A General Approach to Securely Querying XML

Ernesto Damiani [a] Majirus Fansi [b] Alban Gabillon [b] Stefania Marrara [a]
[a] *Università degli Studi di Milano*
Dipartimento di Tecnologie dell'Informazione via Bramante 65 26013 Crema (CR), Italy
[b] *Université de Pau et des Pays de l'Adour*
IUT des Pays de l'Adour 40 000 Mont-de-Marsan, France

Abstract

Access control models for XML data can be classified in two major categories: node filtering and query rewriting systems. The first category includes approaches that use access policies to compute secure user views on XML data sets. User queries are then evaluated on those views. In the second category of approaches, authorization rules are used to transform user queries to be evaluated against the original XML dataset. The aim of this paper is to describe a model combining the advantages of these approaches and overcoming their limitations. The model specification is given using a Finite State Automata, ensuring generality and easiness of standardization w.r.t. specific implementation techniques

Key words:
PACS:

---

1. Introduction

In the last few years, the *eXtensible Markup Language* (XML)[5] has become the format of choice for data interchange. XML-based systems are now widely deployed in a number of application fields. This success has triggered a growing interest in XML security, and several schemes for XML access control have been proposed. They can be classified in two major categories: *node filtering* and *query rewriting* techniques. The first category includes a number of approaches (e.g., [1], [2], [18], [22]; for a complete survey, see [18]) that use access policies to compute *secure views* on XML data sets. User queries are then evaluated on those views. Although views can be prepared off-line, in general, view-based enforcement schemes suffer from high maintenance and storage costs, especially for a large XML repository.

XML access control via *query rewriting* ([21], [20], [19], [16], [15], [7]) has been proposed as a way to remedy these shortcomings. According to this approach, access control rules are not directly applied to the XML dataset to be protected; rather, they are used to translate potentially *unsafe* user queries into *safe* ones, to be evaluated against the original XML dataset. Most current proposals translate the policy's access control rules (ACR) to nondeterministic finite automata (NFSA) to rewrite user queries. However, for policies that include many ACRs, NFSA backtrackings may cause unacceptable overhead. More importantly, NFSA-based models are not entirely suitable for system specification and standardization. Another serious concern is that few of these models provide users with a safe schema representing the information that they are allowed to access. Disclosing the original schema may cause unwanted information leaks. In this paper, we describe our Deterministic Finite Automaton (DFA) based query rewriting approach (Section 2) that overcomes the drawbacks of the NFA-based Systems. The main contributions of this

---

Email addresses: `damiani, marrara@dti.unimi.it`, `janvier-majirus.fansi@etud.univ-pau.fr`, `alban.gabillon@univ-pau.fr` (Stefania Marrara).

work include:

- A security model based on authorization attributes for XML (Section 2.1) in which the security designer inserts the attributes in the XML Schema of the document collection via a GUI. We then obtain a policy-dependent view of the schema (or annotated schema).
- A formalization based on deterministic automata with a high level of generality (an automaton can be implemented in different ways) and suitable for standardization of the enforcement technique. From this formalization we straightforwardly derive algorithms for computing the user view of the schema (Section 2.2) and the rewriting DFA (Section 2.3) from the annotated schema.
- A way to exploit the standard operators `EXCEPT` and `UNION` of XPath to produce a sound and complete rewriting procedure (Section 2.4) of the user query. Detailed Examples (Section 2.6) illustrate the approach.
- A proof that our approach is sound and complete by means of a formal proof of correctness (Section 2.7). The complexity analysis (Section 2.8) shows that the entire procedure is efficient as it is linear with the size (i.e the number of element definitions) and the depth of the repository schema.
- Finally we propose an approach to securely handle XUpdate [8] commands over views. Namely, the authorization designer annotates the repository schema with some write attributes (*insert, update, delete*). The annotated schema is afterward translated into a Deterministic Finite Automaton (*for updates*). Rewriting an XUpdate request into a safe one is done in two steps:
  - · Whenever a user sends an XUpdate request over her view, we first rewrite the expression selecting the nodes to be updated according to the principles described in Section 3. This step is necessary since the user should not be able to update nodes she is not entitled to see.
  - · Then, we rewrite the XUpdate command over the DFA for updates in order to obtain a safe query (i.e a query updating the nodes the user is permitted to update).

In Section 4 we present the experimental results obtained with a prototype tool, while Related Work is discussed in Section 5. Finally, Section 6 concludes this paper and discusses future work.

1.1. *XML Security and actual standards*

With this work, we propose a new use of XPath for writing and enforcing security policies on XML data. Since the beginning of the XACML ([29]) discussion, the security community has focused on three important points:

- a security policy has to define clearly the objects target of the security enforcement;
- every object, target of a user's request, has to be compared with the policy in order to outline which rules have to be applied;
- there exist two different way to securely manage XML data:
  - · use of XPath both to define the target object in the security policy, and to access the object during a request (road followed by XACML);
  - · use of schema annotations to define the security requirements and of XPath to access the protected data.

In both cases, the XPath standard, as it is now, is not the best choice to address XML objects in a security environment, because it presents some important problems (e.g. hidden paths) w.r.t. some advantages (e.g. relative expressions) whose importance and necessity are questionable. For this reason, with this work, we suggest to the W3C to break the monolithic XPath standard into a *two layers* language, where each layer is a well-defined sub-language useful for different research communities. In our setting, we suggest to break XPath into $XPath--$ plus $XPath_{rel}$, where $XPath--$ expressions can be informally defined as follows:

`XPath--`:= $\varepsilon | l | * | p_1/p_2 | \; //p_1 | p[q]$ where p_1 and p_2 are `XPath--` expressions; ε, l, $*$ denote the empty path, a label and a wildcard, respectively; $/$ and $//$ stand for child-axis and descendant-or-self-axis; and finally, q is called a qualifier. We rewrite the request in the subset ζ:=$\{\varepsilon | l | p_1/p_2 | p[q]\}$ of `XPath--` using the functions *union* and *except*. ζ is `XPath--` without descendant-or-self axis ($//$) and wildcards ($*$). Obviously we can define $XPath_{rel}$ as XPath minus $XPath--$.

