XML Access Control with Policy Matching Tree

Naizhen Qi (Naishin Seki) and Michiharu Kudo

IBM Research, Tokyo Research Laboratory,
1623-14, Shimo-tsuruma, Yamato-shi,
Kanagawa 242-8502, Japan
{naishin, kudo}@jp.ibm.com

Abstract. XML documents are frequently used in applications such as business transactions and medical records involving sensitive information. Access control on the basis of data location or value in an XML document is therefore essential. However, current approaches to efficient access control over XML documents have suffered from scalability problems because they tend to work on individual documents. To resolve this problem, we proposed a table-based approach in [28]. However, [28] also imposed limitations on the expressiveness, and real-time access control updates were not supported. In this paper, we propose a novel approach to XML access control through a policy matching tree (PMT) which performs accessibility checks with an efficient matching algorithm, and is shared by all documents of the same document type. The expressiveness can be expanded and real-time updates are supported because of the PTM's flexible structure. Using synthetic and real data, we evaluate the performance and scalability to show it is efficient for checking accessibility for XML databases.

1 Introduction

XML [7] data is becoming more prevalent as more businesses and systems become integrated over the Web. In applications such as business transactions and medical records, sensitive data may be scattered throughout an XML document and access control at the node level (element or attribute) is required to ensure that sensitive data can only be accessed by authorized users. Access control must be expressive and be able to support rules that select data based on the location and value(s) of the data. In practice, the number of access control rules can be on the order of millions, which is a product of the number of document types (in 1,000's) and the number of user roles (in 100's). Therefore, the solution also requires high scalability and performance.

Several XML access control models [4,11,17,23] provide expressive access control over XML documents. These approaches usually support grant or denial access control specifications, a propagation mechanism whereby descendant elements inherit rules from their parents, and conflict resolution in case the data is covered by multiple access control rules. Since these models perform access control by traversing XML documents at runtime, the enforcement imposes heavy

S. De Capitani di Vimercati et al. (Eds.): ESORICS 2005, LNCS 3679, pp. 3–23, 2005.

computational costs especially for deeply layered XML documents with large and expressive access control rules.

Ideas to efficiently provide expressive access control have been proposed in [3,9,12,28,30]. These approaches are effective in efficiently searching for access controlled nodes [3,12,30], or in eliminating unnecessary accessibility checks at runtime [9]. These research efforts have managed to improve the efficiency of expressive access control. However, since they generally focus on document-based optimizations, XML databases with frequent updates of either the documents or access control rules may incur unacceptable costs. In our previous research [28], we proposed an efficient table-driven access control model that takes into account XML document updates. It provides runtime efficiency but has limitations on access control expressiveness and the real-time update of access control rules was not supported.

In this paper, we develop an effifffcient and expressive access control model applicable to existing access control models [4,11,23] for XML documents. The novelties of this access control model are a data-independent optimization so that XML data updates will not trigger any recomputations, and that real-time policy update is supported. The key idea is to build a policy matching tree, a PMT, on the basis of the access control rules. The accessibility check is performed by matching the access request against the PMT and deciding on the basis of the matching results. Since all of the rules in the PMT are isolated from each other, the PMT is capable of handling real-time PMT updates. An accessibility cache improves runtime performance by skipping duplicated accessibility evaluations on the same paths. Through experiments, we show the PMT is capable of supporting millions of access control rules efficiently.

The rest of this paper is organized as follows. After reviewing the concerned access control model in Section 2, we present our solution, the PMT model in Section 3. In Section 4 we describe how to match an access request against the PMT for an accessibility decision. Section 5 describes the access control system on the basis of the PMT. Experimental results are reported in Section 6 and in Section 7 we summarize our conclusions and consider future work.

1.1 Related Work

Many approaches for enforcing XML access control have been proposed. Some of them [17,23] support full [10] expressions to provide expressiveness with straightforward implementations by creating the projection of the access control policy on a DOM [19] tree. However, these approaches incur massive runtime costs when handling a large access control policy or a deeply layered XML document. The mechanisms proposed in [2,4,11,12] perform more efficiently but also encounter the same problem at runtime since node-level access control on a DOM-based view can be expensive when processing large numbers of XML documents.

To overcome this problem, several efficient access control models have been proposed [25,28]. Qi et al. [28], our previous research, presents a method that performs in near-constant time regardless of the number of access control rules. This is achieved by using an access condition table generated from the access control

rules independently of the XML data. However, this approach places limitations on the XPath expressions, and does not provide an efficient runtime evaluation mechanism for value-based conditions. Murata et al. [25] optimized the preprocessing steps by minimizing the number of runtime checks for determining the accessibility of nodes in a query with automata. However, the mechanism was limited to XPath-based languages such as XQuery [6], and cannot handle other query languages or primitive APIs such as DOM. XPath-based document filtering systems [1,8,14] also provide value-based access control enforcement and independence of the XML data through a precomputed data structure. However, these filtering systems focus more on data filtering rather than data selection. For example, they cannot specify denial access on document fragments in a grant subtree. Therefore, they are unable to completely satisfy the needs of real XML database access control applications.

A different approach with document-level optimizations was proposed by Yu et al. [30]. Their scheme enforces efficient access control with an accessibility map that is generated by compressing neighboring accessibility rules to improve the efficiency. However, since the maps are generated on the basis of the documents, document updates or policy updates may trigger expensive computations especially for a large XML ndatabase. In addition, the above efficient enforcement algorithms cannot support real-time updates on the access control rules.

Optimizations were also the focuse in a number of research efforts on XML query languages (e.g., XPath and XQuery). The methods include query optimization based on (i) the tree pattern of queries [9,13,27] (ii) XML data and XML schema [16,21,22,24]; and (iii) the consistency between integrity constraints and schemas [15]. However, these approaches usually perform efficient data selection at the level of documents and require indices. Therefore, in a large XML database, such as a database with 10,000 document collections and 10,000 documents for each document collection, such optimization mechanisms may consume a prohibitive amount of space. Moreover, these technologies are designed for XPath-based languages and they cannot handle other query languages and primitive APIs such as DOM.

