社団法人 電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
TECHNICAL REPORT OF IEICE

Tree-based Access Control Mechanism for XML Databases

Naizhen Qi Michiharu Kudo
{naishin, kudo}@jp.ibm.com
Tel: +81-46-215-4428, +81-46-215-4642
Fax: +81-46-273-7428
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502, Japan

Abstract Many XML database applications call for node-level access control on elements, attributes, and text nodes according to their locations and values in an XML document. However, there is trade-off between the performance and the cost of access control especially when the XML database is large–scaled. In order to achieve both the performance and the cost effectiveness, we introduce a tree-based access control mechanism, in which each access control rule is a set of nodes and edges with a leaf in the tree, to determine the subset of all access control rules matching the requested path. We present some optimizations to our algorithm that improve the performance. We also present the results of simulations that our mechanism runs at a nearly constant speed when the number of access control rules increases and that shows acceptable performance for a large amount of the access control rules.

Keyword XML, XML database, Access control, Access control policy, Accessibility check,

1. Introduction

The Extensible Markup Language (XML[1]) is widely used for data presentation, integration, and management because of its rich data structure. Since data with different security levels may be intermingled in a single XML document, such as business transactions and medical records, access control is required on both the element- and attribute-level to ensure that sensitive data will be accessible only to the authorized users. In most of the current research, such node-level access control is specified with XPath[2] to identify the sensitive portion.

XPath is powerful for addressing parts of an XML document. It can not only select a set of nodes at any depth, but also select based on the data content. However, because of such rich capability of node selections, there is trade-off between the performance and the cost on performing access control on the selected nodes. Some of the approaches (e.g. [3, 4, 5, 6, 7]) improve the runtime performance by generating indexing per document on the basis of the access control rules in a pre-processing. Some of the approaches (e.g. [8, 9, 10, 11]) focus more on rich node selections rather than the runtime performance. And there also some approaches (e.g. [12, 13]) trade part of the capability of node selections off against both of the performance and the cost.

However, the XML database usually stores a large amount of documents. For instance, the number of the medical records of a hospital normally reaches several hundreds of thousands. It is obvious that indexing on a per-document basis definitely consumes massive resources. Meantime, when the rich node selection is required, the unacceptable runtime performance may be fatal to a real application. Moreover, the limitations on node selections may bring serious problems to sensitive data protection that concerned nodes cannot be selected.

In this paper, we introduce a novel tree-based access control mechanism which supports rich capability of node selections with the performance and the cost efficiency. Policy matching tree (PMT) is a tree structure in which each node is a test, and the edges are results of such tests. Each lower level of the tree is a refinement of the tests performed at higher levels, and each leaf represents for the action of an access control rule. The PMT can find the access control rules that match a requested path by traversing the tree starting from the root; at each node, we perform the test prescribed by the node and follow all those edges consistent with the result. Such steps are repeated until we get to the leaves. The leaves that are finally visited correspond to the access control rules that match the requested path. Then we evaluate the accessibility based on the rule subset. When the accessibility results *true*, the user is granted to access the XML database to retrieve the requested data item; otherwise, the database access is denied.

The PMT performs almost constantly even the number of access control rules increases sharply. And the efficiency can be achieved independent of the XML documents that it is free from re-computation as long as the access control rules are not updated. In addition, the PMT can provide applicable access control for various query languages including XQuery, SQL XML, and

XPath.

Outline The rest of this paper is organized as follows. After reviewing some preliminaries in Section 2, we introduce the features of PMT in Section 3 and optimizations on PMT in Section 4. Experimental results are reported in Section 5 and the conclusions are summarized in Section 6.

2. Preliminaries

2.1. XPath

An XPath expression can select nodes based on the document structure and the data contents in an XML document. A structure-based selection relies on the structural relationships, which is expressed by '/' and '//'. For example, /record//name selects all name in the subtree of record regardless of name's depth. Value-based selection is fulfilled by attaching a value-based condition on a specific node: if the condition is satisfied, the node is selected. For instance, the expression disclosure[@status='published'] selects disclosure whose status attribute equals 'published'. In this XPath expression, @status='published' is a value-based condition which is called a *predicate*. In addition, '*' is a wildcard that selects any node of the principle node type. For instance, /record/* selects all of the element children of record, while /record/patient/disclosure/@* selects all of the attribute children of disclosure.