The formal definition of $XPath--$ by the W3C would provide the XML security community of a common, standard, language to access XML objects in the definition of XML security policies and requests management avoiding the problems given by relative paths.

On another side, updating XML data can still be considered a research issue (e.g. see [9,10,8,11,12]); however at least some building blocks of a data manipulation language for XML are now firmly in place. XUpdate is an XML-based host language for instructions tailored for update tasks. In other words, it expresses updates as well-formed XML documents; specifically, each update is represented by an `xupdate:modifications` element. XUpdate has now over a dozen implementations; this relative success is due to the fact that it is easy to understand and simple to implement. XUpdate operations have a required `select` attribute. The value of this attribute is a XPath expression which selects the nodes to update, referred to as *context* nodes. Besides updating XML content, XUpdate operations can create and delete entire XML fragments. An example XUpdate instruction is `<remove select='//vehicles'>`. This command is to remove from a document every fragment which root is an element named `vehicles`. An `xupdate:modifications` element must have a version attribute, indicating the version of XUpdate that the update requires. For the version of XUpdate released in 2000 as working draft, the value should be 1.0. The entire syntax released in 2000 allows the following types of elements:

- xupdate:insert-before
- xupdate:insert-after
- xupdate:append
- xupdate:update
- xupdate:remove
- xupdate:rename
- xupdate:variable
- xupdate:value-of
- xupdate:if

The interested reader can refer to [8] for a complete description of the XUpdate Language.

2. DFA-based Query Rewriting

In this section we present a novel approach for rewriting potentially unsafe user queries into safe ones. Our technique is based on *Deterministic Finite Automata* (DFA). We exploit the tree nature of the XML Schema to derive the DFA, which is the core of the rewriting procedure. We shows that our technique is correct by devising its proof of correctness.

2.1. *Writing the security policy*

The security administrator (SA) uses a Graphical User Interface (GUI) to specify for each user class (role), the part of information that the users are granted or denied access to. Indeed, in order to obtain a policy-dependent view of the schema, the SA annotates the schema using *security attributes*. This technique was first used in SMOQE [27].

We define the following security attributes: `access`, `condition` and `dirty`. Attribute `access` specifies the rights of the user on the node. The value of this attribute is either `allow` or `deny`. Attribute `condition` contains a list of predicates that have to evaluate to true for access to be granted. Attribute `dirty` indicates that some descendants of the current node could be unauthorized. More precisely, a node has a dirty attribute if it has at least one descendant node with either `access=deny` or a non empty `condition` attribute attached to it. Annotating the original schema means appending these attributes to element definitions in the schema. The annotated schema is no longer valid regarding W3C XML Schema recommendation. It is only an internal representation of the security policy that is never disclosed to the user.

Throughout the rest of this paper, we will consider a repository of XML documents valid w.r.t. the schema depicted in Fig.1(a) as a working example. In this example, we also consider user Alice and a policy that allows her access to element `showroom`, conditionally grants her access to elements `available` and `accessory` and denies access to `sold`. Alice is granted access to all other elements (except the descendants of `sold` of course). The annotated schema is depicted in Fig.1 (b), where security attributes are written in bold.

The remainder of the rewriting procedure, presented in the remaining subsections, consists of three steps:

Step 1: The annotated XML schema is transformed according to the policy that applies to each role. According to her role, the user is provided with the view of the schema (in short Sv) she is entitled to see. Then, she can write her query using information available on Sv. Henceforth, unless stated otherwise, the term view will refer to the view of the schema and not to the view of a source document.

Step 2: The annotated schema is translated into an automaton which represents the structure of Sv. Each state within Sv contains some security attributes that will further serve us while rewriting the user request.

Step 3: The user query is rewritten using the finite state automaton.

2.2. *Deriving the user view of the schema (Step 1)*

Deriving the user view from the annotated schema is straightforward. We start at the root of the annotated

```
ma xmlns="http://www.w3.org//2001/XMLSchema">
nent name="showroom">
ement name="vehicles" maxOccurs="unbounded" minOccurs="1" >
:element name="available"  maxOccurs="unbounded" >
  <element name="model" type="string"/>
  <element name="color"  type="string"/>
  <element name="price" type="string"/>
  <element name="accessory" maxOccurs="unbounded">
       <element name="description" type="string"/>
       <element name="price"  type="string"/>
  </element>
:/element>
:element name="sold"  maxOccurs="unbounded" >
  <element name="model" type="string"/>
                              ...
:/element>
ttribute name="city" type="string" use="required"/>
ment>
ema>
```

(a)

```
<schema xmlns="http://www.w3.org//2001/XMLSchema">
  <element name="showroom" access="allow"  dirty="true">
       <element name="vehicles" maxOccurs="unbounded" minOccurs="1"
                                     access="allow" dirty="true" >
            <element name="available"  maxOccurs="unbounded"
                              access="allow" dirty="true" condition="C">
                 <element name="model" type="string" access="allow"/>
                 <element name="color"  type="string" access="allow"/>
                 <element name="price" type="string" access="allow"/>
                 <element name="accessory" maxOccurs="unbounded"
                                    access="allow" condition="C1">
                      <element name="description" type="string" access="allow"/>
                      <element name="price"  type="string" access="allow"/>
                 </element>
            </element>
            <element name="sold"  maxOccurs="unbounded"  access="deny">
                 <element name="model" type="string"/>
                  ...
             </element>
     <attribute name="city" type="string" use="required"/>
  </element>
</schema>
```

(b)

Fig. 1. The Showroom Schema (a) and the corresponding annotated Schema (b)

schema tree, and at each element definition, we proceed as follows:

- If the attribute access is `allow` without any condition then we keep the node as is in the user view.
- If access is `allow` and there is an attribute `condition` set then we redefine the node as optional by adding the attribute `minOccurs=`0. In this way if a query gets to fail because the `condition` is not satisfied, then the querist would not infer the hiding of data.
- If access is `deny` then we discard the sub-tree rooted at the actual node from the user view.