2 Abstract of Access Control Policy

Various access control policy models have been proposed. We used the one proposed by Murata et al. [25] in which an access control policy contains a set of 3-tuple rules with the syntax[1] $(Subject, +/-Action, Object)$. The subject has a prefix indicating the type such as $userID$, $role$, or $group$. A user with a unique $userID$ may be in multiple groups, and the accessibility is decided on the basis of the accessibility results of the rules for the $userID$ and each for each $group$. A '+' stands for positive authorization (granted), while a '−' is for negative authorization (denied). Action can be $read$, $update$, $create$, and $delete$. A capitalized rule with $+Read$ or $-Read$ means that propagation is permitted and that the

[1] The syntax of the policy can be represented in XACML [26]. We use this form for simplicity.

access can be propagated downward to the entire subtree, while $+read$ has an effect only on the selected node. As an example, $(role : Doctor, +Read, /record)$ specifies a doctor's access to $/record$ is allowed and implicitly extended to the descendants. In addition, according to *denial downward consistency* as defined in [25] the descendants of an inaccessible node are also inaccessible, since there is an accessibility dependency between the ancestors and the descendants. Therefore, it is obvious that $-read$ and $-Read$ are equivalent to each other, and therefore we specify denial rules using only $-Read$ in this paper. We call the action permission ('+' or '−') together with the propagation permission the access effect. The object is the expression of the subset of XPath.

In addition, in order to maximize data security, we (i) resolve access conflicts with the *denial-takes-precedence* [25] rule and (ii) apply the default denial permission on the paths if no explicit or implicit access control is specified.

3 Policy Matching Tree (PMT) Model

The PMT model provides a fast matching mechanism to resolve the access control matching problem. The PMT searches for matched rules by matching the request against a tree that is the internal data structure of the access control policy. The PMT outputs matched target(s) upon which the accessibility, *grant* or *deny*, can be decided.

In this section, we first introduce the access control matching problem and then present the PMT model to represent the access control policy. Then, we present the matching algorithm.

3.1 The Access Control Matching Problem

A request defines the requestors' access requests to an XML data object or an XML instance by specifying property values. In the case of an XML instance, each path is individually checked against the access control rules. A matching element *ele* is a condition evaluation for requests. We say a request *req* matches a matching element *ele* if and only if $match(req, ele) = true$. In the access control matching problem, we are given an access request *req* and a finite rule set *Policy* which is translated into a set of matching elements. Subsequently, the goal is to determine all those rules in *Policy* that match *req*.

In our model, each request contains at least five properties: a $userID$, *role*, *group*, *action*, and *path*. The *group* is a list containing all of the groups the requester belonging to. The *path* is the simple path expression requested by the user. Besides these five properties, other properties such as current time and date may also be included. The rule set *Policy* is represented as an internal tree *Policy Matching Tree*.

3.2 PMT Components

The PMT model consists of four components: a node, edge, match target, and link. The node represents a property name of the access request. The edge represents a matching condition on the property name. The matching condition

Element	Symbol	Assignment	Example
Node		Property name	action
Edge		Matching condition linking two nodes, or a node to a match target	= 'update'
Match target		Matching result, and additional data	GRANT_ON_NODE
Link		Linking a match target to a node for further matching	-

Fig. 1. PMT model components

Table 1. Matching result of access effects for *read* action

Access effect	Matching result
+r	GRANT_ON_NODE
+R	GRANT_ON_SUBTREE
-R	DENY
Nothing	UNDECIDED

consists of an operator and the associated value. The edge ends at another node for further refinement or at a match target which is the leaf node representing the matching result. As occasion demands, the match target may also contain a property list, and a link for further matching. Fig. 1 depicts the PMT model components. The matching result in a match target is prepared in accordance with the access effect of the concerned rule. Table 1 shows the access effect for a *read* action with the corresponding matching results.

3.3 Property Matches of PMT

A PMT represents a set of access control rules. Each access control rule is a conjunction of property matches, where each property match represents a condition evaluation on the property name. Suppose an access control rule $rule : (Sub, +/- Act, Obj)$, in which if value-based predicate(s) is imposed, the predicate(s) are represented as $Pred$, and the object after removing $Pred$ is represented as Obj. To match a request req against $rule$, $match(req, rule)$ can be done as follows:

$match(req, rule) := match(req, Sub) \wedge match(req, Act) \wedge match(req, Obj) \wedge match(data\ value, Pred)$

Only when all of the property matches result in *true* is the rule matched and a matching result output. Each property match is represented by nodes and edges. The PMT is therefore constructed by adding or sharing nodes, edges, and match targets for the property matches of each rule. As a consequence, the policy is converted to a PMT consisting of four matching parts which match the request against Act, Sub, Obj, and $Pred$, respectively, as shown in Fig. 2. Since the access control rules are often imposed on the same object, in order to minimize the duplicated parts in the PMT, the $Pred$ matching is shared. For the same reason, the match target of the $Pred$ matching does not hold a matching result

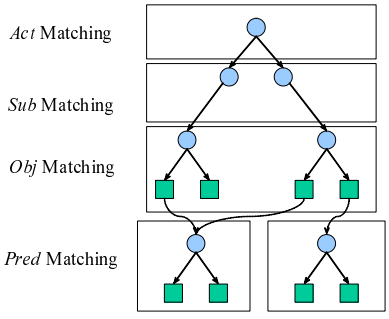

Fig. 2. A Sample PMT

specified for the rules. However, the *Pred* matching consumes extra computation owing to the data retrievals from the XML database. Therefore, to reduce the extra data retrievals, the *Pred* matching is started only when a match target is reached during the *Obj* matching no matter where the location where the predicates are imposed.

The matching result is only held by the match target of the *Obj* matching. If predicates are involved, then the match target of the *Obj* matching also provides a link for the *Pred* matching.

3.4 PMT Construction

Each access control rule is individually converted into three property matchings, or four in the case a predicate(s) is involved, with corresponding match targets.