2.2. Semantics of access control rule

Various access control policy models have been proposed, but we use the one proposed by Murata et al. [12] in which an access control policy contains a set of 3-tuples rules with the syntax: (Subject, Action, Object) as shown in Table 1. Note Action consists of two factors: Permission and Propagation.

Table 1 Access Control Rule Syntax

Field	Description
Subject	A human user or a user process.
Permission	Grant access (+) or denial access (-)
Propagation	r: without propagation; R: propagation
Object	An XPath expression selects affected nodes

The subject has a prefix indicating the type such as *uid*, *role*, and *group*. '+' stands for a grant rule while '-' for a denial one. The rule with +R or -R is propagation permitted that the access can be propagated downward on the entire subtree, while +r is propagation denied. As an example, (uid:Seki, +r, /a) specifies user Seki's access to /a is allowed but to /a/b is implicit specified since *grant* is not propagated down to the descending paths of /a owing to r. Moreover, according to the denial downward consistency in [12] that the descendants of an inaccessible node are either inaccessible, there is an accessibility dependency between the ancestors and the descendants. Therefore, it is obvious that -r is equivalent to -R; and thus, we specify denial rules only with -R in this paper. In addition, in order to maximize the security of the data items, we (i) resolve access conflicts with the *denial-takes-precedence* [14], and (ii) apply the default denial permission on the paths if no explicit access control is specified. In addition, due to the lack of space, we focus on the read in this paper though either type of *update*, *create*, or *delete* can be implemented with the same mechanisms.

3. Policy Matching Tree

To find the access control rules that match a requested path from a set of access control rules can be solved easily by testing the requested path against each access control rule. This naïve solution runs in time proportional to the number of access control rules. Therefore, it is clear that if the paths are requested by the users at a fast rate, then the access control rules need to be matched at a fast rate as well, so the naïve solution may not perform adequately when the number of access control rules is high. In this section, we introduce a tree structure, *policy matching tree* (PMT), that performs significantly better and the all types of XPath expressions are supported as well.

3.1. The PMT generation

Our mechanism initially pre-processes the access control policy into a PMT, which is used to search for the subset of the rules that match the requested path at runtime. Henceforth, we assume that the object of an access control rules is a conjunction of *node checks*, where each node check represents a test on the node name in the XPath expression. Therefore, the object *obj* of an access control rule is as follows:

$$obj := chk_0 \wedge chk_1 \wedge \ldots \wedge chk_k$$

$$chk_i := test_i \rightarrow result_i$$

where the notation $test_i \rightarrow result_i$ means that $test_i$ checks the name of the node at depth i to see whether it is $result_i$. For example, in the access control rule (role:intern, +r, /record/patient), the object /record/patient therefore

consists of two node checks chk_1 and chk_2, where

$chk_0 = test_0 \rightarrow is$ *'record'*

$chk_1 = test_1 \rightarrow is$ *'patient'*

$test_0$*='check the name of the node at depth 0'*

$test_1$*= 'check the name of the node at depth 1'*

In our PMT, each non-leaf node contains a test on the node name, and the edges from that node representing the results of that test. The edge ends at another non-leaf node for further refinement or a leaf node. A leaf node *l* contains the action of the access control rule. Consequently, an access control rule is described by walking the tree from the root node to *l* and taking the conjunction of the node checks.

Here are some simple examples. Suppose access control rules R_1, R_2 and R_3 as follows:

R_1 *= (role:intern, +r, /record)* (1)

R_2 *= (role:intern, +r, /record/patient)* (2)

R_3 *= (role:intern, +R, /record/diagnosis)* (3)

In this case, the PMT for intern is shown in Figure 1. Note the nodes $test_0$ and $test_1$, and the edge 'record' are shared by R_2 and R_3.