The view for user Alice is depicted in Fig.2(a).

2.3. *Constructing the automaton (Step 2)*

Constructing the rewriting automaton from the annotated schema is also straightforward. The automaton *M* derived from the annotated schema consists of an alphabet Σ, a set of states S, a transition function $T : S \times \Sigma \rightarrow S$, a start state $s_0 \in S$, and a set of accepting states $A \subset S$. The automaton is constructed as follows:

The alphabet Σ consists of the values of the attributes `name` of each element definition on the annotated schema.

Creating the states: We start at the root of the annotated schema. The state corresponding to the root (element `schema`) is s_0. We create one state for each element definition which has a `dirty` parent. Indeed, all other nodes (those not `dirty`) and their subtrees are kept unchanged in the secured view. Hence they do not require to be processed by the automaton. When we encounter a denied node, we create a state for that element and skip the entire sub-tree rooted at that node. Each state $s \in S$ ($s \neq s_0$) has attributes which represent the security attributes stated at the corresponding element definition. We give to the state attributes the name and the value of their corresponding security attributes. Each state $s \in S$ ($s \neq s_0$) is a final state (i.e. $A = S \setminus \{s_0\}$).

Defining transitions: There exists a transition from a state s_i to a state s_j if the element definition corresponding to s_i is the parent of the element definition corresponding to s_j. The transition is labeled by the attribute `name` of the element definition corresponding to s_j.

The automaton derived from the annotated schema of Fig.1(b) is represented in Fig.2(b).

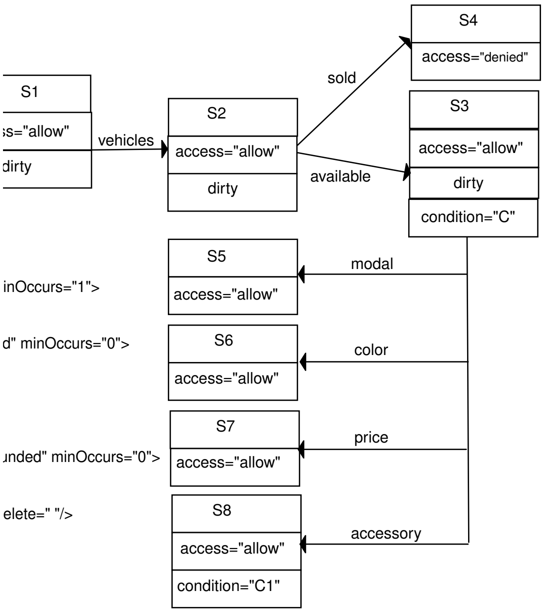
2.4. *Rewriting the request (Step 3)*

We assume that the user writes her request using the subset `XPath--`. [1] Hereby, we alleviate the rewriting process overhead since there is no need to backtrack in the automaton. We therefore rewrite the query in two phases. First, we refine the submitted expression and second, we rewrite the refined expression through the automaton.

Phase 1: refining the expression This step consists in refining the request on the basis of the view the user is permitted to see. We first transform the user query (over the repository) to an equivalent one (over the view). Second, we execute the latter on the user view (Sv) and from the target node we go back up to the root node,

---

[1] In [4] Gottlob, Koch and Picler show that the loss of expressive power of a fragment like `XPath--` w.r.t. XPath is minimal.



(b)

Fig. 2. The User Schema View (a) and the Rewriting FSA (b)

adding the encountered nodes on the path to form the *refined expression*. The goal of this procedure is to eliminate every `//` and `*` within the expression. As an example, if Alice request is `//vehicles/available` then the equivalent expression over the view is `//element[@name="vehicles"]/complexType/sequence/element[@name="availa-ble"]` and the refined expression is `/showroom/vehicles/available`. More examples are given in section 2.6.

Phase 2: Rewriting the request via the automaton The automaton represents the view the user is permitted to see. Rewriting the user request consists of,

- processing the first token[2] of the refined expression
- moving to the next state of the automaton until either the last token is received, or a clean state (i.e., a state that has no attribute dirty) is met or a denied state is encountered.

When processing a token, we consider the two following cases:

- Queries without predicates. After reading the current token, the automaton uses the attributes of the current state and behaves as follows: `Access` is `deny`. It rejects the request. `Access` is allow. There are two possibilities:

(1) If there is no attribute `dirty` then the user has the right to consult the entire sub-tree rooted at that node. The token is kept as such, the value of the attribute `condition` (if any) is attached to the token and the remainder of the source query is appended to the rewritten query. Note that the attribute `dirty` is for optimizing the rewriting procedure. Indeed, if the access is allow and if there is no attribute `dirty` then we do not need to analyze the remaining tokens one by one. We can directly append the remainder of the source query to the rewritten query.

(2) If there is the attribute `dirty` then the token is kept as such and if there is an attribute `condition`, its content is attached to the token. Then, the analyzer asks for the next token (if any).

