A Schema Based Approach to Valid XML Access Control*

CHANGWOO BYUN[1] AND SEOG PARK[2]
[1]*Department of Computer Systems and Engineering*
Inha Technical College
Incheon, 402-752 Korea
[2]*Department of Computer Science and Engineering*
Sogang University
Seoul, 121-742 Korea

As Extensible Markup Language (XML) is becoming a de facto standard for the distribution and sharing of information, the need for an efficient yet secure access of XML data has become very important. An access control environment for XML documents and some techniques to deal with authorization priorities and conflict resolution issues are proposed. Despite this, relatively little work has been done to enforce access controls particularly for XML databases in the case of query access. This work presents an approach to enforce authorizations on XML documents via a filtering system transforming a user query into a rewritten safe query. The basic idea utilized is that a query interaction with only necessary access control rules is modified to an alternative form, which is guaranteed to have no access violations using the metadata of XML schemas and set operations supported by XPath 2.0. This access control mechanism is independent from the underlying XML database engine. Thus, it could be built on top of any XML DBMS, or work as stand-alone services. This work includes other several benefits such as implementation ease, small execution time overhead, and fine-grained controls. The experimental results clearly demonstrate the efficiency of the approach.

Keywords: XML data, XML schema, valid XML access control, access control mechanism, query rewriting

1. INTRODUCTION

As XML [1] is becoming a de facto standard for the distribution and sharing of information, several schemes for XML access control have been proposed. They can be classified in two major categories: view-based *node filtering* and non view-based *query rewriting* techniques. View-based schemes [2-9], suffer from high maintenance and storage costs. To remedy these shortcomings researchers are seeking non view-based query rewriting schemes. Some pre-processing mechanisms have been proposed, such as Static Analysis [10], QFilter [11, 12], and Security View [13, 14]. The advantage of pre-processing methods is that they are independent from the underlying XML database engine, and thus they could be built on top of any XML DBMS, or work as stand-alone services. Despite this, relatively little work has been done to enforce access controls particularly for XML databases in the case of query access. Developing an efficient mechanism for XML databases to control query-based access is therefore the central theme of this paper.

We implemented the Secure Query Filter (*SQ*-Filter) system, which places the focus on the abstraction of only necessary access control rules and the query modification. The abstraction of necessary access rules necessitates the development of an effective and

Received September 16, 2008; revised June 26 & November 13, 2009; accepted January 5, 2010.
Communicated by Chih-Ping Chu.
* This work was supported by the second stage of the Brain Korea 21 Project in 2010.

efficient choosing mechanism that abstracts only the ones appropriate to a user query. The traditional access control enforcement mechanism for XML documents uses all access control rules corresponding to a query requester. The basic notion pursued is one of encoding trees. The *SQ*-Filter system uses the PRE (order) and POST (order) ranks of access control rules and queries to map each (PRE, POST) onto a two-dimensional plane, for instance, the PRE/POST plane. The finding of necessary access control rules is then reduced to process a part of regions in this PRE/POST plane. Query modification is the development of an efficient query rewriting mechanism that transforms an unsafe query into a safe yet correct one that keeps the user's access control policies. Rewriting the query early on will reduce query processing overhead to retrieve the allowed set of XML sub-trees as compared to retrieving and testing. In NFA approaches [10-12], without the metadata of the XML Schema (DTD), the process of rewriting queries may be particularly slower and incorrect because more states are being traversed to process "*" and "//". The basic notion pursued is DTD labeling and a simple label comparison. A "*" node is unnecessary if the DTD shows an actual node from the node (*i.e.*, parent node) before "*" to the node (*i.e.*, child node) after "*". In addition, the "//" axis is also unnecessary if the DTD shows that there is a single path from the node before "//" to the node after "//". If so, the "//" axis is replaced with the deterministic sequence of "/" steps. It is a very important factor to prohibit the query rewriting processor from making an incorrect query. We conducted an extensive experimental study which shows that the SQ-Filter system improves access decision time and generates a more exact rewritten query.

The rest of the paper is organized as follows: Section 2 briefly reviews related works and describes their weaknesses. Section 3 briefly reviews background knowledge on the topic at hand, which consists of the semantics of a valid XML, the role of XPath expression in access control rules, access control polices, definition of secure query rewriting in XML access control, and PRE/POST structure. Section 4 introduces the architecture of the SQ-Filter system, which runs three components. These are the QUERY ANALYZER component which acquires information from a query, the ACR FINDER component which plays an important role in eliminating all the unnecessary access control rules for a query, and the QUERY EXECUTOR component which makes a safe rewritten query by extending/eliminating query tree nodes and combining a set of nodes by the set operations. In section 5, we prove the correctness of the *SQ*-Filter system. Section 6 presents the results of our experiments, which reveal the effective performance of the *SQ*-Filter system. Finally, section 7 summarizes our work.

2. RELATED WORK

The authorizations for XML documents should be associated with protection objects at different granularity levels. In general, existing XML access control models assume access control rules, which are identified as quintuple (Subject, Object, Access-type, Sign, and Type) [2-9]. The subject refers to the user or user group with authority, the object pertains to an XML document or its specific portion, access-type means the kind of operations, the sign could either be positive (+) or negative (−), and the type shows "R(ecursive)" or "L(ocal)." In the studies of E. Damiani *et al.* [2, 3], E. Bertino *et al.* [4-6], A. Gabilon and E. Bruno [7], and C. Farkas *et al.* [8, 9], the object part of access control rule is associated with each XML document/DTD. Because of the hierarchical nature of XML, the notion of scope of an access control rule is introduced in most of the current approaches. The scope can be (i) the node only, (ii) the node and its attributes, (iii) the node and its text node children, and (iv) the node, its attributes, all its descendants and their attributes. If the scope of a rule is (i) or (ii), then the rule is called L(ocal). If its scope is (iii) or (iv), it is called R(ecursive).

The traditional XML access control enforcement mechanism [2-9] as mentioned above is a view-based enforcement mechanism. W. Fan *et al.* [13, 14] introduced a virtual security view mechanism, which is provided to a user as a view DTD and is used for query-rewriting. The security view, however, is based on the user and not the query. Thus, a query-rewriting algorithm made use of unnecessary parts in a security view. It provides a useful algorithm for computing the view using tree labeling. However, aside from its high cost and maintenance requirement, this algorithm is also not scalable for a large number of users.