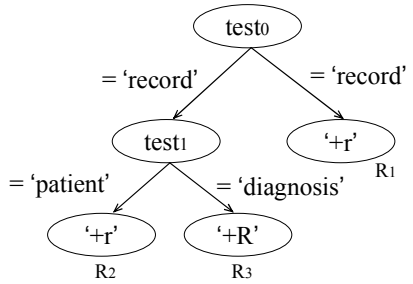


Figure 1 The PMT for R_1, R_2, and R_3

To create the PMT, we start with the empty tree, and we process one access control rule at a time by converting each node in the object (the XPath expression) into a node check, and adding nodes and edges to the PMT as necessary. If the test node and the edge have exists in the PMT, they won't be duplicated. For instance, the processing of R_2 (see Equation 2) creates two non-leaf nodes $test_0$ and $test_1$, and a leaf node containing the action '+r' of Figure 1; and the subsequent processing of R_3 (see Equation 3) creates the remaining leaf node containing the action '+R'. The corresponded edges describing test results are added under related nodes. For instance, the processing of R_2 creates the edges 'record' and 'patient' while R_3 creates the edge 'diagnosis'.

The PMT can be built on a per-subject basis that each uid, role or group has its own tree. However, we can also build a tree for the whole access control system as the occasion demands. In this case, the test on the subject of the access control rule, $test_{sub}$, should be added to the tree as well. For example, in the access control rule (role:intern, +r, /record/patient), the subject *sub* is:

$sub := test_{sub} \rightarrow is$ *' intern'*.

Therefore, to reach a leaf, both the subject check and the node checks should be walked through, where can be represented as: $sub \wedge obj$

3.2. Describe '*' and '//' on the PMT

The PMT can have a special 'don't care edge' to represent the wildcard '*' in XPath expressions that the result of a node check can be anything. For simplicity, we use *-edge to represent it.

Besides the *-edge, we also specify a //-structure to represent the descendant-or-self axis '//' in XPath expressions. The //-structure consists of an edge and a loop, whose relationship to the other is logical *OR* representing as a branch in the PMT from a non-leaf node. The edge represents the result of the node check of the concerned node name, and it reaches to a lower non-leaf node or a leaf node. The loop starts and ends at the same non-leaf node and connects with an edge representing the result of other cases, and each step in the loop performs a node check in order.

Suppose access control rules R_4, R_5, and R_6 as follows:

R_4 *= (role:nurse, +R, /record)* (4)

R_5 *= (role:nurse, -R, /record//info)* (5)

R_6 *= (role:nurse, -R, /record/diagnosis/*)* (6)

The PMT for nurse is shown in Figure 2.

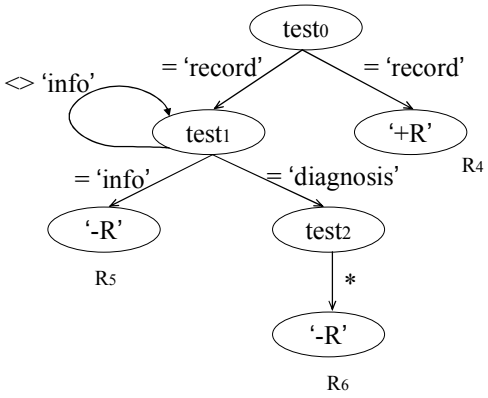


Figure 2 The PMT for R_4, R_5, and R_6

In figure 2, *//info* is described as a loop labeling with 'not info' starts and ends at $test_1$; and the edge 'info' starts at $test_1$ and ends at the leaf node '-R'. Meanwhile, *-edge starts from $test_2$ ends at the leaf node '-R'.

3.3. Describe predicates on the PMT

Predicate is treated as a value filter on XML elements and attributes appearing in an XML document. Since we pre-process the access control rules independent of the XML documents, the data values are unknown beforehand. Therefore, the runtime accessibility checks on predicates cannot finish without accessing the database.

