An XML Access Control Model Considering Update Operations

Meghdad Mirabi, Hamidah Ibrahim, Leila Fathi, Nur Izura Udzir, Ali Mamat
Department of Computer Science
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia, 43400
Serdang, Selangor, Malaysia
{izura, hamidah, ali}@fsktm.upm.edu.my, meghdad.mirabi@gmail.com, fathi_leila67@yahoo.com

ABSTRACT: *Several researches have been proposed over the past years to specify a selective access control for XML document. However, most of the previous researches only consider read privilege while specifying XML access control when access control policies support update rights is untouched. Therefore, a challenging issue is how to define an XML access control model to handle update operations that include insertion, deletion, and modification. In this study, we propose an XML access control model to support update privileges. Moreover, we formalize update operations supported by our proposed XML access control model based on the different default and conflict resolution policies which are applied to them.*

Keywords: Access Authorization, Access Control Policy, XML Access Control Model, XML Updating

Received: 12 March 2011, Revised 2 April 2011, Accepted 8 April 2011

1. Introduction

XML-based systems are now widely deployed in a number of application fields. This success has triggered a growing interest in XML security. Several applications at Internet and Intranet level require mechanisms to support a selective access to XML data available over the Web. In order to develop an access control system, access control rules first should be defined according to which accesses should be controlled and then some executable functions should be implemented to enforce these access control rules [1].

An access control model identifies who can access what and under which conditions and for executing which action. The first requirement to define an XML access control model is subject specification which specifies who can access to the portions of XML documents. In conventional DBMS systems, subjects to which an access control policy pertains are specified based on an identity-based mechanism. However, this approach is not sufficient and desirable in Internet. Actually, the subjects who desire to access the XML documents are very heterogeneous and highly dynamic. Therefore, the number and type of subjects are not known a priori and often changes in time. In this environment, identity-based access control approaches are not adequate to identify a huge number of subjects. In order to overcome this limitation, a good approach is to group the subjects which a policy pertains based on some criteria. In this case, a significant suggestion is RBAC (Role-based Access Control) which policies are specified for roles instead of single users. In this case, a collection of access rights needed to perform a particular action is assigned to roles and users are then assigned to roles. Thus, users can obtain authorizations to execute a particular action [1].

The second requirement to define an XML access control model is object specification which specifies the portions of XML documents to which a policy should be applied. The XML documents can have information at different degrees of sensitivity.

Therefore, the XML access control model has to support varying protection granularity levels, ranging from a set of documents, to a specific document, to a document portion. In this study, XPath [2] as a standard language is used in order to select a differentiated and selective protection of XML data to which a policy should be applied. Note that XPath language supports value based predicates and therefore it is possible to define conditions to specify the portions of the XML documents.

The third requirement of XML access control model is to identify the privilege that subjects can have on the protected objects. We can categorize privileges for XML data into two groups: browsing privileges and authoring privileges. In browsing privilege, subjects have permission to see information in an XML document or to navigate through its links while in authoring privilege, subjects have permission to modify XML document contents and/or XML document structure. An XML access control model must support both browsing and authoring privileges.

There have been a number of researches with different aspects to XML access control: to propose languages to specify access control policies [3-6], to devise mechanisms to enforce access control policies [7-18], and to suggest special data structure to optimize query processing [19, 20]. However, to our knowledge, formalizing the update operations through the access control model is not yet considered. Therefore, we study how to define an XML access control model to handle update operations that include insertion, deletion, and modification. Moreover, we define a formal representation of update operations supported by our proposed XML access control model based on different type of policies which are applied to them.

The rest of paper is organized as follows: related works are analyzed in Section 2; the proposed XML access control model is introduced in Section 3; and, finally the paper is concluded in Section 4.

2. Related Works

The proposed XML access control model in [17] can handle update operations for XQuery [3]. In order to distinguish the update operators semantically, the proposed model in [17] defines new action types which make it easy to manage information of users and access rights. Action types are a set of operators which are classified based on to what degree the operators change the XML documents that include operators which "do not change XML document", "change only the XML document", and "change the structure of XML documents".

XUpdate [6] is an XML based host language which supports update operations in the XML document. Due to its ease of understanding and simplicity of implementation, XUpdate is used to implement many XML based systems needed to support the process of XML document updating. A DFA (Deterministic Finite Automata) based approach is proposed by [7, 16] to support XUpdate operations over XML repository through XML views. The proposed approach rewrites XUpdate requests into a safe one.