To remedy this view-based problem, M. Murata *et al.* [10] simply focused on filtering out queries that do not satisfy access control policies. J. M. Jeon *et al.* [15] proposed the access control method that produces the access-granted XPath expressions from the query XPath expression by using access control tree (XACT), where the edges are a structural summary of XML elements, and the nodes contain access control XPath expressions. Since XACT includes all users' access control rules, it is very complicated and leads to computation time overhead. B. Luo *et al.* [11] and P. Ayyagari *et al.* [12] took extra steps to rewrite queries in combination with related access control policies before passing these revised queries to the underlying XML query system for processing. However, the shared Nondeterministic Finite Automata (NFA) involves many unnecessary access control rules from the query point of view, which further result in a time-consuming decision during which the query should have already been accepted, denied, or rewritten. In addition, this approach is very inefficient for rewriting queries with the descendant-or-self axis ("//") because of the exhaustive navigation of NFA. The many queries on XML data have path expressions with "//" axes because users may not be concerned with the structure of data and intentionally make path expressions with "//" axes to get the intended results.

W. Fan *et al.* [14] and E. Damiani [17] focused on Deterministic Finite Automaton (DFA) based query rewriting approach for avoiding the many backtrackings inherent to NFAs and resolving the original schema disclosure. Although DFA approaches are theoretically efficient, there are no experimental results.

3. PRELIMINARY

We reviews background knowledge on the topic at hand, which consists of the semantics of a valid XML, the role of XPath expression in access control rules, access control polices, and PRE/POST structure which is the metadata of the *SQ*-filter system.

3.1 XML

An XML document consists of elements, attributes, and text nodes. These elements collectively form a tree. The content of each element is a sequence of elements (nested elements) or text nodes. An element has a set of attributes, each of which has a name and a value. Attributes provide additional information on elements, and thus increasing the semantics one can specify for elements.

XML specification [1] defines two types of document: well-formed and valid ones. A well-formed document must conform to the three rules. Valid documents, on the other hand, should not only be well formed but should also have a Document Type Definition (DTD) or XML Schema, which the well-formed document conforms to. If several people are requiring query using XPath or XQuery, the query structure must conform to DTD or the XML Schema structure. In this work, we considered a valid XML with a DTD.

3.2 XPath Expression and Access Control Rule

XPath is a path expression language of a tree representation of XML documents. A typical path expression consists of a sequence of steps. Each step works by following a relationship between nodes in the document: child, attribute, ancestor, descendant, *etc.* Any step in a path expression can also be qualified by a predicate, which filters the selected nodes.

Meanwhile, XPath 2.0 [18] supports operations (*i.e.*, UNION, INTERSECT, and EXCEPT) that combine two sets of node. Although these operations are technically non-path expressions, they are invariably used in conjunction with path expressions so they are useful in transforming unsafe queries into safe ones.

To enforce the fine-level granularity requirement, authorization models for regulating access to XML documents use XPath which is a suitable language for both query processing and the object part of an XML access control authorization.

3.3 Access Control Policies

In general, some hierarchical data models (*e.g.*, Object-Oriented Data Model, XML Data Model, *etc.*) exploit the implicit authorization mechanism combining positive and negative authorizations [19]. "Open policy" grants a query for a node whose access-control information is not defined. "Closed policy," on the other hand, denies a query for a node whose access-control information is not defined. A Positive/Negative authorization mechanism for the closed policy generally assumes that a subject has a null authorization on every object until explicitly authorized. For the strict data security, we used "most specific precedence" for the propagation policy, "denials take precedence" for the conflict resolution policy, and "closed policy" for decision policy to keep the data safe.

The combination of negative authorization with positive authorizations allows the definition of positive authorizations as exceptions to negative authorization at a higher level in granularity hierarchy. M. Murata *et al.* [10] called this combination as "valid read accessibility views." Similarly, the combination of a positive authorization with a negative one specifies exceptions to a positive authorization. This approach leads to two problems, the so-called "invalid read accessibility views."

In this paper, we chose "valid read accessibility views." To ensure that an access control policy is "valid read accessibility views" in a positive/negative authorization mechanism, we proposed a new concept of generating access control rules. We defined this as "Integrity Rule of Access Control Rules."

Definition 1 [Integrity Rules of *ACRs*] It is impossible for any node, which is not in the scope of positive ACR (*A*ccess *C*ontrol *R*ule)s, to have negative *ACRs*.

3.4 PRE/POST Structure

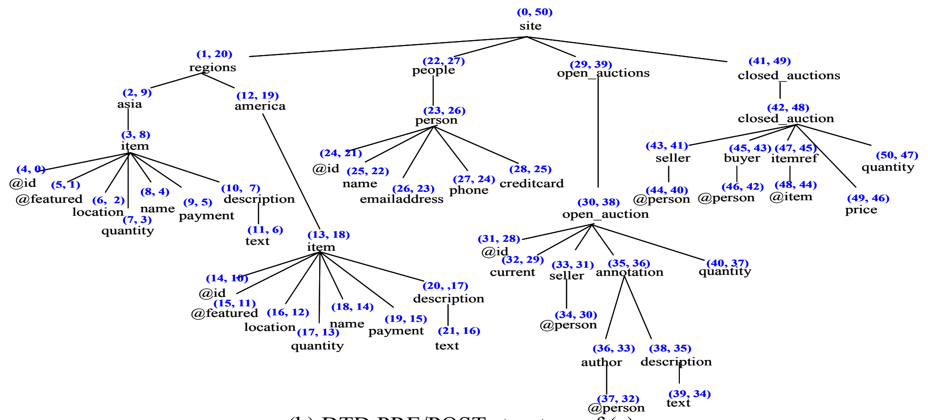
An XPath expression [18] declares the query requirement by identifying the node of interest via the path from the root of the document to the elements which serve as the root of the sub-trees to be returned [20]. The QFilter [11] defines these semantics as "answer-as-sub-trees." We call the root of sub-trees to be returned as target node.

Definition 2 [Target Node] A target node of a given XPath is the last node except for the predicates.