We introduce another type of the non-leaf node to the PMT, which performs tests by interacting with the XML database to obtain the data values for predicate evaluation. Meanwhile, we pre-process the predicates into the non-leaf nodes and edges as well.

Predicates are conditional expressions. We assume the simplest predicate *pred* is:

pred(par) := test_par $\rightarrow$ *Operator value*

where *test_par* retrieves the data value of *par* from the database and evaluates the predicate of the form *par operator value*. A predicate can also contain multiple *pred* connecting with logical *AND* or logical *OR*. For simplicity, predicates are accepted of the form without using parentheses.

Suppose one of the access control rule R_7 is as follows:

R_7 = *(role:intern, +R, /record/patient[@age>50 and @gender='F'])/disclosure* (7)

The part of the PMT generated for R_7 is shown in Figure 3.

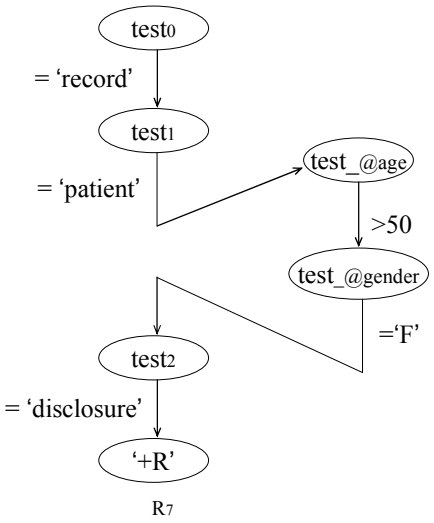


Figure 3 The PMT for R_7

In Figure 3, between the nodes $test_1$ and $test_2$, nodes *test_@age* and *test_@gender* are inserted. Only when both predicate tests succeed in retrieving and checking the values of @age and @gender, the node check on $test_2$ is performed.

4. Tree-based Access Control Mechanism

The algorithm of access control enforcement on a path consists of two sub-algorithms that 1) *Matching*: find the subset of the rules matching the requested path on the PMT; and 2) *Evaluating*: evaluate the accessibility on the basis of the actions of the rule subset.

4.1. Matching algorithm

The algorithm *match* that uses the PMT to find actions of matching rules is given in Figure 4. The idea is to walk the PMT by performing the test prescribed by each node and following the edge that represents the result of the test, and the *-edge if it is present. The set of the actions of the matching rules are the visited leaves. The algorithm *match* in Figure 4 traverses the PMT in a depth-first order, but obviously other orderings, such as breadth-first, also works.

```
procedure match(Tree, path)
  suppose path is presented as the array node.
  visit(Tree, root, 0)

procedure visit(Tree, n, depth)
  if n is a leaf node of Tree then output(n)
  else
    if n is node check
      perform test prescribed by n on node[depth]
      if (n has an edge e with the result of test) or (n has a *-edge e)
      then visit(Tree, (child of n at the endpoint of e in Tree), depth+1)
    else if n is predicate check
      retrieve value of the parameter from DB
      perform test prescribed by n on value
      if (n has an edge e with the result of test)
      then visit(Tree, (child of n at the endpoint of e in Tree), depth)
```

Figure 4 Matching algorithm

For example, to perform access control on the patient records using the PMT in Figure 1. If we use the notation ' $\Rightarrow$ ' to represent the tree traverse, when /record/diagnosis/info is requested, two action leaves are reached through $test_0 \Rightarrow$ *'+r'*, and $test_0 \Rightarrow test_1 \Rightarrow$ *'+R'*. However, in case on the PMT in Figure 2, when /record/patient/disclosure/info is requested, two action leaves are reached through $test_0 \Rightarrow$ *'+R'*, and $test_0 \Rightarrow test_1 \Rightarrow test_1 \Rightarrow test_1 \Rightarrow$ *'-R'* that $test_1$ is walked through for three times because of the loop.