A declarative access constraint specification (DACS) language is proposed by [4] to help DBAs to specify access constraints to be assigned to the XML documents. This language is able to reveal or hide XML nodes and block the structural relationship between a node and its sub-tree. Moreover, an algebraic security view specification language SSX is devised in [5] to hide or recognize XML nodes or XML sub-trees. The approach proposed by [11] simplifies the task of DBAs in specifying access constraints on an XML document. The formal algorithm proposed in [11] uses DACS language to generate possible XML views and then DACS primitives are converted to SSX. Therefore, all the possible security views are generated based on SSX primitives.

The idea of an XML security view is originally proposed in [9]. The security view only contains the information which the users are authorized to access. The access control model proposed by [9] provides an XML view for each user group as well as a DTD view which the XML view conforms to. In contrast to [9], the proposed models in [8, 10] consider general XML DTDs defined in terms of regulations rather than normalized DTDs. Furthermore, [8, 10] do not permit dummy element types in the definition of security views. However, the access control models proposed by [8-10] only support read privilege and specifying XML access control when access policies support update rights is untouched.

The main idea of static analysis proposed by [12] is to make automata for XML queries, XML access control policies, and XML schemas and then compare them. This method classifies an XML query at compile time into three categories: entirely authorized, entirely prohibited, or partially authorized ones. Entirely authorized or entirely prohibited queries can be executed

without access control. However, the static analysis cannot gain any benefits when a query is classified as a partially authorized. QFilter proposed by [13] checks XPath queries against access control policies and rewrite queries based on access control policies. Static analysis method needs runtime checking to filter out the unauthorized data while QFilter solves this problem by rewriting XPath queries to filter out unauthorized part of input queries before passing them to XML query engine. Static analysis and QFilter only support read privileges similar to [8-10].

In [14], a language similar to XPath [2] is employed to specify the objects. The write actions supported by the model proposed in [14] are: append and write. The append privilege allows a subject to modify the content of the XML element while the write privilege allows to modify the content of the XML element by deleting the node. In addition, the proposed model in [14] defines propagation options in order to determine whether an access control rule should be applied to an XML element or to an XML element and all of its descendants.

In [15], XPath expression [2] is employed in order to specify the objects. Access authorizations are specified at two levels: DTD level and document level and they can be local or recursive. A local authorization is propagated to an XML element and its attributes while a recursive authorization is propagated to an XML element and all of its descendants. The write actions supported by the model proposed in [15] are: insert, delete, and update. The accessibility of each XML element for write actions is based on a labeling algorithm. Labeling is the process of signing each XML element with "+" if it is accessible or "-" if it is not accessible.

In [18], an XML access control mechanism integrated with dynamic labeling scheme is proposed in order to accelerate the process of determining the accessibility of XML nodes and facilitate the process of updating the XML document. The key idea in the XML access control mechanism proposed by [18] is that the access authorization as a query condition to be satisfied. In this study, we formalize the XML access control model used in [18] in order to update well-formed XML document.

Constructing an entirely materialized accessibility map is a solution to determine whether a user may gain an access to a specific XML element or not, although such a process may need a huge amount of memory space. Compressed Accessibility Map (CAM) proposed by [19] compresses the entirely materialized accessibility map in order to reduce the need of large memory space to store it. In general, only a small portion of the entirely materialized accessibility map accounts for the total size of the CAM. Moreover, ICAM (Integrated CAM) proposed by [20] is an improvement over the original CAM which combines multiple CAMs into an ICAM and therefore it reduces the need of large memory space to store the CAMs.

3. XML Access Control Model

In the case of supporting update privileges by an XML access control model, subjects can have permission to insert a new node as a child of leaf nodes as well as a sibling or a parent node. Also, they can have privilege to delete leaf nodes as well as rename and update them.

The update operations supported by our proposed XML access control model are as follows:

- InsertChild (source, target)
- InsertBefore | After (source, target)
- InsertParent (source, target)
- Delete (target)
- Update (source, target)
- Rename (source, target)

InsertChild is an insert operation, in which source can be a PCDATA, an element or an attribute. InsertChild inserts source as the child of element denoted by target. If the XML document contains a sequence of information, InsertBefore and InsertAfter are employed. InsertBefore inserts source before element denoted by target, and InsertAfter does after element denoted by target. In addition, InsertParent inserts source as the parent node of element denoted by target. Delete is a delete operation, in which target can be a PCDATA, an element or an attribute. Update is an update operation, in which target can be an element or an attribute, and source can be a PCDATA. Rename is a rename operation, in which target can be an element or an attribute, and source is a new name.