If the last token has been fed into the automaton then we use the operator `except` to eliminate each unauthorized node under the target nodes. If q denotes the rewritten expression after the last token has been fed into the automaton then the final rewritten expression is $q' = q$ `except` $(e_1 \cup e_2 \cup ... \cup e_n)$, where each e_j with $1 \leq j \leq n$ is computed as follows:

The automaton consults one after another the states corresponding to the children of the node represented by the current state. At each state s corresponding to

---

[2] We call token a step in the path expression, for example `showroom` is the first token in `/showroom/vehicles/available`, while `vehicles` is the second. `/` stands for a `lookahead`.

the token l, we have the following:

If the attribute `access` = `deny` then l is appended to q. The result q/l becomes one of the e_j.

If the attribute `access=allow` and there is an attribute `condition` then the negation of the content C of the attribute `condition` is appended to l. The result $l[not(C)]$ is appended to q. $q/l[not(C)]$ becomes one of the e_j. If there is also an attribute `dirty` then the procedure goes deeper into the automaton (i.e. examines the children of the current token l) and starts computing another e_j with q now being equal to $q/l[C]$. Example Q_1 in section 2.6 illustrates this procedure.

2.5. *Queries with predicates.*

A query with a predicate is a query which contains a boolean expression. As instance, the expression `vehicles[condition]` selects the elements `vehicles`, children of the current node, that satisfy the `condition` (i.e. the `condition` evaluation returns `TRUE`).

In order to simplify the query rewriting process, in this work we consider only predicates that can be conjunctions or disjunctions of simple expressions *p* that belong to the set *P* defined as follows:

$P = \{p | p = [exp] \; or \; p = [exp \; op \; val]\}$, where $exp \in \{\epsilon, l, l_1/l_2\}$, $op \in \{=, != , <, \leq, >, \geq\}$ and val is the test value.

Evaluating predicates means to be sure that the user owns the authorization to see the information stored in the nodes that appear in the query predicate p.

The query rewriting process is divided into the following steps:

(i) we start saving the predicate expressions appearing in a query q in a *correspondence table*= $\{(l_i, p_i) | p_i$ is a predicate and l_i is the node on which the predicate is evaluated, $i = [1..n]\}$.

(ii) Then we follows the procedure detailed in the previous section to rewrite the query q without the removed predicate expressions. We obtain the rewritten query q'.

(iii) Finally we replace the predicate expressions in the rewritten query q' obtaining the query q'' and construct the automaton as detailed in the previous section. In this case, we stop processing the automaton when a token with predicate(s) is received. We save the current state and check whether the user has the right to consult the nodes that occur within the predicate(s). If she has the right to, we return to the saved state and continue with the next token. Otherwise the request is rejected.

As instance, let us consider the query q=`//vehicles/available[model="Fiat 500"]/accessory[price≤"150"]`. Following the first step of the procedure, we obtain the *correspondence table* shown in Figure 3. After the removal of predicates the query q is `//vehicles/available/accessory`, which is rewritten into q'=`/showroom/vehicles/available/accessory`. Then the predicates are inserted again and we obtain q''=`/showroom/vehicles/available[model="Fiat 500"]/accessory[price≤ "150"]`. Then, on the basis of query ”q”” we construct the automaton.

Node	**Predicate**
available	[model=”Fiat 500”]
accessory	[price$\leq$”150”]

Fig. 3. Example of correspondence table

2.6. *Rewriting Examples*

Let us now consider the following two queries posed by user Alice. Q_1:`//vehicles` and Q_2:`//vehicles/*` Both queries have to be refined. Q_1 is transformed to `//element[@name="vehicles"]` and executed over the view of user Alice. From the target node to the root, we encounter only the definition of element `showroom`. The request is then refined to Q_1': `/showroom/vehicles`. Likewise Q_2 is transformed to `//element[@name="vehicles"]/complexType/sequence/element`. The target nodes are the definitions of elements `available` and `sold`. Traversing the tree up to the root, we refine Q_2 as Q_2': `/showroom/vehicles/available union /showroom/vehicles/sold`. Then, we come to the second phase, i.e., rewriting the refined expression using the automaton. If the refined expression contains the operator `union` then we rewrite each component of the expression individually and combine the individual results with `union` to form the global outcome. Q_1': `/showroom/vehicles` is rewritten as follows: At state s_0, the automaton receives `showroom` and reaches state s_1. State s_1 indicates via its attributes that the privilege is `allow` and some descendants of `showroom` are inaccessible (attribute `dirty`). The output at this stage is `/showroom`. Since s_1 is `dirty`, the automaton reads the next token, that is `vehicles`, leading to state s_2. This state is allowed but some of its descendants are inaccessible (attribute `dirty`). We use the operator `except` to discard all unauthorized nodes. The final result is `/showroom/vehicles except (/showroom/vehicles/sold Union /showroom/vehicles/available[not(C)] Union /showroom/vehicles/available[C]/ accessory [not(C1)])` where C (resp. C1) is the condition expressed for the element `available` (resp. `accessory`) in the annotated schema (see Fig. 1 (b)).

2.7. *Correctness of our query rewriting method*

We show that our query rewriting method is correct by means of the loop invariant [3] technique.

Let us assume that the system receives a user query $xp \in$ `XPath--`. The first rewriting phase transforms xp to a refined query q which is a set of expression $q_i \in \zeta$ joined together with XPath operator `union` (i.e $q = q_1 \cup q_2 \cup ... \cup q_n$). Phase 2 rewrites in turn each q_i into q'_i.

Definition. We say that q'_i is correct with regard to q_i, if the result of executing q'_i over the repository is exactly the same as the answer to q_i if q_i were executed over the XML repository with access controls correctly enforced.