***Act* Matching and *Sub* Matching Construction.** *Act* matching and *Sub* matching generation is simple in that "Act" and "Sub" are the nodes, and the values of *Act* and *Sub* specified by the rule are on the edges. As an example, we have an access control policy P1 as:

$R1 : (role : employee, +read, /Record)$
$R2 : (role : employee, +Read, /Record/Item[Key = \$userID])$
$R3 : (group : manager, +Read, /Record)$
$R4 : (group : manager, -Read, /Record//Info)$ In $P1$, $\$userID$ represents

a unique employee identifier. The corresponding *Act* matching and the *Sub* matching of $P1$ are generated as in Fig. 3. In the figure, two edges are coming from the *action* node, and the one to the role node is shared by $R1$ and $R2$. The other edge to the group node is for $R3$ and $R4$. Note the matching conditions on the edges from the action node are both $='read'$ in which the propagation permission is not included. In our PMT model, the propagation permission is carried by the *Matching Result* in the *Obj* matching.

***Obj* Matching Construction.** Here we consider the XPath expressions as compositions of five basic elements: $/x$, $/*$, $//x$, $//*$, and an $[x\ operator\ value]$

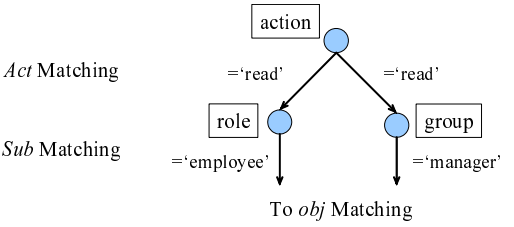

Fig. 3. *Act* matching and *Sub* matching of $P1$

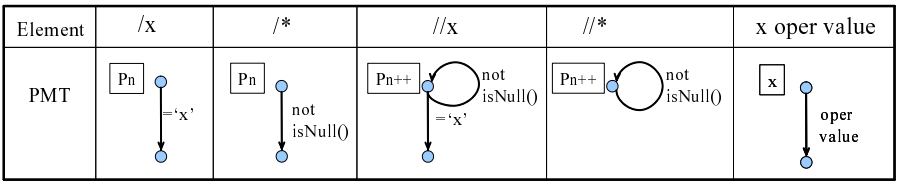

Fig. 4. Five basic elements to represent XPath expressions

where the last one represents a predicate and the operator can be any mathematical operator. Though other axes are not included as they would be in a general access control model, these elements are sufficient to select the concerned nodes.

The *Obj* matching is represented by the first four basic elements, while the *Pred* matching uses the last one. Fig. 4 shows how the five basic elements are represented.

P_n represents the node name of *Obj* at depth n. However, P_{n++} appearing in $//x$ and $//*$ automatically increases the depth counter variable for each loop. The corresponding *Obj* matching for $P1$ is shown in Fig. 5.

All of the match targets hold a matching result, which not only represents the access permission, but also represents the propagation permission to the descendants. For r, the positive authorization is not permitted to propagate downward, so a test on whether the requested path is the descendant of the object is performed by $isNull()$ as $R1$ shows. Beyond the normal matching result, the match target of $R2$ also contains a property list and a link to further refinement on $/Record/Item$ through a predicate.

The property list contains pairs of the property name and the data name appearing in the predicate where the property name is used in the *Pred* matching in place of the data name. The data name is more than a single element node name or attribute name, since the relative path from the object to the node being appended with the predicate is also included. For example, when it is $/Record/Item[Key = \$empID]/Priority$, the property list is $pred0 =$ "$../Key$", since to traverse from *Priority* to *Key*, we must go up to *Item* first, which is represented by '../'. To retrieve XML data from the database for predicate evaluation, the data name is passed to the data retrieval processor at runtime. Since the runtime has knowledge of the accessed node location with both the matched

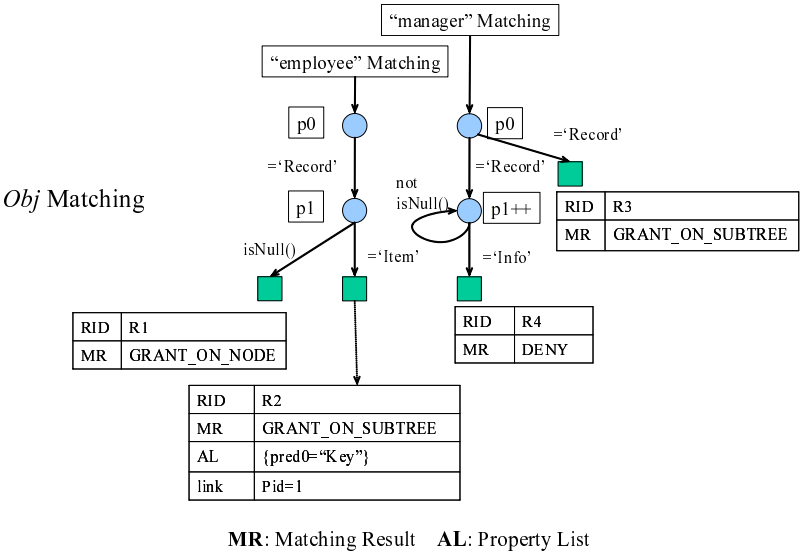

Fig. 5. Obj matching of P1

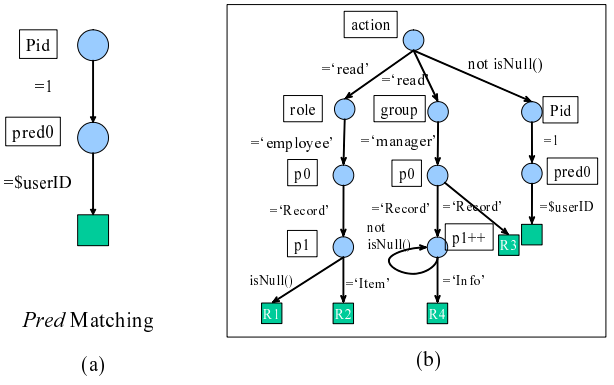

Fig. 6. Pred matching and the entire PMT

node location and the relative path from the matched node, the location of the required data can be found.

***Pred* Matching Construction.** A predicate is a condition for comparison to the XML data or a conjunction of such conditions. The condition is directly converted to the *Pred* matching. By separating the Pred matching from the *Obj* matching, the predicate evaluation is optimized in the way: 1) unless the match on the *Obj* matching reaches a match target, the match for the *Pred* matching is not started; 2) multiple predicates imposed on the same object are processed in a single *Pred* matching operation; and 3) the *Pred* matching is shared by the set of access control rules.