4.2. Evaluating algorithm

To evaluate the accessibility of a path from a set of actions (of rules) can be solved easily by testing from the root node toward the last node in the path to see whether the accessibility is evaluated *true* on each node. This naïve solution runs in time proportional to the product of the path length and the number of the matching action. It

is clear that the naïve solution may not perform adequately when the requested path is long.

We introduce an efficient evaluating mechanism which runs in time proportional to the number of the matching action. An *effect-value* is calculated for the requested path that each level is replaced by a one-digit number which reflects the action imposed to the node on that level. The effect-value can be calculated by traversing all of the matching actions for only one time.

There are four types of actions in our access control system: '-R(-r)', '+r', '+R', and 'no access control (NAC)'. We use 4 different one-digit numbers from 0 to 3 in Table 2 to represent these actions.

Table 2 Actions and their notation

Action	Notation(1 digit number)
-R(-r)	0
NAC	1
+r	2
+R	3

'-R' denies the access to the path, the smallest number is used to represent this action. Sorting with the action range, the rest of the three actions are ordered in NAC, '+r', and '+R' that: NAC applies no access control on the nodes; '+r' applies access control on the object node itself; and '+R' applies not only on the object, but also on its subtree. Consequently, we use 1, 2, and 3 to represent NAC, '+r', and '+R' respectively. The access to a path will be granted if its effect-value alternatively complying with the following two regular expressions.

reg1=(2)+

*reg2=(2)*3(1|2|3)**

It is obvious that if the effect-value does not contain a 0 digit and is bigger than (2)+ in value, the access to the path is granted.

The algorithm *evaluate* in Figure 5 calculates the effect-value from the matching actions for the request path, and evaluates the accessibility. The result of *evaluate* is *true* or *false* and the access is allowed when *true*.

For example, to perform access control on the patient records with the PMT in Figure 1 which representing R_1, R_2, and R_3. When an intern accesses /record/diagnosis/info, the effect-value is initiated to be 111. The effect-value has three digits since the length of the requested path is three. As described in Section 4.1, two action leaves are reachable in this case. Because of '+r' of R_1 that the depth of the object is 0, the effect-value is updated to be 211; and with '+R' of R_3, with the object depth 1, the effect-value is updated to be 231. Since 231 is bigger than 222, the intern's access to /record/diagnosis/info is granted.

```
suppose path length is l
suppose nth element of Actions imposes action_n on the node at
the depth of depth_n
initiate array e with l digits that each digit is 1

procedure evaluate(Actions)
   calculate(Actions, 0)
   generate effectvalue from e
   if effectvalue is not smaller than the number with l digits
         that each digit is 2
   then true else false

procedure calculate(Actions, n)
   get the nth action_n from Actions
   if action_n is '-R' then stop
   else
      if action_n is '+r' then
         if e[depth_n] is not either 2 or 3
         then e[depth_n] = 2
      else
         if e[depth_n] is not 3
         then e[depth_n] = 3
 calculate(Actions, n+1)
```

Figure 5 Evaluating algorithm

4.3. Optimization with ACT-based cache

ACT stands for Access-Condition-Table [13] which is a table storing Boolean expressions for runtime efficient accessibility checks. The Boolean expressions are the results of accessibility checks on the paths, and they are preprocessed on the basis of the access control rules. There are two types of expressions in the ACT: access condition (AC) for the target path itself; and descendant access condition (DAC) for the subset of the path. At runtime, given a path, the ACT provides a proper Boolean expression to decide the accessibility.

Both ACT and PMT are generated independent of the XML documents, and their goals are to enforce efficient access control; however, they have pros and cons which are shown in Table 3.