Let us assume that q_i contains n_i direct child axis (i.e $q_i = /l_1/l_2/.../l_{n_i}$). Let us call l_j; $j \leq n_i$ the `current` label being processed by the automaton. Let $q'_{i_{(j-1)}}=/l'_1/\ l'_2/.../l'_{j-1}$ (note that each l'_k with $k \leq j-1$ might include a predicate) be the rewritten query of $q_{i_{(j-1)}} = /l_1/l_2/.../l_{j-1}$.

We define the following loop invariant: $q'_{i_{(j-1)}}$ is correct with regard to $q_{i_{(j-1)}}$.

We must show that, this loop invariant holds prior to the first iteration of the second phase of the rewriting procedure, that each iteration maintains the invariant and that the invariant also holds when the procedure terminates.

- `Initialization`: The loop invariant holds before the first label is fed into the automaton. In fact, prior to the first iteration the rewritten query is $q'_{i_{(0)}} = \varepsilon$. This query is obviously correct with regard to $q_{i_{(0)}} = \varepsilon$.
- `Maintenance`: We show that each iteration maintains the loop invariant. Let us assume that the loop invariant is true before the label l_j is received. i.e $q'_{i_{(j-1)}}=/l'_1/\ l'_2/.../l'_{j-1}$ is the rewritten path of the sub-expression $/l_1/l_2/.../l_{j-1}$ and $q'_{i_{(j-1)}}$ is correct with regard to $q_{i_{(j-1)}}$. Let us assume that the current state of the automaton is s_{j-1}. According to the first phase of the rewriting procedure, there exists a transition from s_{j-1} on l_j. Let s_j be the state reached by that transition.

  Since $q'_{i_{(j-1)}}$ is correct regarding $q_{i_{(j-1)}}$, $q'_{i_{(j-1)}}$ returns the same set of nodes R as $q_{i_{(j-1)}}$ would do if it were executed over the XML repository with access controls correctly enforced.

  When the automaton receives l_j, it proceeds to the state s_j and consults s_j's attributes. If the attribute `access` is `allow` then the token is kept as it is. Content C of the attribute `condition` (if any) is then appended to it.

  $q_{i_{(j)}}$ would return all the child nodes l_j of the nodes belonging to R for which the user has an authorization. Now, the nodes l_j for which the user does not have an authorization are filtered out by the predicate C. Therefore, $q'_{i_{(j)}}$ returns the same set of nodes as $q_{i_{(j)}}$ would do. Hence, $q'_{i_{(j)}}=/l'_1/l'_2/.../l'_{j-1}/lj[C]$ is correct regarding $q_{i_{(j)}}=/l_1/l_2/.../l_j$. If there is no attribute condition, it simply means that the user has an authorization for all the child nodes l_j of the nodes belonging to R. In that case we also have $q'_{i_{(j)}}=/l'_1/l'_2/.../l'_{j-1}/l_j$ which is correct with regard to $q_{i_{(j)}}=/l_1/l_2/.../l_j$.
- `Termination`: The loop terminates in the following cases: (1) The loop meets a state where `access=allow` and there is no attribute `dirty`. Let s_j be that state ($j \leq n_i$). The (possibly empty) remaining path (i.e $/l_{j+1}/.../l_{n_i}$) is appended to the already rewritten expression (i.e $q'_{i_{(n_i)}}=/l'_1/l'_2/.../\ l'_{j-1}/l'_j/l_{j+1}.../l_{n_i}$). As shown in the maintenance step, $q'_{i_{(j-1)}}$ is correct with regard to $q_{i_{(j-1)}}$. The absence of the attribute `dirty` means that the user is allowed to access to the entire subtrees rooted at nodes l'_j addressed by $q'_{i_{(j)}}$. Thus $q'_{i_{(n_i)}}=q'_{i_{(j-1)}}/l'_j/l_{j+1}.../l_{n_i}=q'_i$ returns the same answer as $q_{i_{(n_i)}}=q_{i_{(j-1)}}/l_j/l_{j+1}/.../\ l_{n_i}=q_i$ would do if executed over the XML repository with access controls correctly enforced. Hence $q'_{i_{(n_i)}}=q'_i$ is correct with regard to $q_{i_{(n_i)}}=q_i$.

  (2) The loop meets a state where `access=deny`. let s_j be that state ($j \leq n_i$). The entire rewritten expression is replaced by the empty (ε) path.

  The fact that access is `deny` means that the user is forbidden to access any descendant node of the nodes addressed by $q'_{i_{(j-1)}}$. Therefore $q_{i_{(n_i)}}=/l_1/l_2/.../l_j/.../l_{n_i}$ would return the empty set if executed over a repository with access controls correctly enforced. Hence q'_i=$q'_{i_{(n_i)}}$=ε is correct with regard to $q_{i_{(n_i)}}$=q_i.

(3) The last token is fed into the automaton: If `access=allow` and there is the attribute `dirty` then it means that some sub-trees of the target nodes cannot be accessed to and, of course, $q_{i_{(n_i)}}$ executed over the XML repository with access controls correctly enforced would not return these sub-trees. Now, for such a case, the rewriting procedure we have defined in section 2.4 includes a supplementary step which is for filtering out these sub-trees by means of the operator `except`. Therefore we have $q'_{i_{(n_i)}}$=q'_i which is correct with regard to $q_{i_{(n_i)}}$=q_i. Else, termination meets either case (1) or (2).

By applying this proof for each q_i, we show that q' is correct with regard to $q = q_1 \cup q_2 \cup \ldots \cup q_n$.