The *Pred* matching of $P1$ is shown in Fig. 6(a) and Fig. 6(b) gives the entire PMT for $P1$. Beyond the condition matching for $[pred0 = \$userID]$, the link which connects the *Obj* matching with the *Pred* matching is also shown in the figure. Owing to the link ID *Pid*, the *Pred* matching can be shared by multiple rules, if the same predicate is involved.

However, when a predicate *Pred* appears after with //, since multiple requested paths match the object owing to //, the property list for *Pred* cannot be specified in the match target during PMT generation without knowledge of the requested path. Therefore, in the current PMT model, a limitation is placed on the expressions that predicates never come after //. As a consequence, XPath expressions such as $/Record//Info[@type =' classified']$, and $//*[@type =' classified']$ are not supported.

3.5 Complexity Analysis

PMT Generation Time Complexity. For each access control rule that we need to add to the PMT, we spend generation time proportional to the number A of properties, where the properties are the action, the subject, the node tests appearing in the object, or the nodes involved if there is a predicate(s). Therefore, if there are N access control rules, the total time spent on PMT generation is $O(AN)$. If the maximum depth of the object is D, and the maximum number of conditions in a predicate is P, A is no bigger than (3+D+P+2), where that 3 is from the *Act* matching, the *Sub* matching, and a link from the *Obj* matching to the *Pred* matching, and the 2 is for the match targets of the *Obj* and *Pred* matchings. Therefore, A is no larger than (5+D+P), which are integers, and hence the PMT generation time is linear with respect to the number of access control rules.

PMT Space Complexity. For the space complexity, note that each rule can add at most (5+D+P) nodes to the PMT. Thus, the largest space required for the PMT is $O((5 + D + P)N)$, that is, linear in the number of access control rules.

4 PMT Matching

Our approach resolves the access control matching problem by matching the request against the PMT to determine the access effects of the rules that match the request. In this section, we present the matching algorithm, and then analyze the complexity.

4.1 PMT Matching Algorithm

The input of the PMT is an access request that contains at least five properties: a userID, role, group, action, and path. However, the path cannot be input directly into the PMT, since the PMT cannot match the path against the nodes

and edges. Therefore, preprocessing is required to convert the path into a set of node tests. The output of the PMT is the set of matching results as shown in Table 1. By combining the matching results and resolving any access conflicts, the accessibility decision, *grant* or *deny*, is decided. The PMT matching algorithm is given in Fig. 7. The idea is to walk the PMT from the root node by performing the matching prescribed by each node and following the edge that satisfies the condition for that node. The set of matching results are the match targets that are visited. This algorithm is independent of the traversal ordering on the PMT. We can traverse the tree in a depth-first order, but it is clear that other orderings, such as breadth-first, would also work.

As an example matching $M1$, we demonstrate how to decide the accessibility using the PMT in Fig. 6(b). Suppose the access request req is {*userID:'T29595', group:'manager', action:'read', path:'/Record/Item/Address'*}. Path is further processed to a property list {*p0:'Record', p1:'Item', p2:'Address'*}. As a result, the pairs of name and value input into the PMT eventually becomes {*userID:'T29595', group:'manager', action:'read', p0:'Record', p1:'Item', p2:'Address'*}.

The input data traces two routes in the PMT. One is stopped halfway before reaching a match target, and the other one reaches the match target of $R3$ as:

$$action(='read') \rightarrow group(='manager') \rightarrow p0(='Record') \rightarrow p1(='Item') \rightarrow p2(='Address') \rightarrow p3$$
$$action(='read') \rightarrow group(='manager') \rightarrow p0(='Record') \rightarrow R3's\ match\ target$$

Since the matching result of $R3$ is *GRANT_ON_SUBTREE*, the accessibility is therefore decided as *grant*, and the manager's access to $/Record/Item/Address$ is permitted. Suppose one more PMT input is {*userID:'T29595', role:'employee', group:null, action:'read', p0:'Record', p1:'Item', p2:'Address'*}. The match target of $R2$ is reached by the route:

Input The pair of property name *n* and value *v* of the access request

Step 1 If *v* of *n* satisfies the condition on the edge from *n*, then follow the edge.
If no matched *n*, then return *UNDECIDED*.

Step 2 If the node below the edge is node *n*, then repeat Step 1.

Step 3 If the node below the edge is a match target, then
- if it contains a link, then retrieve the value, repeat Step 1 with the data value and the access request.
- if it contains *GRANT_ON_NODE* or *GRANT_ON_SUBTREE*, return *GRANT*.
- else it is the match target of *Pred* matching, returns *true* to *Obj* matching.
  If the matching result of *Obj* matching is *GRANT_ON_NODE* or *GRANT_ON_SUBTREE*, then return *GRANT*. Otherwise, return *UNDECIDED*.

Output The set of returned matching results.
- if *DENY* exists, the decision is *DENY*.
- else if all of the matching results are *UNDECIDED*, the decision is *DENY*.
- otherwise, the decision is *GRANT*.

Fig. 7. PMT Matching Algorithm

$action(='read') \rightarrow role(='employee') \rightarrow p0(='Record') \rightarrow p1(='Item') \rightarrow R2's\ match\ target$

In this case, beyond the matching result, a property list $pid0 ='Key'$ and a link $Pid = 1$ are also included. Therefore, the value of the required data 'Key' is retrieved and the $Pred$ matching is performed. The PMT input is updated to {*userID:'T29595', role:'employee', group:null, action:'read', Pid:1, pred0:'T29590'*}. The $Pred$ matching becomes:

$action(\neq isNull()) \rightarrow Pid(=1) \rightarrow pred0(\neq \$userID)$

The match target is not reachable in the $Pred$ matching, and hence $false$ is returned. With the matching results in $GRANT\_ON\_SUBTREE$ from the Obj matching and $false$ from the Pred matching, the accessibility decision results in *deny*, and the access to $/Record/Item/Address$ is denied.

4.2 Matching Time Complexity

We measured the access request matching times by counting the number of PMT nodes that are visited during the match. In any reasonable implementation of the matching algorithm, this number is proportional to the actual time necessary to match the access request, since the algorithm performs a simple matching for each node, which is assumed to take constant time.

The matching time is a function of the access request being matched, since different access requests cause different sets of nodes to be visited during matching even if the set of access control rules is constant.