Table 3 Pros and cons of ACT and PMT

		ACT	PMT
Limitations	'//'	1. Appear once. 2. Identify one node after '//'	No
	'*'	Only accept '//*'	No
	Predicates	No.	No.
Check speed		Constant and faster than PMT	Slower than ACT

From the table, we know the ACT performs faster than the PMT, but it restricts the expressiveness of the access control rules. However, in many XML documents the identical path may appear hundreds or thousands time. For instance, in the case of patient records, each XML document has a record root, and a patient child of record. Rather than evaluate the accessibility every time when record is accessed, to evaluate once and cache the result costs much less. Therefore, we use the ACT as a runtime cache to get a better performance. The ACT is generated at runtime from the results obtained by the PMT. Each of the cached entry contains four fields: *Subject*, *Path*, *AC*, and *DAC*. *Subject* is the value of *uid*, *role*, or *group* of the user sent the request. *Path* is the path expression of an XML node that the subject requests. *AC*, the accessibility check result of the path; while *DAC* is the accessibility check result of the subtree. The algorithm that generates ACT cache from the result of the PMT is given in Figure 6.

```
procedure cache(sub, Tree, path)
  if during match(Tree) walked through //-loop then stop
  else
    if the accessibility check of path results false
    then add cache entry (sub, path, false, false)
    else
      if there are multiple edges with the same result at the last
          test of match(Tree, path)
      then add cache entry (sub, path, true, -)
      else
          if effectvalue applies to the regular expression (2)+
          then add cache entry (sub, path, true, false)
          else
              add cache entry (sub, path, true, true)
```

Figure 6 Caching algorithm

At runtime, the access control system queries the ACT-based cache at first for the accessibility result of a path. If there is no proper entry cached for the path, the system requests the PMT for the result. The PMT performs accessibility check, returns the result, and inserts a new cache entry into the ACT if necessary.

For example, *intern* requests a group of paths in a patient record as follows:

P_1: /record

P_2: /record/patient

P_3: /record/patient/name

P_4: /record/diagnosis

P_5: /record/diagnosis/pathology

P_6: /record/chemotherapy

The accessibility checks are performed on the basis of the PMT in Figure 1. The ACT-based cache is created at the same time as Table 4 shows when intern has finished the accesses.

Table 4 The example of ACT-based optimization

	PMT	ACT cache entry	result(mechanism)
P_1	*true*	(intern, P_1, true, -)	*true* (PMT)
P_2	*true*	(intern, P_2, true, false)	*true* (PMT)
P_3			*false*(ACT cache)
P_4	*true*	(intern, P_4, true, true)	*true* (PMT)
P_5			*true*(ACT cache)
P_6	*false*	(intern, P_6, false, false)	*false*(PMT)

When P_1 is requested, since no cached object exists in the ACT, the PMT runs for the accessibility check. The result from the PMT is *true*. Since there are two edges with the result 'record' of $test_0$ that multiple actions are specified on the nodes in the subtree, the DAC cannot be uniquely decided; therefore, the cache entry is (intern, /record, true, -) in which '-' stands for *not available* (for the sake of the space, we use P_1 to represent the path /record in Table 4). When P_2 is requested, the ACT cache is checked at first that P_2 is not in the ACT but the entry for the ancestor node record is in the ACT. However, the DAC of record is '-' which showing the PMT-based access control should be performed. The PMT evaluates the accessibility for P_2 and inserts a new cache entry (intern, /record/patient, true, false) into the ACT. In this case, the DAC can be decided since only one action is imposed on the subtree. Then P_3 is requested, the ACT cache is checked, and there is no entry for P_3. However, the DAC of /record/patient is provided to show the access to P_3 is denied since the access to any node in the subtree of /record/patient is prohibited. In addition, P_4, P_5, and P_6 are evaluated in the same way.

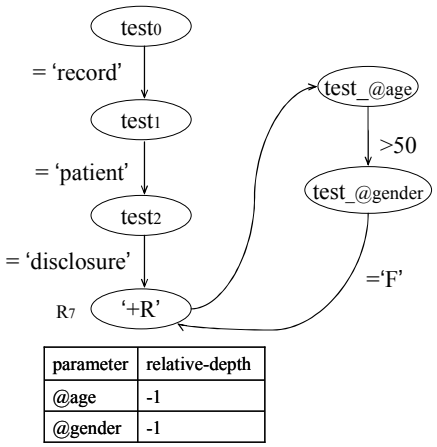
On the other hand, since value-based access control is XML document-dependant that the accessibility cannot be decided on the path level, the ACT cache entry won't be added.