For example, the target node of an XPath, */site/people/person* is a *person* node. Another example is the target node of an XPath, */site/regions/aisa/item[@id="xxx"]* which is an *item* node. Fig. 1 (a) shows a portion of an auction.DTD source extracted from the XMark [21], which we considered as a running example in our paper. Fig. 1 (b) shows that the nodes of the DTD tree were assigned with PRE (order) and POST (order) ranks,

```
<!ELEMENT site              (regions, people, open_auctions, closed_auctions)>
<!ELEMENT regions           (asia, america)>
<!ELEMENT asia              (item*)>
<!ELEMENT america           (item*)>
<!ELEMENT item              (location, quantity, name, payment, description)>
<!ATTLIST item              id ID #REQUIRED featured CDATA #IMPLIED>
<!ELEMENT location          (#PCDATA)>
<!ELEMENT quantity          (#PCDATA)>
<!ELEMENT name              (#PCDATA)>
<!ELEMENT payment           (#PCDATA)>
<!ELEMENT description       (text)>
<!ELEMENT text              (#PCDATA)>
<!ELEMENT people            (person*)>
<!ELEMENT person            (name, emailaddress, phone?, creditcard?)>
<!ATTLIST person            id ID #REQUIRED>
<!ELEMENT emailaddress      (#PCDATA)>
<!ELEMENT phone             (#PCDATA)>
<!ELEMENT creditcard        (#PCDATA)>
<!ELEMENT open_auctions     (open_auction*)>
<!ELEMENT open_auction      (current, seller, annotation, quantity)>
<!ATTLIST open_auction      id ID #REQUIRED>
<!ELEMENT current           (#PCDATA)>
<!ELEMENT seller            EMPTY>
<!ATTLIST seller            person IDREF #REQUIRED>
<!ELEMENT annotation        (author, description?)>
<!ELEMENT author            EMPTY>
<!ATTLIST author            person IDREF #REQUIRED>
<!ELEMENT closed_auctions   (closed_auction*)>
<!ELEMENT closed_auction    (seller, buyer, itemref, price, quantity)>
<!ELEMENT buyer             EMPTY>
<!ATTLIST buyer             person IDREF #REQUIRED>
<!ELEMENT itemref           EMPTY>
<!ATTLIST itemref           item IDREF #REQUIRED>
<!ELEMENT price             (#PCDATA)>
```

(a) Auction.DTD.



(b) DTD PRE/POST structure of (a).

Fig. 1. A running example.

Tag-Name	Pre	size	level	post	Parent
site	0	50	0	50	
regions	1	20	1	20	site
asia	2	9	2	9	regions
item	3	8	3	8	asia
@id	4	0	4	0	item
@featured	5	0	4	1	item
location	6	0	4	2	item
quantity	7	0	4	3	item
name	8	0	4	4	item
payment	9	0	4	5	iem
description	10	1	4	7	item
text	11	0	5	6	description
…	…	…	…	…	…
people	22	6	1	27	site
person	23	5	2	26	people
@id	24	0	3	21	person
name	25	0	3	22	person
emailaddress	26	0	3	23	person
phone	27	0	3	24	person
creditcard	28	0	3	25	person

Tag-Name	Pre	size	level	post	Parent
open_auctions	29	11	1	39	site
open_auction	30	10	2	38	open_auctions
@id	31	0	3	28	open_auction
current	32	0	3	29	open_auction
seller	33	1	3	31	open_auction
@person	34	0	4	30	seller
annotation	35	4	3	36	open_auction
author	36	1	4	33	annotation
@person	37	0	5	32	author
description	38	1	4	35	annotation
text	39	0	5	34	description
quantity	40	0	3	37	open_auction
closed_auctions	41	9	1	49	site
closed_auction	42	8	2	48	closed_auctions
seller	43	1	3	41	closed_auction
@person	44	0	4	40	seller
buyer	45	1	3	43	closed_auction
…	…	…	…	…	…
quantity	50	0	3	47	closed_auction

(c) Relational storage of (b).

Fig. 1. (Cont'd) A running example.

as seen when parsing the DTD tree sequentially. Fig. 1 (c) shows the actual relational DTD representation. We called this the PRE/POST Structure (PPS) of the DTD tree. LEVEL refers to a DTD tree level, while SIZE is the sub-tree size of any node. This PRE/SIZE/LEVEL encoding is equivalent to PRE/POST since POST = PRE + SIZE – LEVEL [22, 23]. PARENT refers to the parent node of the Tag-Name node in the DTD tree. We avoided any unnecessary *ACRs* through PPS information. PARENT information enables efficient finding of the nodes for XPath expressions with "//" axes. Further details are provided in section 4.

4. OVERVIEW OF THE SQ-FILTER SYSTEM

In general, if there may exist positive access control rules ($ACR^{+}s$) and negative access control rules (ACR^{-}s), the safe rewritten query (Q') against a query (Q) becomes as follows [10, 11]:

$$\begin{aligned} Q' &= Q \text{ INTERSECT } (ACR^{+}s \text{ EXCEPT } ACR^{-}s) \\ &= (Q \text{ INTERSECT } ACR^{+}s) \text{ EXCEPT } (Q \text{ INTERSECT } ACR^{-}s). \end{aligned}$$

In this paper, we put two problems:

• First is the scope of the access control rules which is compared with the query.

Here, the *ACRs* are all the access control rules of a user. $ACRs_{Q}$, on the other hand, are all the access control rules of a user query.

$$\begin{aligned} Q' &= Q \text{ INTERSECT } (ACR^{+}s_{Q} \text{ EXCEPT } ACR^{-}s_{Q}) \\ &= (Q \text{ INTERSECT } ACR^{+}s_{Q}) \text{ EXCEPT } (Q \text{ INTERSECT } ACR^{-}s_{Q}) \end{aligned}$$

where $ACR^{+}s_Q \subseteq ACR^{+}s$, $ACR^{-}s_Q \subseteq ACR^{-}s$.

- Second is the correctness of the rewritten query of (Q INTERSECT ACR) with "*" or "//."

The objective of the *SQ*-Filter system is to select only the necessary *ACRs* for processing a user query, and to rewrite the unsafe query into a new safe query. In this section, we proposed the architecture of the *SQ*-Filter system and described its core components.

4.1 The Architecture of the *SQ*-Filter System

The architecture of the *SQ*-Filter system is shown in Fig. 2. The primary input is a user query. If an input query is accepted, the output may be the original input query or a modified query which has filtered out the conflicting or redundant parts from the original query. Otherwise, the result may be rejecting the query.

The *SQ*-Filter system is divided into two parts. At compile time, the *SQ*-Filter system constructs the PPS of a DTD (in section 3.4). It also constructs *ACR* and *PREDICATE* databases. At run time, the *SQ*-Filter system runs three components, namely, *QUERY ANALYZER*, *ACR FINDER*, and *QUERY EXECUTOR*.

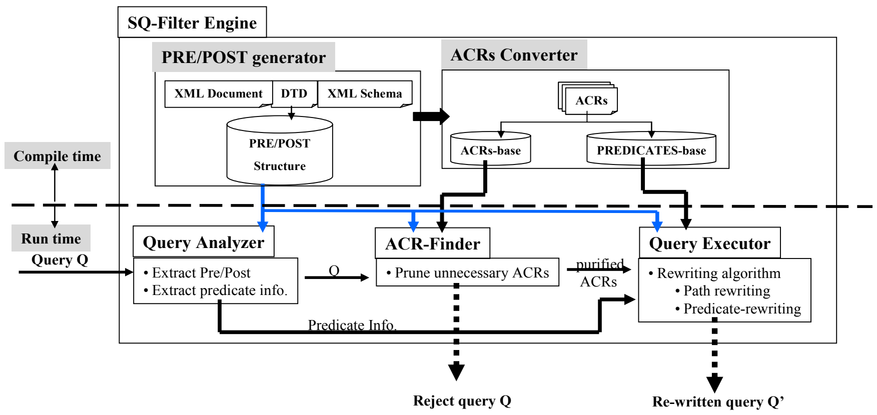


Fig. 2. The architecture of the *SQ*-Filter system.