In the rest of this section, we compute the expected time to match an access request, and show the expected time is sub-linear with respect to the number of access control rules. We assume that all properties range over the same set of values for simplicity. Henceforth, let K be the number of properties of a rule; V be the number of possible values for a property; P be an arbitrary set of rules; $PMT(P)$ be the PMT generated for P; and $T(P)$ the expected time to match a request.

Theorem 1. *Given that all access requests are equally likely, then the expected time $T(P)$ to match a random access request is bounded by*

$$T(P) \leq 2(K+1)|P|^{1-\lambda}(lnV + ln(K+1)) \text{ where } \lambda := \frac{lnV}{lnV + ln(K+1)}.$$

For each node n in $PMT(P)$, we define $cost(n)$ to be the number of times that this node is visited when we run the matching algorithm with all the possible VK requests. The probability that a node n is visited when matching a random request is therefore equal to $V^{-K}cost(n)$. Thus, the expected number $T(P)$ of nodes of $PMT(P)$ visited is:

$$T(P) = V^{-K} \sum_{n \in nodes(PMT(P))} cost(n) \tag{1}$$

where $nodes(PMT(P))$ is the set of nodes of $PMT(P)$.

Lemma 2. *For $0 \leq j \leq K$, $PMT(P)$ contains at most $(K+1)((K+1)V)^j$ nodes with cost V^{K-j}.*

Proof. A node n has cost V^{K-j} if and only if the route from the root to the node has exactly j non-* edges. The number of edges i is between j and K; the positions of the non-* edges are j distinct numbers between the root and n, and so there are $\sum_{i=j\ldots K} \binom{i}{j} = \binom{K+1}{j+1}$ ways of choosing routes without involving *-edges. Moreover, we can assign V distinct values for each non-* edge. Therefore, the number of paths in PMT(P) with exactly j non-* edges is at most $V^j \binom{K+1}{j+1}$. Moreover,

$$V^j \binom{K+1}{j+1} \leq V^j \frac{(K+1)^{j+1}}{(j+1)!} \leq (K+1)((K+1)V)^j. \tag{2}$$

Lemma 3. *$PMT(P)$ has at most $(K+1)|P|$ nodes.*

Proof. An access control rule is associated with subject, action, and object, leading to K edges and $K+1$ nodes including the match target. Thus, if the policy has $|P|$ rules, it has at most $(K+1)|P|$ nodes.

Let $f(i)$ be cost of the i-th node in order, by using Equation (1) and Lemma 3, we have that $T(P) = V^{-K} \sum_{i=1}^{(K+1)|P|} f(i)$.

Let $g(x) := (Ax+B)^{-\lambda}$ where $A := V^{-(K+1)/\lambda}(V - 1/(K+1)), B := V^{-(K+1)/\lambda}$, and $\lambda := \frac{lnV}{lnV+ln(K+1)} < 1$.

Lemma 4. *$f(x) \leq g(x)$.*

Proof. By Equation (2) and the definition of $f(x)$, we have that for each i such that $0 \leq i \leq K$ and for each j such that $\sum_{p=0\ldots i-1}(K+1)((K+1)V)^p < j \leq \sum_{p=0\ldots i}(K+1)((K+1)V)^p$, the following holds: $f(j) \leq V^{K-i}$.

Now, $g\left(\sum_{p=0\ldots i}(K+1)((K+1)V)^P\right) = g\left(K\frac{((K+1)V)^{i+1}-1}{(K+1)V-1}\right)$. By using the definition of $g(x)$, we conclude $g\left(\sum_{p=0\ldots i}(K+1)((K+1)V)^P\right) = V^{K-i}$.

Proof (Proof of Theorem 1). We have that $T(P) = V^{-K}\sum_{x=1}^{K|P|} f(x) \leq V^{-K}\sum_{x=1}^{K|P|} g(x) \leq V^{-K}\int_0^{K|P|} g(x)dx = V^{-K}\frac{(AK|P|+B)^{1-\lambda}-B^{1-\lambda}}{A(1-\lambda)}$. By replacing the values of A and B and simplifying, we obtain

$T(P) \leq \frac{V(K+1)((V(K+1)|P| - |P|+1)^{1-\lambda}-1)}{(V(K+1)-1)(1-\lambda)}$. After using $V(K+1)|P| - |P| + 1 \leq V(K+1)|P|$ and $(V(K+1))^{1-\lambda} = K+1$, and after replacing the value of λ, we obtain

$$T(P) \leq \frac{V(K+1)((K+1)|P|^{1-\lambda}-1)(lnV+ln(K+1))}{(V(K+1)-1)ln(K+1)}. \tag{3}$$

Since $V \geq 2$ and $K+1 \geq 2$, we have $V(K+1)/(V(K+1)-1) \leq 4/3$ and $1/ln(K+1) < 3/2$. By introducing thees values in (3), we obtain $T(P) \leq 2(K+1)|P|^{1-\lambda}(lnV + ln(K+1))$.

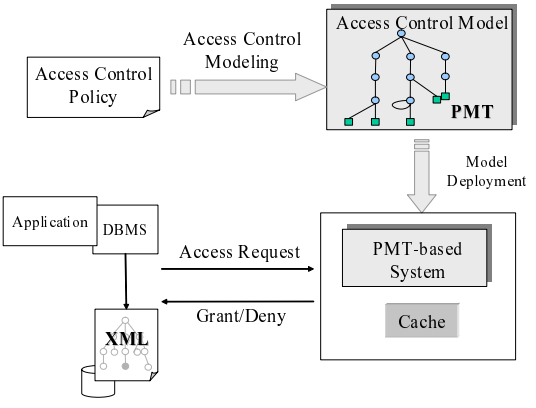

Fig. 8. PMT-based Access Control System

5 PMT Access Control System

In this section, we present the construction of the access control system. Then we show a cache optimization to improve performance by skipping the PMT matching and accessibility evaluation. Lastly, we describe the runtime policy update mechanism.

5.1 Access Control System

The proposed access control system is constructed through Access Control Modeling and Model Deployment shown in Fig. 8. In Access Control Modeling, the access control policy is converted to an internal data structure, a PMT, so that each rule is represented by a set of nodes and by edges with one or two match targets. In Model Deployment, an empty cache table is prepared.