We define an access authorization as 4-tuple <subject, object, action, permission> where subject is the user or role concerned by the authorization; Object is presented by XPath expression [2] which contains the element(s) of the XML document; Action is an executable action which can be InsertChild, InsertBefore, InsertAfter, InsertParent, Delete, Update, and Rename; Permission represents the acceptance (+) or denial (-) of rights.

In the following, the semantic of an access authorization <subject, object, action, permission> is explained informally for each update action supported by our proposed XML access control model. We consider only positive access authorizations for simplicity and negative access authorizations are left.

Let *T* be the well-formed XML document and *R* be the set of XML nodes returned from the evaluation of XPath expression of object on *T*.

- <subject, object, InsertChild, +>: subject can insert a new node as a child node of the nodes in *R*.
- <subject, object, InsertBefore/InsertAfter, +>: subject can insert a new node as a preceding/following sibling node of the nodes in *R*.
- <subject, object, InsertParent, +>: subject can insert a new node as a parent node of the nodes in *R*.
- <subject, object, Delete, +>: subject can delete the nodes in *R*.
- <subject, object, Update, +>: subject can update the content of nodes in *R*.
- <subject, object, Rename, +>: subject can rename the nodes in *R*.

Given an access control policy *ACP* which is a set of access authorizations and a well-formed XML document *T*, the semantic of *ACP* determines the XML elements of *T* to which a subject can apply a certain update operation.

Access control policy defined by the security administrator which contains a set of access authorizations must be complete and consistent. Access control policy is incomplete if no authorization is specified for an access to a specific object and it is inconsistent if there are both a negative and a positive authorizations for an access to a specific object [1].

Completeness can be easily obtained using one of either the open and closed policies as a default. The closed policy allows an access to a specific object if there is a positive authorization for it, and denies otherwise while the open policy denies an access to a specific object if there is a negative authorization for it, and allows otherwise. If no tough protection is needed, then the open policy can be applied by default. However, the closed policy is mostly applied to the systems to ensure an enhanced protection [1].

An authorization can be classified into explicit or implicit. An explicit authorization is explicitly specified on an XML element while an implicit authorization is implied by an explicit authorization which is specified on the nearest ancestor of XML element. Explicit positive/negative authorizations overriding the "*by default"* principle associated with propagation. Note that it is not needed to distinguish between explicit and implicit access authorizations since this distinction is already done by the XPath language [2] which is used to specify the objects. For example, the sequence of all descendants of XPath expression *p* is specified by *p*/descendant-or-self::*.

The problem of inconsistency is from potential conflict among access authorizations. The conflict resolution policy says that either a positive authorization overrides a conflicting negative one, or a negative authorization overrides a conflicting positive one. Several approaches have been suggested to solve the problem of conflicts between access authorizations such as deny overrides and grant overrides [1]. In deny overrides, negative access authorizations have priority over positive access authorizations while in grant overrides, positive access authorizations have priority over negative access authorizations.

In order to provide more flexibility in our XML access control model, authorizations are classified into two categories: strong and weak. A strong authorization does not permit an implicit authorization to be overridden while a weak authorization allows an explicit authorization to override an implicit authorization.

Figure. 1 shows the combination of *default policy* with *denies override* as *the conflict resolution policy* while Figure. 2 shows the combination of *default policy* with *grants override* as *the conflict resolution policy*. Here, *P* denotes the truth value of *has positive permission to carry out the update operation*, *N* denotes the truth value of *has negative permission to carry out the update operation*, *S* denotes the truth value of *strong authorization*, *W* denotes the truth value of *weak*

authorization, *A* denotes *accessible*, and *NA* denotes *inaccessible*. For instance, Figure. 1 says that the XML element *e* is accessible if and only if "a positive strong/weak authorization is defined for the XML element *e* and the XML element *e* is not in the scope of a negative authorization: [(*P*, *S* | *W*), (not *N*, -)]" OR "a positive strong authorization and a negative weak authorization are defined for the XML element *e*: [(*P*, *S*), (*N*, *W*)] " for *closed* as the *default policy*.