After a security officer determines the *ACRs* of which each object part uses as an XPath expression in Fig. 3, the *ACR Converter* stores the XPath information into the *ACR* and *PREDICATE* databases in Fig. 4 at compile time. Note that the entity of PRE and POST columns may be more than one. For example, the target node of *R*1 is *item*. The (PRE, POST) value set of the *item* is (3, 8) and (13, 18). This value set is stored into the PRE and POST columns, respectively. Moreover, *R*1 has one predicate ([location = 'xxxx']). The parent element of the predicate is *item*. The entity of Parent-PRE column is (3, 13), and the entities of property, operator, and value columns are 'location', '=', and 'xxxx', respectively. Finally, *P*1 as Predicate ID is stored into the *P*_link column in the *ACRs* database. The entity of *Ptr_ACR*-column means corresponding negative *ACRs*. In a similar way, other *ACRs* are stored into the *ACRs* and *PREDICATES* databases.

Positive ACRs

(R1): /site/regions/*/item[location="LA"]

(R2): /site/people/person[name = "chang"]

(R3): //open_auction[quantity]/seller

Negative ACRs

(R4): /site/regions/*/item/payment

(R5): /site/people/person/creditcard

(R6): /site/*/open_auction[@id>50]/seller[@person="chang"]

Fig. 3. Sample positive/negative *ACRs*.

ACR$^+$-base

rule	path	Pre	Post	P_link	Ptr_ACR$^-$
R1	/site/regions/*/item	3, 13	8, 18	P1	R4
R2	/site/people/person	23	26	P2	R5
R3	//open_auction/seller	33	31	p3	R6

ACR$^-$-base

rule	path	Pre	Post	P_link
R4	/site/regions/*/item/payment	9, 19	5, 15	
R5	/site/people/person/creditcard	28	25	
R6	/site/*/open_auction/seller	33	31	p4, p5

PREDICATES-base

P-id	Parent-PRE	property	operator	value
P1	3, 13	location	=	LA
P2	23	name	=	chang
P3	30	quantity		
P4	30	@id	>	50
p5	33	@person	=	chang

Fig. 4. Sample *ACR* and *PREDICATE* databases.

4.2 QUERY ANALYZER Component

The objective of the *QUERY ANALYZER* (QA) is to acquire information from a query. First, the QA looks up the PPS and gets the (PRE, POST) pairs of the target node of a query. Second, the QA eliminates unnecessary (PRE, POST) pairs. Finally, the QA divides the query into sub-queries for each remaining (PRE, POST) pair and obtains the predicate information of each sub-query.

Given a query Q_1, */site/people/person[name="chang"]/phone/*, the target node of Q_1 is *phone* node. The QA looks up the PPS and gets the (27, 24) value of the target node *phone*. Then it gets the (23, 26) value of *person* node, which has a predicate information (name, "=", "chang").

Note that there may also be more than one (PRE, POST) pairs of the target node of a query. Let us take a query */site//name*, for example. The target node of the query is the name node. Thus, the (PRE, POST) pairs of the name are (8, 4), (18, 14), and (25, 22). In this case, a query Q_i is divided into three sub-queries Q_{i1}, Q_{i2}, and Q_{i3} (Q_i = Q_{i1} UNION Q_{i2} UNION Q_{i3}). Its main idea is that the preorder (postorder) value of each node of a query is less (greater) than that of the target node of the query. Fig. 5 shows the pruning algorithm that eliminates the unsuitable (PRE, POST) pairs of a query. Henceforth, a query refers to each sub-query.

4.3 ACR FINDER Component

The objective of the *ACR FINDER* (ACR-F) component is to eliminate all the unnecessary *ACRs* for a query. As shown in Fig. 6, namely, the PRE/POST plane of *ACRs*, the target node of a query induces five partitions of the plane. Each partition has a special relation with the query.

```
Input: a query
Output: suitable (PRE, POST) values of target node of the query
BEGIN
1. for each ($Pr_{tn}$, $Po_{tn}$) value of the target node of the query
2. {    for ($Pr_{step}$, $Po_{step}$) value of each node of the query
3.          if (!($Pr_{step} < Pr_{tn}$ and $Po_{step} > Po_{tn}$))
4.              break;
5.      suitable (PRE, POST) set ← ($Pr_{tn}$, $Po_{tn}$)
6. }
END
```

Fig. 5. The Prune-TNs algorithm.

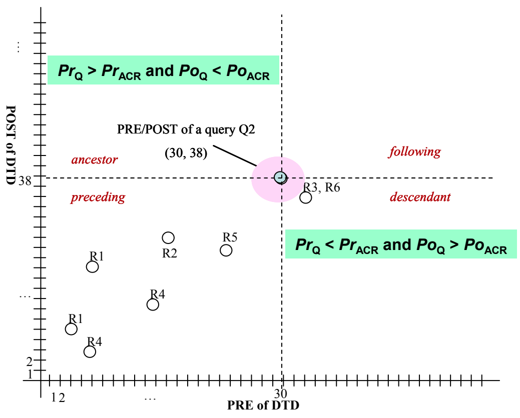


Fig. 6. Semantics of PRE/POST plane of positive *ACRs*.

Let (Pr_Q, Po_Q), (Pr_{ACR}, Po_{ACR}), and ($Pr_{Q'}$, $Po_{Q'}$) pairs be the (PRE, POST) values of a query Q, an *ACR*, and a safe rewritten query Q', respectively.

Definition 3 [upper-right partition: FOLLOWING ACR] *FOLLOWING ACR* means that the preorder and postorder values of an *ACR* are greater than those of a query subject to $Pr_Q < Pr_{ACR}$, $Po_Q < Po_{ACR}$.

Definition 4 [lower-left partition: PRECEDING ACR] *PRECEDING ACR* means that the preorder and postorder values of an *ACR* are less than those of a query subject to $Pr_Q > Pr_{ACR}$, $Po_Q > Po_{ACR}$.

The FOLLOWING and PRECEDING *ACRs* have no connection with the query Q. Thus, we can put aside the *ACRs* for processing the query Q.