At runtime, given an access request from an application or a DBMS, the access control system runs an evaluation for the accessibility decision. The system may add a new entry to the cache, as long as the corresponding entry is not found in the cache and the accessibility decision is uniquely decided owing to independency of the data values. Otherwise, the previously cached value is returned directly without PMT matching.

This system structure separates the access control system from the database engine so that security-related support is not required from the underlying database. In addition, it enables any DBMS to offer access control even if it is an off-the-shelf product.

5.2 Optimization with an Accessibility Cache

Since the accessibility status of a path remains unchanged even if the XML data changes as long as the access control does not involve any value-based evaluations, the result of an accessibility evaluation can be cached, thereby improving runtime performance. This is particularly beneficial if a path appears multiple

times in an XML document(s), or in a document type, because obtaining the accessibility result by looking at the accessibility cache allows the system to avoid repeating accessibility evaluations.

The accessibility cache, generated individually for each action type, is a table of 3-tuples: *subject*, *path*, and *accessibility decision*. During an evaluation with no predicate, an entry will be added to the accessibility cache table. Otherwise, when the accessibility depends on the data values in the XML document, the accessibility results may be different with different subtrees and different documents, and therefore, the accessibility evaluation involving predicates must be performed at every access.

We show an example using the access control policy $P1$. The initial accessibility cache table is empty. When $M1$ as presented in Sect. 4.1 finishes, we obtain an accessibility cache table for *read* action which contains 1 entry:

{*group:'manager', /Record/Item/Address, grant*}.

There is only one entry in the cache table since the second access request was evaluated on the basis of an XML data value. Meanwhile, though the first access request contains both userID and group data, the subject of the corresponding cache entry only holds the group. This is because for PMT matching, the userID does not affect the accessibility decision that a manager's access to */Record/Item/Address* is always granted without regard to the userID. If any manager accesses the same path, */Record/Item/Address*, in another XML document, the *grant* is looked up and directly returned as the accessibility decision.

When a user is bound to multiple groups such that the accessibility is decided by multiple matching results, rather than generate a single cache entry, it is efficient to generate one entry for each group. The cache entries can easily be reused by other requesters in the same group(s). However, for rules involving a userID or a role, the number of entries inserted into the accessibility cache table is limited to one.

To generate a proper accessibility cache entry, the *subject* information that decides the accessibility is required. Howevner, without recording each visited path in the PMT, there is no way to know the corresponding matched subject. This calls for extra computation to obtain the matched object information with the current PMT structure, especially when multiple match targets for userID and groups are visited. To resolve this problem, we enhance the match target of the *Obj* matching by retaining the subject of the access control rule. Because each access control rule is bound to an individual match target, carrying the subject along with the match target can meet the requirements for accessibility cache generation.

5.3 Access Control Policy Update

In some real applications, the access control policy may be updated at runtime. For instance, if a new role is introduced into the running system, then the corresponding access control rules should be added to the system as well. In our previous work Qi et al.[28], real time policy update was not supported, since

multiple access control conditions were combined into an access condition expression. In the PMT model, it is possible to perform real time updates on the PMT when the policy is updated during runtime, since each rule has a corresponding match target which is distinguished by a unique rule ID RID.

The RID in the match target plays a crucial role in runtime rule removal. The match target, nodes and edges are removed bottom-up if the components are not shared by other rules. When a rule is updated, the PMT reacts by removing and adding the corresponding components without changing the PID in the match target. It costs $O(m(2+D+1))$ time to add, remove, or update the rules, where m is the number of new rules, 2 is from the Act matching, and the Sub matching, D is the maximum depth of the objects, and 1 is for the match target.

6 Experiments

In this section, we describe our experiments to evaluate the performance of our PMT-based access control mechanism for XML documents. All of the experiments were conducted on a machine with a 1.8GHz Pentium 4 CPU, 1.5GB of main memory, and IBM JDK1.4.2. We discuss the experimental data in Sect. 6.1 and present the results in Sect. 6.2.

6.1 Experimental Data

To demonstrate the scalability of the system, we examined the memory requirements when a large access control policy is loaded into main memory, and the access control processing times when a large XML document is processed. To show the expressiveness of the access control specification, we ran experiments involving predicates and we collected the update performance. In addition, we show the performance gains achieved with the accessibility cache and evaluation-skipping mechanisms.

We use two XML document types in our experiments. The first one, *Orders*, is a real data set describing business transactions, and we prepared two different documents of that type, with sizes of 100 KB and 4 MB. The other type is the XMark benchmark data, where we used a 111MB *standard.xml* file. Both document types contain subtrees with similar structures occurring repeatedly at the same level.

6.2 Results

Scalability for Large Access Control Policies. In practice, the number of access control rules is on the order of millions, which is a product of the number of document types (in 1000's) and the number of user roles (in 100's). The main purpose of this experiment is to see whether or not the PMT model can support large access control policies. For simplicity, we specified 760,000 access control rules for 30,400 users for the *Orders* document type. Each user was associated with a set of 25 access control rules specified with simple path expressions and $+r$. Fig. 9(a) shows the rule set for User1, and Fig. 9(b) shows

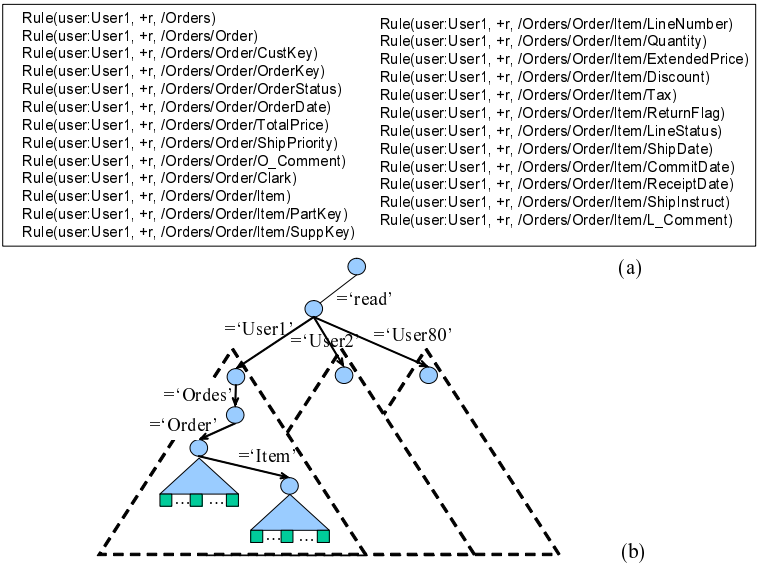

Fig. 9. Sample Access control rules and the corresponding PMT

the corresponding PTM image, showing that the subtree structure for the *Sub* matching is actually identical. The experiment shows that 760,000 access control rules use almost 640 MB implying each rule takes about 0.84KB on average.