2.8. *Complexity analysis*

The complexity of our approach is determined by that of steps 1, 2 and 3 of the rewriting procedure. Let us assume that the repository schema contains n definitions of element nodes. Deriving the user view of the schema (Section 2.2) takes at most $O(n)$ time. Constructing the automaton (Section 2.3) also requires at most $O(n)$ time as well. If m is the depth of the schema, then refining the expression (Section 2.4) takes $O(m)$ time. Since we rewrite the refined expressions by simply traversing the deterministic automaton, this phase takes $O(n)$ time. Hence, the overall time complexity of this proposal is $O(n + m)$.

3. Updating XML

Talking about XML security, not only the `read` privilege needs to be taken into consideration, but also the `write` privilege plays an important role. In this setting we do not have a proper standard for XML updates but refer to the XUpdate working draft. In our model, we consider the following write privileges: `DELETE`, `INSERT`, and `UPDATE`. The semantics of these privileges can be stated as follows:

- if user `s` holds the `INSERT` privilege on node `n` then user `s` has the right to add a new sub-tree to node `n`.
- if user `s` holds the `UPDATE` privilege on node `n` then user `s` has the right to update node `n` (i.e., change the values of its immediate children of type `text`).
- if user `s` holds the `DELETE` privilege on node `n` then user `s` has the right to delete the sub-tree of which node `n` is the root.

Below, for each XUpdate operation we list the write privilege that user `s` should hold.

Creating node operations There are three XUpdate instructions for creating XML fragments: `insert-before`, `insert-after` and `append`.

`Insert-before` inserts a given fragment as the preceding sibling of every context node, and `insert-after` inserts it as the following sibling of every context node. The operation `append` allows a node to be created and appended as a child of every context node.

- `insert-before/insert-after`: user `s` needs the `INSERT` privilege on the parent node of every context node.
- Operation `append`: user `s` needs the `INSERT` privilege on every context node.

Update operations There are two XUpdate instructions for updating XML nodes: `update` and `rename`. Operation `update` can be used to update the content of existing nodes. Operation `rename` allows an attribute or element node to be renamed after its creation.

- `update`: if context nodes are elements, then user `s` needs the `UPDATE` privilege on the content (text node) of every context node. If context nodes are attributes, then user `s` needs the `UPDATE` privilege on every context node.
- `rename`: user `s` needs the `UPDATE` privilege on every context node.

Renaming an attribute or updating its value requires the `UPDATE` privilege on the context node. This choice is consistent with the XPath data model, where an attribute node encapsulates both the attribute and its value. On the contrary, renaming an element requires the `UPDATE` privilege on the context node and updating its content requires the `UPDATE` privilege on the content node itself (i.e., the text child of the context node).

Delete operations There is one XUpdate instruction for deleting XML fragments: `remove`. Operation remove deletes all sub-trees having a context node as the root. For this operation, user `s` needs the `DELETE` privilege on every context node.

3.1. *Securing update operations*

Simply considering the write privileges held by a subject is not sufficient to make XML updates secure. The reason for this can be best understood by considering an analogy with SQL. Let us consider *user_A* who is the owner of an `Employee` database table and who has granted to *user_B* the `UPDATE` privilege on it. As a result, user_B is not permitted to see user_A's `Employee` table

```
SQL> SELECT * FROM user_A.employee;
ERROR ORA-01031: insufficient privilege
```

Fig. 4. automaton for updates (a) and write privilege annotated schema (b)

but user_B is permitted to update it:

```
SQL> UPDATE user_A.employee SET
salary=salary+100 WHERE salary > 3000;
2 rows updated
```

The simple example above shows that although user_B was not permitted to see user_A's employee table, she was able to learn, through an update command, that there were two employees with a salary greater than 3000. This is due to the fact that the `WHERE` clause did perform a `read` operation on `Employee` despite the fact that user_B did not hold the `SELECT` privilege on that table. In XUpdate operations the `select` attribute plays the same role as the `WHERE` clause in a SQL `UPDATE/DELETE` command. Therefore, in order to avoid the inference problem caused by write operations performing read action, we rewrite the XPath expression selecting context nodes according to the principles described in Section **??**. Securely controlling an XUpdate operation is then done in two steps.

(i) The XPath expression selecting the context nodes is rewritten according to the read privileges held by the user submitting the XUpdate operation. This step is described in section **??**. It corresponds to the work presented in [28] and uses the DFA for queries. However, when rewriting the XPath expression, we use the *answer-as-nodes* technique which stipulates that the XPath expression should return the target nodes rather than the entire subtrees rooted at them. Consequently, we spare the operation *except* that eliminates forbidden nodes within the sub-tree rooted at the target node.

(ii) The XUpdate operation should succeed for the context nodes on which user s holds the proper write privilege and fail for the others. In order to implement this principle, we rewrite a second time the XPath expression selecting the context nodes, so that only the nodes on which user s holds the proper write privilege are selected.

For rewriting the XPath expression according to the write privileges held by the user, we use the following technique: the policy author inserts for each user class (role), the authorization attributes in the XML Schema of the document collection creating the annotated schema. These attributes include *insert*, *update* and *delete*. The value of these attributes is either empty or equal to a list of predicates stating under which conditions the operation should be performed. A sample annotated schema is shown in Figure 4(b). The annotated schema is afterwards translated into a deterministic finite automaton for updates (see Figure 4(a)). The automaton traverses its states according to the tokens [3] of the rewritten expression produced by step 1, until the last token gets through. Then, the automaton transits to the state corresponding to the target node of the expression. At this position, the finite state machine behaves as follows [4] :

Case 1: The operation is `insert-before` or `insert-after`. The automaton backtracks to the previous state, which is the state corresponding to the parent of the context node. Indeed, the user needs the `INSERT` privilege on the parent node of every context node. If the attribute insert is present at that state then its (possibly empty) value is appended to the XPath expression and returned. If the attribute insert is not present then the expression is rejected.