Definition 5 [upper-left partition: ANCESTOR ACR] *ANCESTOR ACR* (*including PARENT ACR*) means that the preorder (postorder) value of an *ACR* is less (greater) than that of a query subject to $Pr_Q > Pr_{ACR}$, $Po_Q < Po_{ACR}$.

Definition 6 [lower-right partition: DESCENDANT ACR] *DESCENDANT ACR* (*including CHILD ACR*) means that the preorder (postorder) value of an *ACR* is greater (less) than that of a query subject to $Pr_Q < Pr_{ACR}$, $Po_Q > Po_{ACR}$.

```
Input: ($Pr_Q$, $Po_Q$) := QA(query), ACRs
Output: applicable ACRs'
BEGIN
for each rule R1 in ACRs of a query requester
    if (($Pr_Q \geq Pr_{R1}$ and $Po_Q \leq Po_{R1}$) or // ANCENSTOR
       ($Pr_Q \leq Pr_{R1}$ and $Po_Q \geq Po_{R1}$))    // DESCENDANT
          ACRs' ← R1;
END
```

Fig. 7. The *ACR-FINDER* algorithm.

Definition 7 [SELF ACR] *SELF ACR* means that the (PRE, POST) pair of an *ACR* is equal to that of a query subject to $Pr_Q = Pr_{ACR}$, $Po_Q = Po_{ACR}$.

Given a query Q_2, *//open_auction[@id<100]*, the target node of Q_2 is an *open_ aunction* node. Recall the *ACR* database in Fig. 4. By the QA, the (PRE, POST) value of the target node *open_aunction* is (30, 38). Only R_3 and R_6 are the necessary *ACRs* for Q_2 because they are the *DESCENDANT ACRs* of Q_2. R_1 to R_5 are the *PRECEDING ACRs* of the query. They are identified as unnecessary *ACRs* for Q_2.

Fig. 7 shows the *ACR-FINDER* algorithm which finds the *DESCENDANT* (or *AN-CESTOR* or *SELF*) *ACRs* for a query.

Theorem 1 The *ACR-FINDER* algorithm abstracts all related access control rules for a query.

Proof: Let (Pr_{TN}, Po_{TN}) be the (PRE, POST) value of the target node of a query Q. Suppose that SN is any node in sub-nodes of Q. Then the (PRE, POST) value of SN is as follows:

$$Pr_{TN} < Pr_{SN} < Pr_{TN} + \text{SIZE}(TN),\ Po_{last\_sibling\_node\_of\_TN} < Po_{SN} < Po_{TN}. \tag{1}$$

Suppose that FN is any node in following nodes of Q. Then the (PRE, POST) value of FN is as follows:

$$Pr_{TN} < Pr_{FN},\ Po_{TN} < Po_{FN}. \tag{2}$$

Especially, the (PRE, POST) value of the first following node (*first_FN*) of Q is as follows:

$$Pr_{first\_FN} = Pr_{TN} + \text{SIZE}(TN) + 1. \tag{3}$$

Thus, by Eqs. (1) and (2), $Po_{SN} < Po_{TN} < Po_{FN}$, and by Eqs. (1) and (3), we obtain Eq. (4).

$$Pr_{SN} < Pr_{TN} + \text{SIZE}(TN) < Pr_{first\_FN} = Pr_{TN} + \text{SIZE}(TN) + 1. \tag{4}$$

Eq. (4) means that the preorder value of any node (SN) in the sub-nodes of Q is less than that of the first following node (*first_FN*) of Q. Therefore, $Pr_{SN} < Pr_{FN}$. Therefore, we can obtain $Pr_{SN} < Pr_{FN}$ and $Po_{SN} < Po_{FN}$. That is, the *FOLLOWING ACRs* have no connection with the query, as will be proven. The same is the case with the *PRECEDING ACRs*. ❑

4.4 QUERY EXECUTOR Component

The goal of the *QUERY-EXECUTOR* (QE) is to make a rewritten safe query by extending/eliminating query tree nodes and combining a set of nodes by the set operations. The positive and negative *ACRs* that passed by the *ACR*-F are classified into three groups (*SELF*, *ANCESTOR*, and *DESCENDANT ACRs*). The simple process of the QE is as follows:

- Compare the query with each negative access control rule and produce a modified query.
- Combine each modified query by UNION operation.
- Compare the query with each positive access control rule and produce a modified query.
- Combine each modified query by UNION operation.
- Combine each result query of ④ and ② by EXCEPT operation.
- Output the final result query of ⑤, (Q INTERSECT $ACRs^{+}_{Q}$) EXCEPT (Q INTERSECT $ACR^{-}s_{Q}$).

However, we may semantically obtain an efficient process method using an access control group such as *ANCESTOR*, and *DESCENDANT ACRs*.

4.4.1 No predicates

If “predicates” are not used in a query and *ACRs*, there are five types of them as shown in Fig. 8. By the “Integrity Rule of *ACRs* (Definition 1),” the scope of a negative *ACR* must be included in the scope of a positive *ACR*. Type 1 means that positive and negative *ACRs* are the *ANCESTOR ACRs* for a query. For Type 1, the result is an empty set. Type 2 means that a positive *ACR* is the *ANCESTOR ACR* to control the parts of a query, but negative *ACRs* do not exist. Type 5 means that a positive *ACR* is a *DESCENDANT ACR*, and negative *ACRs* do not exist. For Types 2 and 5, the latter set is empty, making the result the former set.

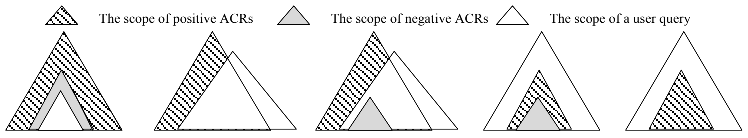

The scope of positive ACRs
The scope of negative ACRs
The scope of a user query
(a) Type 1. (b) Type 2. (c) Type 3. (d) Type 4. (e) Type 5.

Fig. 8. Semantics between a query without predicates and *ACRs* without predicates.

Type 3 means that a positive *ACR* is the *ANCESTOR ACR* to control the parts of a query, but parts (or all) of negative *ACRs* are the *DESCENDANT ACRs*. Type 4 means that both positive and negative *ACRs* are the *DESCENDANT ACRs*. The result for type 3 and 4 is (Q INTERSECT $ACR^{+}s_{Q}$) EXCEPT (Q INTERSECT $ACR^{-}s_{Q}$).

One example could be a positive *ACR* ACR_{j1}, two negative *ACRs* ACR_{j2} and ACR_{j3}, and two queries Q_{j1} and Q_{j2}.

ACR_{j1}: /site (+), the (PRE, POST) value of the *site* node is (0, 50).

