A General Approach to Securely Querying XML

Ernesto Damiani¹, Majirus Fansi², Alban Gabillon² and Stefania Marrara¹

¹ Università degli Studi di Milano
Dipartimento di Tecnologie dell’Informazione
via Bramante 65 26013 Crema (CR), Italy

² 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 view 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 general query rewriting technique to securely querying XML. The model specification is given using a Finite State Automata, ensuring generality and easiness of standardization w.r.t. specific implementation techniques.

1 Introduction and Related Work

In the last few years, the Extensible Markup Language (XML)[2] 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., [6, 7, 14, 13, 17, 24, 5, 12]; for a complete survey, see [14]) 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, 19, 20, 10, 18, 9, 3]) 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 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 [4] to produce a sound and complete rewriting procedure (Section 2.4) of the user query.
- The complexity analysis (Section 2.5) 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.

A proof that our approach is sound and complete by means of a formal proof of correctness has been presented in [8]. Finally, section 3 concludes this paper and discusses future work.

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 show 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 [11].

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 $S_V$) she is entitled to see. Then, she can write her query using information available on $S_V$. 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 $S_V$. Each state within $S_V$ 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.

Fig. 1. The Showroom Schema (a) and the corresponding annotated Schema (b).
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 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 $\Sigma$, 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 $\Sigma$ 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--` of XPath expressions informally defined as follows:

\[
\text{XPath--} := \varepsilon | l | * | p_1/p_2 | \ell p_1 | p[q] \quad \text{where } p_1 \text{ and } p_2 \text{ are } \text{XPath-- expressions; } \varepsilon, l, * \text{ denote the empty path, a label and a wildcard, respectively; } / \text{ and } // \text{ stand for child-axis and descendant-or-self-axis; and finally, } q \text{ is called a qualifier. We rewrite the request in the subset } \zeta := \{\varepsilon | l[p_1/p_2[p[q]]\} \text{ of } \text{XPath-- using the functions union and except. } \zeta \text{ is } \text{XPath-- without descendant-or-self-axis } (\text{//}) \text{ and wildcards } (*). \text{ 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 \((S_v)\) and from the target node we go back up to the root node, 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="available"] and the refined expression is /showroom/vehicles/available.

**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,

---

3 In [15] Gottlob, Koch and Pichler show that the loss of expressive power of a fragment like `XPath--` w.r.t. XPath is minimal.
processing the first token\(^4\) 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. (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. (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 \( q' = q \text{ except } (e_1 \cup e_2 \cup ... \cup e_n) \), 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 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 \( l[\text{not}(C)] \) is appended to \( q \). \( q/l[\text{not}(C)] \) 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 \( q/l[C] \).

- Queries with predicates. The idea here is to 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.

Examples which illustrate the rewriting procedure are provided in [8]. Due to space limitation, we cannot include them in this paper.

\(^4\) 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.
2.5 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 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 (see [8]). Also, we 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 [16], [1] enforce client-based access control to XML. Indeed, in [16] and [1], 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 to the client side. Finally, Current standards for access control languages that can be used for protecting XML information ([23, 22, 25]) lack a standard technique for enforcing policies via secure query rewriting. We are investigating interfacing our technique with standard policy languages like XACML[23]. 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.

Acknowledgements

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