Scalability for Large XML Documents. In many access control systems, it is necessary to support access control for large XML documents. For example, XML-formatted documents for record retention may be several megabytes in size. In this experiment, we show the performance of the system by examining the total processing time when the XML documents are 100 KB and 4 MB.

For each subject, we specified 25 access control rules such as shown in Fig. 9(a) for *Orders.xml*. All rules specified a $+r$ permission. Both documents contain repeated sub-structures under */Orders/Order* and so most parts of the access control are duplicated at multiple locations. We used the SAX API of the XML parser to parse the entire document, and checked the accessibility when encountering either an element or an attribute. The processing time includes the XML parsing time, the access control time, and the garbage collection time if it occured. In this experiment, we defined the total time excluding the parsing time as the AC Time. We also measured the performance improvements achieved with caching. In Fig. 10, the processing times of the full documents are shown.

From the bars, it is clear that the accessibility cache makes a significant improvement in processing times. For the 100 KB document, the accessibility cache reduces the AC time by almost 51%. For the 4 MB document, the AC time is reduced to 21%. Since 4MB document contains more duplicated *Order* subtrees, it derives more benefit from the accessibility cache.

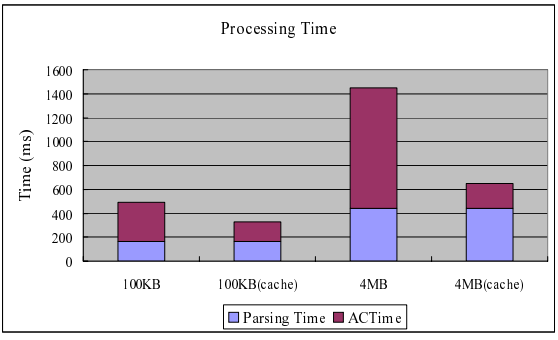

Fig. 10. Processing time for entire documents

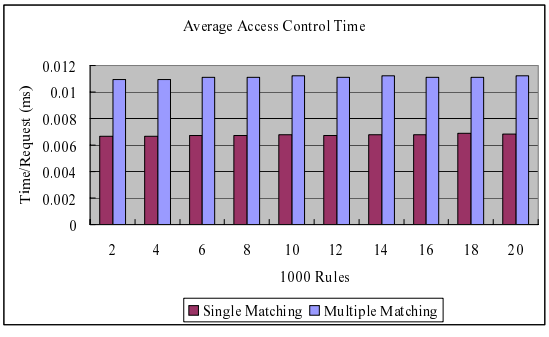

Fig. 11. Access control times vs. #s of Rules

Effect of Policy Size. In this experiment, we see the relationships between the access control performance, mostly decided by the matching time, and the policy size. The accessibility cache was disabled in this experiment. We specified various policy sizes from 2,000 to 20,000 rules, in which 80 $\sim$ 800 users are bound to the kind of access control rules presented in Fig. 9(a). For comparison, we also prepared a group of rules that led to multiple matched targets by specifying parts of the rules in Fig. 9(a) with $+R$.

By processing the paths of an entire XML instance *record.xml*, we calculated the average access control times on random paths. The results are shown in Fig. 11, from which the XML parsing time has been eliminated.

To match an access request against the PMT, on average it costs close to 6.6 microseconds when a single target is matched (S), and 11 microseconds when multiple targets are matched (M). The results show that the access control time is affected by the number of matched targets, but little affected by the policy size for our access control model.

In Sect. 4.2, we showed why the expected matching times and the number of visited nodes during the matching should be sub-linear with regards to the policy size. The experimental results are different from our analysis in that to the same access control request each matching on the corresponding PMT of

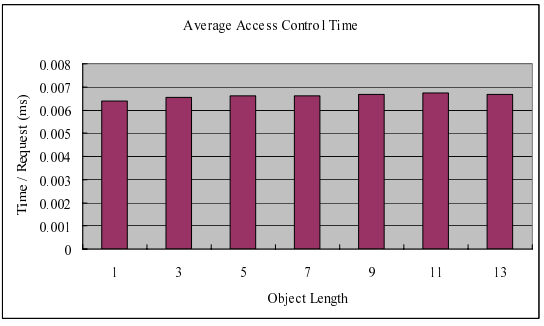

Fig. 12. Access control times vs. Object Length

both S and M results in static outgoing edge(s) in the *Act* matching, the *Sub* matching, and the *Obj* matching. As a consequence, the number of visited nodes for a specific user is static without regard to the policy size.

Effect of Object Length. This experiment was run to examine the relationship between the access control performance and the depths of the targets of the access control rules. The accessibility cache was disabled in the experiment. We specified policies with different object paths varying from $1 \sim 13$ for *standard.xml*. The experimental results are shown in Fig. 12. As the figure shows, the average access control time varies from $6.4 \sim 6.8$ microseconds, showing that it increases only slightly as the object length increases.

As above two experiments show, for the current implementation, the number of matched targets has more effect on the access control performance. The reason is that when multiple match targets are visited during the match, the accessibility can be decided by combing the matching results, during which access conflicts should be resolved.

Performance on Predicates. As presented in the section on the *Pred* matching construction in Sect. 3.4, regardless of the number of the predicates involved, the predicate evaluation is performed only once during the *Pred* matching. Since the performance of data retrieval from the XML database depends on the performance of database itself, we eliminated the time spent on data retrievals in the experiment. The accessibility cache was disabled in this experiment.

Since the match on the PMT is performed twice, once for the *Obj* matching and again for the *Pred* matching, the results show that the access control time is almost twice the time without predicate evaluation: $11 \sim 17$ microseconds per path. The cost was independent of the number of predicates.