ACR_{j2}: /site//asia (–), the (PRE, POST) value of the *asia* node is (2, 9).
ACR_{j3}: /site//person/name (–), the (PRE, POST) value of the *name* node is (25, 22).
Q_{j1}: /site//name, the (PRE, POST) values of the *name* node are (8, 4), (18, 14), and (25, 22). Thus, Q_{j1} is divided into three sub-queries such as Q_{j11} for (8, 4), Q_{j12} for (18, 14), and Q_{j13} for (25, 22), respectively.
Q_{j2}: /site//person, the (PRE, POST) value of the *person* node is (23, 26).

The *ACRs* related to Q_{j11} are ACR_{j1} and ACR_{j2}. The ACR related to Q_{j12} is ACR_{j1}. The *ACRs* related to Q_{j13} are ACR_{j1} and ACR_{j3}. The *ACRs* related to Q_{j2} are ACR_{j1} and ACR_{j3}. Q_{j11} and Q_{j13} are Type 1, Q_{j12} is Type 2, and Q_{j2} is Type 3. In the case of Type 1, a query should be rejected at once because the scope of a query is included in the scope of a negative *ACR*. In the case of Type 2, authorized parts which have no connection with a query, should be eliminated by the INTERSECT operation. In the case of Type 3, not only authorized parts which have no connection with a query but also unauthorized parts of a query should be eliminated by the EXCEPT operation.

Another example would be two positive *ACRs* ACR_{k1} and ACR_{k3}, a negative *ACR* R_{k2}, and two queries Q_{k1}, and Q_{k2}.

ACR_{k1}: site//item (+), the (PRE, POST) values of the *item* node are (3, 8), and (13, 18).
ACR_{k2}: site//item/name (–), the (PRE, POST) values of the *name* node are (8, 4) and (18, 14).
ACR_{k3}: site//open_auction/annotation (+), the (PRE, POST) value of the *annotation* node is (35, 36).
Q_{k1}: site//america, the (PRE, POST) value of the *america* node is (12, 19).
Q_{k2}: site//open_auction, the (PRE, POST) value of the *open_auction* node is (30, 38).

The *ACRs* related to Q_{k1} are ACR_{k1} and ACR_{k2}, while the *ACR* related to Q_{k2} is ACR_{k3}. Therefore, Q_{k1} is Type 4, and Q_{k2} is Type 5. In the case of Types 4 and 5, the unauthorized parts of a query should be eliminated by the INTERSECT operation.

4.4.2 Handling *Q* INTERSECT *ACR*

The handling *Q INTERSECT ACR* algorithm is very simple. First, it is to decide a *standard* XPath expression between a query and an *ACR*, which has a larger PRE value and a smaller POST value of each target node between them. Without additional computation, a *standard* XPath expression may however be selected by passing the ACR-F. While an *ANCESTOR ACR* is a *standard* XPath espression for a query, a query is a *standard* XPath expression for a *DESCENDANT ACR*. Second, it is to replace a wild card "*" with an actual node name (element tag) and remove unnecessary "//" axes. An "*" element is unnecessary if the DTD shows an actual element from the element (*i.e.*, parent node) before "*" to the element (*i.e.*, child node) after "*". In addition, the "//" axis is also unnecessary if the DTD shows that there is a single path from the element before "//" to the element after "//". If so, the "//" axis is replaced with the deterministic sequence of "/" steps.

Let us take an example of a *standard* XPath expression, *//regions/* /item/name*. Since the (PRE, POST) value set of the target node (*name*) of the XPath expression is (8, 4) and (18, 14), this algorithm begins with (8, 4) and makes a temporal query, "*/name*". In the reverse order, it meets the *item* node, and the *item* node links up the temporal query

```
Input: PPS, a standard XPath expression, and ($Pr_s$, $Po_s$) of the target node of the XPath expression
Output: a safe rewritten query
BEGIN
1. temp_query := null string;
2. for reverse order of the standard XPath {
3.     if node is not * and $                         // $ means "//" axis
4.        temp_query := concatenate("/node", temp_query);
5.     else if node is * {
6.        find ($Pr_{former\_node}$, $Po_{former\_node}$, $LEVEL_{former\_node}$) of the former node of "*";
7.        find ($Pr_{parent}$, $Po_{parent}$, $LEVEL_{parent}$) value set of the parent nodes of the former node;
8.        for each ($Pr_{parent}$, $Po_{parent}$, $LEVEL_{parent}$) value
9.           if (($Pr_{parent} < Pr_{former\_node}$) and ($Po_{parent} > Po_{former\_node}$) and ($LEVEL_{former\_node} - LEVEL_{parent} = 1$))
10.             temp_query := concatenate(parent, temp_query); }
11.    else if node is $ {                            // $ means "//" axis
12.       find ($Pr_{former\_node}$, $Po_{former\_node}$, $LEVEL_{former\_node}$) of the former node of "//";
13.       latter_node := get the next node of "//" node;
14.       if latter_node is empty node {
15.          find and concatenate each parent node continuously into temp_query untill root node;
16.          break; }
17.       else {
18.          find ($Pr_{latter\_node}$, $Po_{latter\_node}$, $LEVEL_{latter\_node}$) value set of the latter node;
19.          for each ($Pr_{latter\_node}$, $Po_{latter\_node}$, $LEVEL_{latter\_node}$) value
20.          if (($Pr_{latter\_node} < Pr_{former\_node}$) and ($Po_{latter\_node} > Po_{former\_node}$))
21.             if ($LEVEL_{former\_node} - LEVEL_{latter\_node} = 1$)
22.                pass and break;
23.             else if ($LEVEL_{former\_node} - LEVEL_{latter\_node} = 2$) {
24.                convert "//" into "/*" and find an actual node of *;
25.                temp_query := concatenate(actual_node, temp_query);
26.                break; }
27.             else {
28.                find and concatenate each parent node continuously into temp_query before latter_node;
29.                break; }
30.       }   // line 17
31.    }      // line 11
32.}         // line 2
33.return safe_query := temp_query;
END
```

Fig. 9. The handling *Q* INTERSECT *ACR* algorithm.