$[(P, S), (N, S)] \rightarrow NA/NA$
$[(P, S), (N, W)] \rightarrow A/A$
$[(P, W), (N, S)] \rightarrow NA/NA$
$[(P, W), (N, W)] \rightarrow NA/NA$
$[(\text{not } P, -), (\text{not } N, -)] \rightarrow A/NA$
$[(\text{not } P, -), (N, S)] \rightarrow NA/NA$
$[(\text{not } P, -), (N, W)] \rightarrow NA/NA$
$[(P, S), (\text{not } N, -)] \rightarrow A/A$
$[(P, W), (\text{not } N, -)] \rightarrow A/A$

Figure. 1 Open/closed as the default policy if deny is the conflict resolution policy

$[(P, S), (N, S)] \rightarrow A/A$
$[(P, S), (N, W)] \rightarrow A/A$
$[(P, W), (N, S)] \rightarrow NA/NA$
$[(P, W), (N, W)] \rightarrow A/A$
$[(\text{not } P, -), (\text{not } N, -)] \rightarrow A/NA$
$[(\text{not } P, -), (N, S)] \rightarrow NA/NA$
$[(\text{not } P, -), (N, W)] \rightarrow NA/NA$
$[(P, S), (\text{not } N, -)] \rightarrow A/A$
$[(P, W), (\text{not } N, -)] \rightarrow A/A$

Figure. 2 Open/closed as the default policy if grant is the conflict resolution policy

Now, we consider when an XML node is accessible for a specific update operation. Assume that P_a / N_a denotes a set of positive negative permissions for a specific action *a*; PS_a / NS_a denotes a set of positive/negative strong permissions for a specific action *a*; PW_a / NW_a denotes a set of positive/negative weak permissions for a specific action *a*. Therefore, we have:

• $P_a = PS_a \cup PW_a$: The set of positive permissions for a specific action *a* is the union of the set of positive strong permissions for a specific action *a* and the set of positive weak permissions for a specific action *a*.

• $N_a = NS_a \cup NW_a$: The set of negative permissions for a specific action *a* is the union of the set of negative strong permissions for a specific action *a* and the set of negative weak permissions for a specific action *a*.

Moreover, assume that *R* is a set of XML nodes obtained by evaluating the XPath expression of target parameter in update operation on a well-formed XML document *T* and ∨ denotes OR and ∧ denotes AND.

a) Open as the default policy and deny overrides as the conflict resolution policy

• InsertChild (source, target): "insert source as a child node of target node" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative InsertChild; *n* is in the scope of a negative weak InsertChild and a positive strong InsertChild. Formally the above conditions can be expressed as follows:

1. $[\wedge_{f \in N_{Insertchild}} not\ self :: f]$
2. $[\vee_{p \in PS_{Insertchild}} self :: p, \vee_{f \in NW_{Insertchild}} self :: f]$

• InsertBefore|After (source, target): "insert source as a preceding/following node of target node" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative InsertBefore|After; *n* is in the scope of a negative weak InsertBefore|After and a positive strong InsertBefore|After. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{InsertBefore}} not\ self :: f], [\Lambda_{f \in N_{InsertAfter}} not\ self :: f]$
2. $[\forall p \in PS_{InsertBefore} self :: p, \forall f \in NW_{InsertBefore} self :: f], [\forall p \in PS_{InsertAfter} self :: p, \forall f \in NW_{InsertAfter} self :: f]$

• InsertParent (source, target): "insert source as a parent node of target node" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative InsertParent; *n* is in the scope of a negative weak InsertParent and a positive strong InsertParent. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{InsertParent}} not\ self :: f]$
2. $[\forall p \in PS_{InsertParent} self :: p, \forall f \in NW_{InsertParent} self :: f]$

• Delete (target): "delete the target node*"* is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative Delete; *n* is in the scope of a negative weak Delete and a positive strong Delete. Formally the above conditions can be expressed as follows:

1. $[\Lambda f \in N_{Delete} not\ self :: f]$
2. $[\forall p \in PS_{Delete} self :: p, \forall f \in NW_{Delete} self :: f]$

• Update (source, target): "update the content of the target node with source" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative Update; *n* is in the scope of a negative weak Update and a positive strong Update. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{Update}} not\ self :: f]$
2. $[\forall p \in PS_{Update} self :: p, \forall f \in NW_{Update} self :: f]$

• Rename (source, target): "rename the target node with source" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative Rename; *n* is in the scope of a negative weak Rename and a positive strong Rename. Formally the above conditions can be expressed as follows:

1. $[\Lambda f \in N_{Rename} not\ self :: f]$
2. $[\forall p \in PS_{Rename} self :: p, \forall f \in NW_{Rename} self :: f]$

b) Closed as the default policy and deny overrides as the conflict resolution policy