Case 2: The operation is `rename` or `update`. If the attribute `update` is present then its (possibly empty) value is appended to the XPath expression and returned. If the attribute update is not present then the expression is rejected.

Case 3: The operation is `remove`. If the attribute `delete` is present then its (possibly empty) value is appended to the XPath expression and returned. If the attribute `delete` is not present then the expression is rejected.

Case 4: The operation is append. If the attribute `insert` is present then its (possibly empty) value is appended to the XPath expression and returned. If the attribute `insert` is not present then the expression is rejected.

It is worthwhile making a few observations about the operation `remove`. Let us consider a `remove` operation on a node `n`. When the user removes node `n` then she actually deletes the sub-tree rooted on node `n`. Some of the nodes which belong to that sub-tree may not be visible (i.e. the user may not be permitted to see them). Shall we reject the operation if some nodes of the deleted sub-tree do not belong to the user's view? On one hand, this would preserve the integrity of data the user is not permitted to see. On the other hand, it would reveal to the user the existence of data she is not entitled to see. In fact there is no definite answer to this question. This is typically a case of conflict between confidentiality and integrity. Here, we prefer to emphasize the confidentiality, and the command is accepted.

4. Tool description and experimental results

5. Related work

In the last few years, several XML access control models have been proposed (after the initial proposal appeared in [1], refinements were described in [2], [22], [18], [6], [13], [17]) which use access policies to compute secure views on XML data sets. These models addressed issues like granularity of access, access-control inheritance, overriding, and conflict resolution. All these proposals require provision for view materialization. Although views can be prepared offline, in general, view-based enforcement schemes suffer from high maintenance and storage costs, especially for a large XML repository.

A different approach has been explored in [21]. In a nutshell, [21] performs a static analysis that classifies a XML query to be either always-granted or always-denied before submitting it to an XML engine. For partially authorized XML queries, the solution in [21] relies on expensive run-time security checks to filter out the data nodes that users do not have authorizations to access. In [16], [23], the problem of unsafe query is solved by rewriting the input query based on the notion of security view. A security view is a restricted view of the document's DTD that exposes the schema structure the user is authorized to use when writing a XPath query. However, in [16], [23] there is no control of the query portion under the query target nodes, and forbidden nodes which are descendants of the target ones are disclosed to the requester. Our approach uses the XPath operator EXCEPT to filter out those conflicting portions from the input query.

QFilter [20] is an NFA-based query rewriting technique for XML. Authors in [20] constructs one NFA for each ACR and for each role. Thus this approach can be very inefficient for rewriting queries with ”//” axis because of the many backtrackings in the Automaton. This

---

[3] We call token a step in the path expression, for example `showroom` is the first token in `/showroom/vehicles/available`, while `vehicles` is the second. `/` stands for a `lookahead`.

[4] Here, for the sake of simplicity, we consider only commands and privileges addressing element nodes.

claim is confirmed by the complexity analysis done in [20] which shows that queries with ”//” and ”*” dramatically aggravates the access control overhead. Also when the input query has predicates in it, they are simply appended to the rewritten query and then can cause information leaks. Moreover, [7] shows that *QFilter* is not correct by deriving from [20] examples of incorrectly rewritten queries. On the contrary, our proposal uses a DFA-based technique which decreases the complexity of the rewriting procedure and always checks whether the user has the right to consult the nodes that occur within the predicates. We proved its correctness in section 2.7.

Authors in [19] argue that restricting access to relationships is as important as restricting access to nodes. To this aim, [19] introduced a *Security Specification Language for XML* (SSX). The SSX enforcement algorithm produces a security view schema for each user. XPath queries against security schemata are then rewritten according to the annotations attached to the annotated schema. The main drawback of this solution resides in the SSX language itself, which is based on schema manipulation primitives like `copy` or `delete` that appear to be unfit to large-scale access control policy specification. Experiments conducted in [19] show that on average, the approach has a performance which is quite similar to that of materialized views.

The approach proposed in [7] includes a two-phase filtering scheme: the first phase selects access control rules that are related to the user query. The second phase modifies an unsafe query into a safe one. This approach is interesting, but relies on underlying relational DBMS. Also the user is provided with the entire DTD and then can infer sensitive information. Our proposal overcomes these shortcomings by carefully computing the user view of the schema. Our DFA-based system is designed for any XML database. We propose the same technique for securely handling XUpdate commands, but due to space limitation we cannot include it in this paper.

Finally, we note that current standards for access control languages that can be used for protecting XML information ([29,30,**?**]) lack a standard technique for enforcing policies via secure query rewriting. For instance, XACML allows to write policies in XML stating access authorization to any type of resources, including XML data. However, XACML does not mandate any specific enforcement algorithm, but relies on different specifications of enforcement according to the protected resource type. Our DFA-based approach is general enough to provide a standard semantics for the enforcement of most XACML policies when applied to protect XML information.

6. Conclusion

In this paper, we describe a Deterministic Finite Automata (DFA) based approach to rewrite unsafe queries into safe ones, thus avoiding the many backtrackings inherent to NFAs. We highlighted how our approach improves w.r.t. previous works in the area. Also, we prove the correctness of the approach., and show that our technique is linear with the size and the depth of the repository schema. Although our rewriting procedure is theoretically efficient and suggests good performances, experiments remain work to be done. Moreover, our proposal leaves space for further work. Other inspiring approaches [14], [24] enforce client-based access control to XML. Indeed, in [14] and [24], the document is encrypted at the server side and decrypted at the client side. The input of their system is then XML data and the output is also XML data, while in our approach both the input and output is an XML query. We are investigating the possibility to diminish the workload at the server side by transferring the rewriting procedure at the client side. Finally, we are investigating interfacing our technique with standard policy languages like XACML. Our DFA-based approach is general enough to specify the enforcement of most XACML policies when applied to protect XML data. We plan to develop this topic in a future paper.