5. Implementation

We implemented the PMT access control mechanism with Java. In this section, we present the details of the implementation of the predicates.

As described in Section 3.3, the *predicate check* interacts with the database and checks the value at runtime. To retrieve the values from the database, the node name and the node location related to the context node are required. Moreover, when multiple nodes are checked, rather than query the database multiple times, to query once and obtain all of the values is definitely more efficient. Therefore, we implement the predicate check in

a different way. As first, we give an example PMT in Figure 7, which represents R_7.



parameter	relative-depth
@age	-1
@gender	-1

Figure 7 The PMT' for R_7

As Figure 7 shows, we use a parameter table to store the necessary information for data retrieval from the database. Each entry in the parameter table contains a parameter name, an element node or an attribute node, and a relative-depth showing the relative distance between the requested node and the node imposed with the predicate. In Figure 7, there are two entries in the parameter table, @age and @gender, both the children of patient, and the relative-depth from patient to disclosure is both -1.

With both the parameter name and its relative-depth, the system knows whose value is requested by the predicate evaluation. However, it also has some limitations. The notation of relative-path can easily represent the ancestor traversal or descendant traversal, but the combination of the both types can not be handled. For example, when the parameter is the sibling of the requested node, relative-depth is 0. However, the relative-depth of the request node itself is also 0, so the relative-depth is not enough to identify them. To deal with such complicated cases, we use a relative-path list which describes the traversal. In this case, the relative-path list is (-1, 1) which means the context node should go to the parent and then find the child with the parameter as its node name.

6. Experiments

To validate the efficiency of the PMT, we ran a variety of experiments to see how our techniques perform. The purposes of the experiments are to explore: 1) the performance of the simple path expressions, 2) the performance of the path expressions containing a predicate, and 3) the scalability in terms of the rule size.

All of the experiments were run on a 1.80GHz Pentium M processor with 1.50GB RAM. The XML document used in the experiments is 9.84MB in size with 328,699 nodes and the maximum node depth is 4.

6.1. The performance of simple paths

We explored the accessibility check time per path when simple path expressions are used to specify the access control target. In this case, the path expressions do not contain predicates, '*', or '//'. The XML document used in the experiments is about 200KB in size with more than 4,700 nodes.

With various access control rule size, from 60 to 470, our experimental results show the accessibility check time on path is 0.006ms at average.

6.2. The performance of handling predicates

We also explored the accessibility check time when a predicate is imposed. The XML document used in the experiments is about 10MB in size with more than 4,700 nodes. We specified a rule set consisting of 12 access control rules, in which 11 rules containing a predicate, as follows:

(uid:U, +r, /Orders)
(uid:U, +r, /Orders/Order[UserKey='U'])
(uid:U, +r, /Orders/Order[UserKey='U']/OrderKeyInfo)
(uid:U, +r, /Orders/Order[UserKey='U']/UserKeyInfo)
(uid:U, +r, /Orders/Order[UserKey='U']/OrderStatusInfo)
(uid:U, +r, /Orders/Order[UserKey='U']/TotalPriceInfo)
(uid:U, +r, /Orders/Order[UserKey='U']/ClarkInfo)
(uid:U, +r, /Orders/Order[UserKey='U']/ShipPriorityInfo)
(uid:U, +r, /Orders/Order[UserKey='U']/Comment)
(uid:U, +r, /Orders/Order[UserKey='U']/Item)
(uid:U, +r, /Orders/Order[UserKey='U']/Item/ShipInstruct)
(uid:U, +r, /Orders/Order[UserKey='U']/Item/Comment)

We control the rule size by replacing 'U' in the rules with user-ids that the rule size equals 12*user_size. We managed to run the experiments for 64,000 users' 768,000 rules without any Java garbage collection triggered. Figure 8 shows the average speed of the accessibility check when the rule size is various from 96,000 to 768,000.