Performance on Policy Updating. Real-time update is supported by the PMT, as shown in Sect. 5.3, and both adding and removing access control rules costs $O(m(3 + D))$ time, where m is the number of access control rules, D is the maximum depth of he objects. We tested the update performance by measuring the times for adding 20,000 rules and removing 20,000 rules. From the experiments, the results show that the average time to add an access control

rule to an existing PMT is 33 microseconds, and that the average time to remove an access control rule from the PMT is almost 17 microseconds. Therefore, to update a rule, it takes almost 50 microseconds, since we first remove the original rule and then add the new one. This time is considered to be reasonable for a real access control system.

7 Conclusions and Future Work

In this paper, we have proposed a policy matching tree (PMT) model for providing expressive and scalable access control for XML databases. We first present the PMT generated on the basis of the access control rules. We then described the accessibility evaluation mechanism that matches each access request against the PMT, and decides the accessibility on the basis of the matched results. The PMT also supports // and predicates involved access control, and the expected matching times are shown to be sub-linear relative to the policy size. To improve the performance, we enhanced the access control system with a cache mechanism that eliminates the need for matching when the same subject accesses the same path repeatedly. Comparing with our previous work, the limitations on the expressiveness are expanded, and the runtime PMT updates are supported for policy updates.

To demonstrate the scalability and efficiency of the proposed model, we performed experiments using synthetic and real XML documents. Experimental results show that the PMT model supports 760,000 access control rules and can perform accessibility checks in $6.4 \sim 11$ microseconds per path. This model also supports access control involving predicates at a cost of $11 \sim 17$ microseconds. An accessibility cache further improves performance by a factor of 2 to 5, depending on the XML data structures.

Through these experiments, we also found that when multiple match targets are matched, the performance is twice as slow as the case for a single matched target. In future work, we plan to improve the PMT implementation to achieve better performance for multiple matched targets. We also plan to build the PMT with less memory.

References

1. M. Altinel and M. Franklin: Efficient filtering of XML documents for selective dissemination of information. VLDB (2000) pp.53-64.
2. E. Bertino, S. Castano, E. Ferrari, and M. Mesiti: Controlled access and dissemination of XML documents. ACM WIDM (1999) pp.22-27.
3. E. Bertino, S. Castano, E. Ferrari, and M. Mesiti: Specifying and Enforcing Access Con-trol Policies for XML document Sources. World Wide Web Journal (2000), Vol. 3, No. 3, pp. 139-151.
4. E. Bertino and E. Ferrari: Secure and selective dissemination of XML documents. ACM TISSEC (2002) pp.290-331.
5. M. Bishop, L. Snyder. The transfer of information and authority in a protection system. Proc. 17th ACM Symposium on Operating Systems Principles, 1979.

6. S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon: XQuery 1.0: An XML query language, W3C Working Draft 12 November 2003. http://www.w3.org/TR/xquery/.
7. T. Bray, J. Paoli, and C. M. Sperberg-McQueen: Extensible Markup Language (XML) 1.0. W3C Recommendation. Available at http://www.w3g.org/TR/REC-xml (Feb. 1998).
8. C. -Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi: Efficient filtering of XML documents with XPath expressions. ICDE (2002) pp.235-244.
9. S. Cho, S. Amer-Yahia, L. V. S. Lakshmanan, and D. Srivastava: Optimizing the secure evaluation of twig queries. VLDB (2000) pp.490-501.
10. J. Clark and S. DeRose: XML Path Language (XPath) version 1.0. W3C Recommendation. Available at http://www.w3g.org/TR/xpath (1999).
11. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati: Design and Implementation of an Access Control Processor for XML documents. WWW9 (2000).
12. E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati: A Fine-Grained Access Control System for XML Documents. ACM TISSEC (2002) pp.169-202.
13. A. Deutsch and V. Tannen: Containment of regular path expressions under integrity constraints. KRDB (2001).
14. Y. Diao, P. Fischer, M. Franklin, and R. To.: YFilter: Efficient and scalable filtering of XML documents. Demo at ICDE (2002) pp.341.
15. W. Fan and L. Libkin: On XML integrity constraints in the presence of DTDs. Symposium on Principles of Database Systems (2001) pp.114-125.
16. M. F. Fernandez and D. Suciu: Optimizing regular path expressions using graph schemas. ICDE (1998) pp.14-23.
17. A. Gabillon and E. Bruno: Regulating Access to XML Documents. Working Conference on Database and Application Security (2001) pp.219-314.
18. L. Gong: A Secure Identity-Based Capability System. Proc. IEEE Symposium on Security and Privacy, pp.56-65, 1989.
19. A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne: Document Object Model (DOM) Level 3 Core Specification. Available at http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205 (2004).
20. A. K. Jones, R. J. Lipton, and L. Snyder. A Linear Time Algorithm for Deciding Security. Proc. 17th Symposium on Foundations of Computer Science, pp. 33-41, 1976.
21. R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth: Covering indexes for branching path queries. ACM SIGMOD (2002) pp.133-144.
22. D. D. Kha, M. Yoshikawa, and S. Uemura: An XML Indexing Structure with Relative Region Coordinate. ICDE (2001) pp.313-320.
23. M. Kudo and S. Hada: XML Document Security based on Provisional Authorization. ACM CCS (2000) pp.87-96.
24. Q. Li and B. Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB (2001) pp.361-370.
25. M. Murata, A. Tozawa, M. Kudo and S. Hada: XML Access Control Using Static Analysis. ACM CCS (2003) pp.73-84.
26. OASIS. OASIS Extensible Access Control Markup Language (XACML), Feb. 2003. http://www.oasis-open.org/committees/xacml/docs.
27. F. Neven and T. Schwentick: XPath containment in the presence of disjunction, DTDs, and variables. ICDT (2003) pp.315-329.

28. N. Qi and M. Kudo: Access-condition-table-driven access control for XML databases. ESORICS (2004) pp.17-23.
29. R. S. Sandhu, E. J. Coyne, H. L. Feinstein, C. E. Youman. Role-Based Access Control Models. IEEE Computer, Volume 29, No 2, pp.38-47, February 1996.
30. T. Yu, D. Srivastava, L. V. S. Lakshmanan, and H. V. Jagadish: Compressed Accessibility Map: Efficient Access Control for XML. VLDB (2002) pp.478-489.