(*i.e.*, */item/name*) (Lines (3)-(4) in Fig. 9). If this algorithm meets the "*" node, it gets the former node (*i.e.*, *item* node (3, 8)). Then it finds one among many parent nodes of the former node by $PRE_{parent} < PRE_{former\_node}$, $POST_{parent} > POST_{former\_node}$, and $|LEVEL_{former\_node} - LEVEL_{parent}| = 1$. After which, the parent node links up the temporal query (*i.e., /asia/item/name*) (Lines (5)-(10) in Fig. 9). If this algorithm meets the "//" node, it gets the former node (*i.e., regions* node (1, 20)) and the latter node. If the latter node is an empty node, this algorithm finds and links up each parent node into the temporal query until the root node (*i.e., /site/regions/aisa/item/name*) (Lines (14)-(16) in Fig. 9). If the latter node exists, this algorithm compares each LEVEL value between the former node and the latter node. If $|LEVEL_{former\_node} - LEVEL_{latter\_node}| = 1$, the two nodes have a parent-child relation. Thus, "//" is converted into "/" (Lines (21)-(22) in Fig. 9). If $|LEVEL_{former\_node} - LEVEL_{latter\_node}| = 2$, this algorithm converts "//" into "/*" and finds an actual node of "*" (Lines (23)-(26) in Fig. 9). If $|LEVEL_{former\_node} - LEVEL_{latter\_node}| > 2$, then the two nodes have an ancestor-descendant relation. Thus, this algorithm finds and links up each parent node into temp_query continuously before the latter node (Lines (27)-(29) in Fig. 9).

4.4.3 Handling predicates

A predicate is a qualifying expression that is used to select a subset of the nodes of a valid XML instance. Therefore, the predicates of *ACRs* or a query is appended to a safe rewritten query. There are three cases.

Case 1: Query with predicates vs. *ACRs* without predicates: A safe rewritten query is the same case where the predicates are not used in a query or *ACRs* because the scope of a query without predicates includes the scope of a query with predicates.
Case 2: Query without predicates vs. *ACRs* with predicates: Although both positive and negative *ACRs* are the *ANCESTOR ACRs* for a query of Type 1, the query is affected by any step with predicates in *ACRs*, which filters the selected nodes. Thus, the query should be transformed into a safe query rather than be rejected.
Case 3: Query with predicates vs. *ACR* with predicates: It is similar to Case 2.

Let us extend the *Q INTERSECT ACR* algorithm to handle predicates. Recall a query Q_2 and two *ACRs*, R_3 and R_6. By the *ACR*-F, R_3 is a *DESCENDANT* positive *ACR* of Q_2, while R_6 is a *DESCENDANT* negative *ACR* which is Type 4.

Q_2: *//open_auction[@id<100]*
R_3: *//open_auction[quantity]/seller*
R_6: */site/*/open_auction[@id>50]/seller[@person="chang"]*

A *standard* XPath expression is R_3 between Q_2 and R_3, and a *standard* XPath expression is R_6 between Q_2 and R_6. Thus, each immediate rewritten query is as follows:

Immediate Q_2 INTERSECT R_3: */site/open_auctions/open_auction//seller*
Immediate Q_2 INTERSECT R_6: */site/open_auctions/open_auction//seller*

By the QA (see section 4.2), the QE obtains a hash table of each predicate's content and position information of the query. Thus, the extended algorithm is to append the predicates of both *ACR* and *Q* to the immediate rewritten query.

The predicates of Q_2 (*i.e.*, (30, @id, <, 100)) and R_3 (*i.e.*, (30, quantity, ,)) are appended to an immediate Q_2 INTERSECT R_3.

Q_2 INTERSECT R_3: */site/open_auctions//open_auction[@id<100][quantity]/seller*

The predicates of Q_2 (*i.e.*, (30, @id, <, 100)) and R_6 (*i.e.*, (30, @id, >, 50), (33, @person, =, chang)) are appended to an immediate Q_2 INTERSECT R_6.

Q_2 INTERSECT R_6: */site/open_auctions/open_auction[@id<100 and @id>50]/seller [@person="chang"]*

The final rewritten safe query Q_2' is as follows:

Q_2' = (Q_2 INTERSECT R_3) EXCEPT (Q_2 INTERSECT R_6)
= (*/site/open_auctions//open_auction[@id<100][quantity]/seller*) EXCEPT
(*/site/open_auctions/open_auction[@id<100 and @id>50]seller[@person
="chang"]*)

5. CORRECTNESS

The goal of the *SQ*-Filter system is to transform an unsafe query into a safe one as shown in section 4.4.

Q' = Q INTERSECT ($ACRs^+_Q$ EXCEPT ACR^-s_Q)
= (Q INTERSECT $ACRs^+_Q$) EXCEPT (Q INTERSECT ACR^-s_Q),

where

$$ACRs^+_Q \subseteq ACRs^+, ACR^-s_Q \subseteq ACR^-s. \quad (5)$$

The (Q INTERSECT $ACRs^+_Q$) set operation excludes unauthorized parts from a query. The (Q INTERSECT ACR^-s_Q) set operation finds the access-denied parts of the query. Although the semantics of each result query is different, both set operations are the same from the process method point of view.

We defined a symbol "$\cap$" between two XPath expressions, p_i and p_j.

Definition 8 [$p_i \cap p_j$] $p_i \cap p_j$ is the XPath expression(s) representing the nodes specified by p_i and p_j. If the preorder value of p_i is less than that of p_j and the postorder value of p_i is greater than that of p_j, then

$$Pr_{pi\cap pj} = Pr_{pj},\ Po_{pi\cap pj} = Po_{pj}.$$

If the preorder value of p_i is equal to that of p_j and the postorder value of p_i is equal to that of p_j, then

$$Pr_{pi\cap pj} = Pr_{pj} = Pr_{pi},\ Po_{pi\cap pj} = Po_{pj} = Po_{pi}.$$

Theorem 2 The *QUERY EXECUTOR* algorithm always generates the correct and safe rewritten query when Q and ACR are limited to *XPath* expressions without predicates.

Proof: Let Q and ACR be a query and an access control rule, respectively. By Definition 8, $Q \cap ACR$ is the XPath expression representing the nodes specified by Q and ACR. Since each target node of a DTD structure has a unique path, generating the correct and safe rewritten query is as follows:

$$Pr_{Q'} = Pr_{Q\cap ACR},\ Po_{Q'} = Po_{Q\cap ACR}.$$

By the ACR-F, as shown in section 4.3, an *ACR* for *Q* is one out of *ANCESTOR ACR*, *DESCENDANT ACR*, or *SELF ACR*. The proof follows a case-by-case analysis of each of them.

By Definition 5, *ANCESTOR ACR*, $Pr_Q > Pr_{ACR}$, $Po_Q < Po_{ACR}$, $Pr_{Q'} = Pr_Q$, and $Po_{Q'} = Po_Q$, and by Definition 8, $Pr_{Q \cap ACR} = Pr_Q$, $Po_{Q \cap ACR} = Po_Q$, then $Pr_{Q'} = Pr_{Q \cap ACR}$, $Po_{Q'} = Po_{Q \cap ACR}$.