7. Acknowledgments

This work was supported in part by the Italian Basic Research Fund (FIRB) within the KIWI and MAPS projects, by the European Union within the PRIME Project in the FP6/IST Programme under contract IST-2002-507591 and by funding from the French ministry for research under ”ACI Sécurité Informatique 2003 - 2006. projet CASC”. Majirus Fansi holds a Ph.D scholarship granted by the ”Conseil Général des Landes”. The authors wish to thank Sabrina De Capitani di Vimercati, Pierangela Samarati and Stefano Paraboschi for common work and valuable suggestions.

References

[1] Damiani E., De Capitani di Vimercati S., Paraboschi S. Samarati P.: Securing XML Documents. In Proc. of the 2000 International Conference on Extending Database Technology (EDBT2000).

[2] Damiani E., De Capitani di Vimercati S., Paraboschi S., Samarati P.: A fine-grained access control system for XML

documents. In ACM Trans. Inf. Syst. Secur., Vol. 5(2). ACM Press, New York (2002) 169–202.

[3] Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.: Introduction to Algorithms. the MIT Press, 2003.

[4] Gottlob G., Koch C., Pichler R.: The Complexity of XPath Query Evaluation. In Proc. of the 22nd ACM SIGACT SIGMOD SIGART Symposium on Principles of Database Systems (PODS-02). ACM Press, San Diego (2003)179–190.

[5] Bray T., Paoli J., Sperberg-McQueen C. M.: eXtensible Markup Language (XML) 1.0 (2nd Ed). W3C Recommendation, 2000

[6] Stoica A., Farkas C.: Secure XML Views. In Proc. of the 16th IFIP WG11.3 Working Conference on Database and Application Security, 2002.

[7] Byun C. W., Park S.: An Efficient Yet Secure XML Access Control Enforcement by Safe and Correct Query Modification. In Proc. of the 17th International Conference on Database and Expert Systems Applications (DEXA), 2006.

[8] A. Laux and L. Martin. Xml update language (xupdate). *xml:db working draft, http://xmldb-org.sourceforge.net/xupdate*, 2000.

[9] E. Bruno, J. L. Matre, and E. Murisasco. Extending xquery with transformation operators. *In Proc. of the 2003 ACM Symposium on Document Engineering (DocEng 2003)*, 2003.

[10] D. Chamberlin, D. Florescu, and J. Robie. Xquery update facility. *W3C working draft*, May 2006.

[11] G. M. Sur, J. Hammer, and J. Simeon. updatex-an xquery-based language for processing updates in xml. *In Proc. of the 2004 International Workshop on Programming Language Technologies for XML (PLAN-XML)*, 2004.

[12] I. Tatarinov, Z. G. Yves, A. Y. Halevy, and D. S. Weld. Updating xml. *In Proc. of ACM SIGMOD*, 2001.

[13] Cuppens F., Cuppens-Boulahia N., Sans T.: Protection of relationships in xml documents with the xml-bb model. In Proc. of ICISS2005.

[14] Kodali N., Wijesekera D.: Regulating access to SMIL formatted pay-per-view movies. In Proc. of the 2002 ACM workshop on XML security.

[15] De Capitani di Vimercati S. and Marrara S. and Samarati P.: An access control for querying xml data. In Proc. of SWS05 workshop.

[16] Fan W. and Chan C. and Garofalakis M.: Secure XML Querying with security views. In Proc. of SIGMOD 2004 Conference.

[17] Finance B., Medjdoub S., Pucheral P.: The Case for access control on xml relationships. In Proc. of CIKM 2005.

[18] Gabillon A.: A formal access control model for XMl databases. In Proc. of the 2005 VLDB Workshop on Secure Data Management (SDM).

[19] Mohan S., Sengupta A., Wu Y., Klinginsmith J.: Access Control for XML - a dynamic query rewriting approach. In Pro c. of VLDB 2005 Conference.

[20] Luo B., Lee D., Lee W., Liu P.: QFilter: Fine-Grained run-time XML Access Control via NFA-based Query Rewriting. In Proc. of CIKM 2004.

[21] Murata M., Tozawa A., Kudo M.: XML Access Control using Static Analysis. In Proc. of CCS 2003.

[22] Kudo M., Hada S.: XML document security based on provisional authorization. In Proc. of ACM CCS 2000.

[23] Kuper G., Massaci F., Rassadko N.: Generalized xml security views. In Proc. of the 10th SACMAT, 2005.

[24] Bouganim L., Ngoc F. D., Pucheral P.: Client-Based Access Control Management for XML documents. In Proc. of the 30th VLDB Conference, 2004.

[25] Clark J., DeRose S.: XML Path Language (XPath). W3C Recommendation, 1999. http://www.w3.org/TR/xpath.

[26] Gabillon A., Bruno E.: Regulating Access to XML documents. In Proc. of the 15th Annual IFIP WG 11.3 Working Conference on Database Security, 2001.

[27] Fan W., Geerts F., Jia X. Kementsietsidis A.: SMOQE: A System for Providing Secure Access to XML. In Proc. of the 32nd VLDB Conference, 2006.

[28] E. Damiani, M. Fansi, A. Gabillon, and S. Marrara. A general approach to securely querying xml. *In Proc. of the 5th International Workshop on Security in Information Systems (WOSIS 2007)*, June, 2007.

[29] http://xml.coverpages.org/xacml.html

[30] http://xml.coverpages.org/xacml.html