• InsertChild (source, target): "insert source as a child node of target node" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative InsertChild but *n* is in the scope of a positive InsertChild; *n* is in the scope of a negative weak InsertChild and a positive strong InsertChild. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{Insertchild}} not\ self :: f, \forall p \in P_{Insertchild} self :: p]$
2. $[\forall p \in PS_{Insertchild} self :: p, \quad \forall f \in NW_{Insertchild} self :: f]$

• InsertBefore|After (source, target): "insert source as a preceding/following node of target node" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative InsertBefore|After but *n* is in the scope of a positive InsertBefore|After; *n* is in the scope of a negative weak InsertBefore|After and a positive strong InsertBefore|After. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{InsertBefore}} not\ self :: f, \forall p \in P_{InsertBefore} self :: p,], [\Lambda_{f \in N_{InsertAfter}} not\ self :: f, \forall p \in P_{InsertAfter} self :: p]$
2. $[\forall p \in PS_{InsertBefore} self :: p, \forall f \in NW_{InsertBefore} self :: f], [\forall p \in PS_{InsertAfter} self :: p, \forall f \in NW_{InsertAfter} self :: f]$

• InsertParent (source, target): "insert source as a parent node of target node" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative InsertParent but *n* is in the scope of a positive InsertParent; *n* is in the scope of a negative weak InsertParent and a positive strong InsertParent. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{InsertParent}} not\ self :: f, \vee_{p \in P_{InsertParent}} self :: p]$
2. $[\vee_{p \in PS_{InsertParent}} self :: p, \quad \vee_{f \in NW_{InsertParent}} self :: f]$

• Delete (target): "delete the target node*"* is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative Delete but *n* is in the scope of a positive Delete; *n* is in the scope of a negative weak Delete and a positive strong Delete. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{Delete}} not\ self :: f, \vee_{p \in P_{Delete}} self :: p]$
2. $[\vee_{p \in PS_{Delete}} self :: p, \quad \vee_{f \in NW_{Delete}} self :: f]$

• Update (source, target): "update the content of the target node with source" is granted to node *n* if: $n \in R$; *n* is not in the scope of a negative Update but *n* is in the scope of a positive Update; *n* is in the scope of a negative weak Update and a positive strong Update. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{Update}} not\ self :: f, \vee_{p \in P_{Update}} self :: p]$
2. $[\vee_{p \in PS_{Update}} self :: p, \quad \vee_{f \in NW_{Update}} self :: f]$

• Rename (source, target): "rename the target node with source" is granted to node *n* if: *n R*; *n* is not in the scope of a negative Rename but *n* is in the scope of a positive Rename; *n* is in the scope of a negative weak Rename and a positive strong Rename. Formally the above conditions can be expressed as follows:

1. $[\Lambda_{f \in N_{Rename}} not\ self :: f, \vee_{p \in P_{Rename}} self :: p]$
2. $[\vee_{p \in PS_{Rename}} self :: p, \quad \vee_{f \in NW_{Rename}} self :: f]$

In this way, we can also express formally the conditions in the case of *open* as *the default policy* and *grant overrides* as *the conflict resolution policy* and *closed* as *the default policy* and *grant overrides* as *the conflict resolution policy*.

4. Conclusion and Future Works

In this study, we proposed an XML access control model to handle update operations. Moreover, we defined a formal representation of update operations supported by our proposed XML access control model based on different default and conflict resolution policies which are applied to them.

Our proposed XML access control model can be used with different XML access control mechanisms for updating the well-formed XML document. Therefore, as a future study, we intend to apply our proposed XML access control model to different access control mechanisms to enforce access authorizations for updating the well-formed XML documents.

References

[1] Samarati, P., Vimercati, S.C.d. (2001). Access Control: Policies, Models, and Mechanisms. *In*: Foundations of Security Analysis and Design, LNCS 2171. p.137-196.

[2] Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J. (2010). XML Path Language (XPath) 2.0 (Second Edition). http://www.w3.org/TR/xpath20/.

[3] Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Siméon, J. (2007). XQuery 1.0: An XML Query Language. htrp://www.w3.org/TR/xquery/.

[4] Mohan, S., Wu, Y. (2006). IPAC: An Interactive Approach to Access Control for Semi-Structured Data. *In*: *Proceedings of the 32nd International Conference* on Very Large Data Bases. p.1147-1150.