Please note except the path /Orders, the accessibility checks on all of the other paths call for an additional predicate check, in which the data retrieval from the database occurs. However, all experimental data eliminate the time cost on data retrievals.

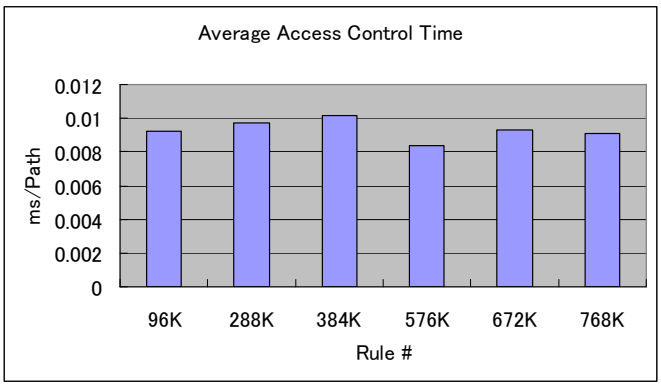


Figure 8 Access control time when rule size changes

From Figure 8, we can see the accessibility check time is between 0.008 - 0.01ms and there is no obvious huge change in value. In other words, the rule size has little influence on the accessibility check speed.

Therefore, we can infer from our experimental results that the accessibility check is close to 0.006 ms when there is no predicate evaluation involved, and 0.008 to 0.01 ms when a predicate evaluation is required. Moreover, we can also figure out that the speed for the objects containing a predicate is almost twice as much as the speed of the simple path expression since the PTM engine runs an additional matching for the predicate test. Therefore, the more predicate tests involve, the more accessibility check time costs; and we can estimate the accessibility check time by considering the predicate number.

7. Conclusion

In this paper, we proposed a PMT-based access control mechanism to provide efficient node-level access control. Using the tree structure, the PMT is capable of handling complicated target selection that there is no limitations on the usages of predicates, '*', or '//'. We built a prototype to demonstrate the effectiveness of the PMT, and we also showed the performance and the scalability through a group of experiments. As our future work, we will extend the current model to support more rules with more efficient memory utilization.

Reference

[1] Extensible Markup Language (XML) 1.0, World Wide Web Consortium (W3C), http://www.w3c.org/TR/REC-xml (Oct. 2000).

[2] XML Path Language (XPath) 1.0, World Wide Web Consortium (W3C), http://www.w3c.org/TR/xpath (Nov. 1999).

[3] M. F. Fernandez and D. Suciu: Optimizing regular path expressions using graph schemas. ICDE (1998) pp.14-23.

[4] R. Kaushik, P. Bohannon, J. F. Naughton, and H. F. Korth: Covering indexes for branching path queries. ACM SIGMOD (2002) pp.133-144.

[5] D. D. Kha, M. Yoshikawa, and S. Uemura: An XML Indexing Structure with Relative Region Coordinate. ICDE (2001) pp.313-320.

[6] Q. Li and B. Moon: Indexing and Querying XML Data for Regular Path Expressions. VLDB (2001) pp.361-370.

[7] 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.

[8] E. Bertino and E. Ferrari: Secure and selective dissemination of XML documents. ACM TISSEC (2002) pp.290-331.

[9] 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).

[10] M.Kudo and S.Hada: XML Document Security based on Provisional Authorization, in ACM Conference Computer and Communications Security (2000).

[11] A. Gabillon and E. Bruno: Regulating Access to XML Documents. Working Conference on Database and Application Security (2001) pp.219-314.

[12] M. Murata, A. Tozawa, M.Kudo and H.Satoshi: XML Access Control Using Static Analysis, in 10th ACM Conference on Computer and Communications Security (Oct, 2003)

[13] N. Qi and M.Kudo: Access-Condition-Table-driven Access Control for XML Databases. ESORICS2004 (Sep. 2004)

[14] E. Bertino, P. Samarati, and S. Jajodia: An extended authorization model for relational database. IEEE trans. on Knowledge and Data Engineering (1997).