \cap Q_k$, where $(\sigma(Q_i) \rightarrow q_i) \in R$ ($i = 1, \dots, k$)
Compute V , set of data-variables in subtree of childnode

if V is not empty **then**
 foreach $q \in Q_{required}$ **do**
 foreach Q_{aN} constructed for Σ **do**
 if $q \in Q_{aN}$ and $q \in Q_{v_i}$ and $q \in Q_d$ **then**
 if v_i is not assigned any value **then**
 $v_i \leftarrow N$
 else if $N \neq$ value previously assigned to v_i **then**
 Remove nodes labeled v_i in T and restart Algorithm 6

push($S, ListOfStateSets$)
return

Definition 11. (Data-characteristic state set)

Let $X = \{\Sigma, Q, q_0, Q_f, \delta\}$ a XML Pattern Automata, and T be an arbitrary XML pattern given as input. Data-characteristic state set $Q_d = \{q_1, q_2, \dots, q_k\}$ is a set of states, such that $Q_d \subseteq Q$ and X can enter a state $q_i \in Q_d$ after scanning a node N in T , iff exactly one pcdta constant occurs in subtree under node N .

Definition 12. (Variable-characteristic state set)

Let $X = \{\Sigma, Q, q_0, Q_f, \delta\}$ a XML Pattern Automata, and T be an arbitrary XML pattern with data-

variables $V = \{v_1, v_2, \dots, v_k\}$. For any variable $v_i \in V$, its *variable-characteristic state set* $Q_{v_i} = \{q_1, \dots, q_k\}$ is a set of states, such that $Q_{v_i} \subseteq Q$, and X may enter q_i on scanning node (in T) labeled v_i or on scanning an ancestor node of such node.

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: Let us 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, \dots, 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 $p_1 \cup_S p_2 \cup_S \dots \cup_S p_k$ 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 presented a bottom-up tree automata (pattern-automata) based technique for filtering XML association before query evaluation. We gave algorithms for constructing query-pattern, pattern automata, and for the detection of disclosure of an association security object in a query-pattern. In addition we extended the query-pattern with schema information to evaluate all data that the query answer may reveal to the user. We also showed that our security-analysis is complete, i.e., our method detects all possible disclosures.

We have considered only simple XQueries in this work. In our future work, we hope 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). In future work we hope to extend our schema-level security analysis to incorporate regular tree languages, such as RELAX NG. Finally, we plan to integrate query-optimization with security analysis.

7. ACKNOWLEDGEMENTS

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

REFERENCES

- BERTINO, E., CASTANO, S. and FERRARI, E. (2001): Securing XML documents with author-X. *IEEE Internet Computing* 5(3):21–31.
- BERTINO, E., CASTANO, S., FERRARI, E. and MESITI, M. (1999): Controlled access and dissemination of XML documents. In *Proc. of 2nd ACM Workshop on Web Information and Data Management*, 22–27, Kansas City.
- BERTINO, E., CASTANO, S., FERRARI, E. and MESITI, M. (2000): Specifying and enforcing access control policies for XML document sources. *World Wide Web* 3(3):139–151.
- BRAY, T., PAOLI, J. and SPERBERG-McQUEEN, C. M. (2000): Extensible markup language language 1.0 specification. World Wide Web Consortium, W3C recommendation edition. <http://www.w3.org/TR/2000/REC-xml-20001006>. Accessed 06-Oct-2000.
- CLARK, J. and DeROSE, S. (1999): XML path language (XPath) version 1.0. World Wide Web Consortium, W3C recommendation edition. <http://www.w3.org/TR/xpath>. Accessed 27-Jun-2001.
- DAMIANI, E., DI VIMERCATI, S. D. C., PARABOSCHI, S. and SAMARATI, P. (2000): XML access control systems: A component-based approach. In *Proceedings of the IFIP WG11.3 Working Conference on Database Security*, 39–50, The Netherlands.
- DRIDI, F. and NEUMANN, G. (1998): Towards access control for logical document structure. In *Proc. of the Ninth International Workshop of Database and Expert Systems Applications*, 322–327, Vienna, Austria.

- FERNÁNDEZ, M., MALHOTRA, A., MARSH, J., NAGY, M. and WALSH, N. (2003): XQuery 1.0 and XPath 2.0 data model. World Wide Web Consortium, working draft edition. <http://www.w3.org/TR/query-datamodel/>. Accessed 12-Dec-2003.
- GABILLON, A. and BRUNO, E. (2002): Regulating access to XML documents. In *Das'01: Proceedings of the fifteenth annual working conference on Database and application security*, 299–314, Norwell, MA, USA: Kluwer Academic Publishers.
- GOWADIA, V. and FARKAS, C. (2003): RDF metadata for XML access control. In *Proceedings of the 2003 ACM workshop on XML security*, 39–48, ACM Press.
- KUDO, M. and HADA, S. (2000): XML document security based on provisional authorization. In *CCS '00: Proceedings of the 7th ACM conference on Computer and communications security*, 87–96, New York, NY, USA: ACM Press.
- LUO, B., LEE, D., LEE, W.-C. and LIU, P. (2004): QFilter: fine-grained run-time XML access control via NFA-based query rewriting. In *CIKM '04: Proceedings of the Thirteenth ACM conference on Information and knowledge management*, 543–552, New York, NY, USA: ACM Press.
- MURATA, M., LEE, D., MANI, M. and KAWAGUCHI, K. (2005): Taxonomy of XML schema languages using formal language Theory. *ACM Trans. on Internet Technology*.
- MURATA, M., TOZAWA, A., KUDO, M. and HADA, S. (2003): XML access control using static analysis. In *CCS '03: Proceedings of the 10th ACM conference on Computer and communications security*, 73–84, ACM Press.
- THOMPSON, H. S., BEECH, D., MALONEY, M. and MENDELSON, N. (2001): XML schema Part 1: Structures. Technical report, W3C Consortium. <http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/>. Accessed 12-May-2005.

BIOGRAPHICAL NOTES

Vaibhav Gowadia is a doctoral student in the Department of Computer Science and Engineering at the University of South Carolina, Columbia. His current research interests include information systems security, semi-structure data security, network security and semantic web. His dissertation focuses on development of access control technologies for XML. He received a MS in Computer Engineering from University of South Carolina Columbia and Bachelor of Technology in Instrumentation and Control Engineering from National Institute of Technology, Jalandhar, India.



Vaibhav Gowadia

Csilla Farkas is an assistant professor at the Department of Computer Science and Engineering of University of South Carolina, Columbia. Her current research interests include information systems security, semi-structure data security, database inferences, intrusion detection and database management systems. Csilla Farkas received her PhD from George Mason University, Fairfax. In her dissertation she studied the inference and aggregation problems in multilevel secure relational databases. She received a MS in computer science from George Mason University and BS degrees in computer science and geology from SZAMALK, Hungary and Eotvos Lorand University, Hungary, respectively.



Csilla Farkas