By Definition 6, *DESCENDANT ACR*, $Pr_Q < Pr_{ACR}$, $Po_Q > Po_{ACR}$, $Pr_{Q'} = Pr_{ACR}$, $Po_{Q'} = Pr_{ACR}$, and by Definition 8, $Pr_{Q \cap ACR} = Pr_{ACR}$, $Po_{Q \cap ACR} = Po_{ACR}$, then $Pr_{Q'} = Pr_{Q \cap ACR}$, $Po_{Q'} = Po_{Q \cap ACR}$.

By Definition 7, *SELF ACR*, $Pr_{Q'} = Pr_Q = Pr_{ACR}$, $Po_{Q'} = Po_Q = Po_{ACR}$, and by Definition 8, $Pr_{Q \cap ACR} = Pr_Q$ (or Pr_{ACR}), $Po_{Q \cap ACR} = Po_Q$ (or Pr_{ACR}) , then $Pr_{Q'} = Pr_{Q \cap ACR}$, $Po_{Q'} = Po_{Q \cap ACR}$. ❑

Theorem 3 The *QUERY EXECUTOR* algorithm always generates the correct and safe modified query when *Q* and *ACR* are *XPath* expressions with predicates.

Proof: For an XPath expression (*pp*) with predicates, we removed the predicates to construct an XPath expression (*p*) without predicates. The special relation is as follows:

$$Pr_{pp} = Pr_p,\ Po_{pp} = Po_p,\ pp \cap p = pp. \tag{6}$$

Let a query with predicates, the query without predicates, an access control rule with predicates, and the access control rule without predicates be *Q*, *Q**, *ACR*, and *ACR**, respectively. Then we have

$$Pr_Q = Pr_{Q^*},\ Po_Q = Po_{Q^*},\ Q \cap Q^* = Q, \tag{7}$$
$$Pr_{ACR} = Pr_{ACR^*},\ Po_{ACR} = Po_{ACR^*},\ ACR \cap ACR^* = ACR. \tag{8}$$

Let Q' (= ($Q \cap ACR$)) and $Q^{*\prime}$ (= ($Q^* \cap ACR^*$)) be the output of QE. By Eq. (6), $Q' = Q' \cap Q^{*\prime}$. Thus, $Q' = Q \cap ACR \cap Q^* \cap ACR^*$. By Eqs. (7) and (8), $Q' = Q \cap ACR$. ❑

6. EXPERIMENTS

We implemented the *SQ*-Filter and the *SQ*-NFA (combining *SQ*-Filter with the NFA technique) in the Java programming language. And then we compared the performance of them with the QFilter [11] according to syntactic data sets generated by the publicly available XMark [21] for each experiment[1].

6.1 Experiment I: Correctness of Detecting Rejection Queries

The many queries on XML data have path expressions with "//" axes. The shared NFA method without the metadata of the DTD, the process of rewriting queries may be incorrect because more states are being traversed to process "*" and "//". To demonstrate this fact we made 20 intentional rejection XPath queries for each query type, and actually measured the number of filtering the rejection queries. Rejection query is a query that is always denied. In Fig. 10, we show that though the QFilter's rate of filtering rejection queries starting with the "//" axis is 0%, but on the other hand, the *SQ*-Filter and the *SQ*-

[1] The experiments were performed on a Pentium IV 2.66GHz, an MS-Windows XP and 1 GB main memory.

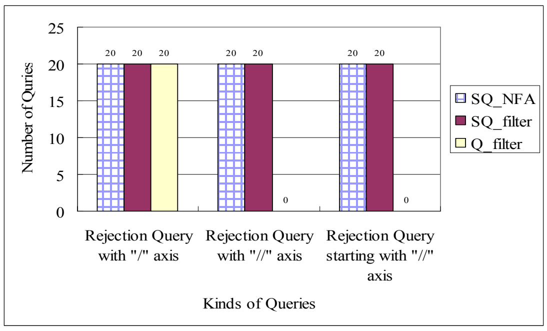


Fig. 10. The number of rejecting prohibited queries corresponding to various query types.

NFA completely filter the rejection queries. Without order information among elements, if a user query contains //-child and a shared NFA does not contain /-child or //-child state, the navigation of the shared NFA runs to each final state. If answer model is "*Answer -as-nodes*", the query is rejected. However, if answer model is "*Answer-as-sub-trees*", the QFilter appends //-child to each final state. As a result, the output of the query Q_2 may be an incorrectly rewritten query as follows:

((/site/regions/* /item[@location="LA"]//open_auction[@id<100]) UNION
(/site/people/person[name="chang"]// open_auction[@id<100]) UNION
(/site/open_auctions/ open_auction[@id<100]) UNION
(//open_auction[quantity] [@id<100]/seller))
EXCEPT
((/site/regions/* /item/payment// open_auction[@id<100]) UNION
(/site/people/person/creditcard// open_auction[@id<100]) UNION
(/site/*/open_auction[@id<100and id>50]))

6.2 Experiment II: Estimating the Average Processing Time Against No Predicates

Second, we made 100 XPath queries for each query type. In section 3.3, for the strict data security, we used "most specific precedence" for the propagation policy, and "closed policy" for decision policy. In Definition 1, it is impossible for any node, which is not in the scope of positive *ACRs*, to have negative *ACRs*. For each query, so, we made 4 positive *ACRs* at higher level nodes and 8 negative *ACRs* at lower level nodes randomly. And then we measured the average processing time for the output (rejection, rewritten query) of the *SQ*-Filter, *SQ*-NFA, and QFilter.

Before estimating the average processing time, we measured the speed of each filter construction. The *SQ*-Filter or the *SQ*-NFA construction time refers to the hash table generation time of a DTD as shown in Fig. 1 (c). The QFilter construction time means two shared NFA generation time (*i.e.*, negative and positive *ACRs*). Fig. 11 shows each construction time. From this, the metadata of a DTD has minimal overhead.

In case of queries with only "/" axes as shown in Fig. 12, the *SQ*-Filter, *SQ*-NFA and QFilter make little difference. In case of queries with "//" axis, the *SQ*-Filter and the *SQ*-NFA can rewrite queries faster than the QFilter which includes the exhaustive navigation of NFA.

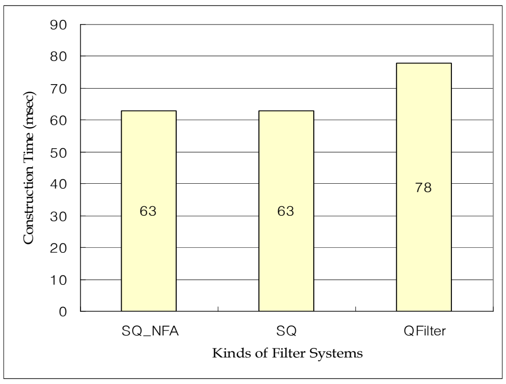


Fig. 11. The construction time of each filter system.

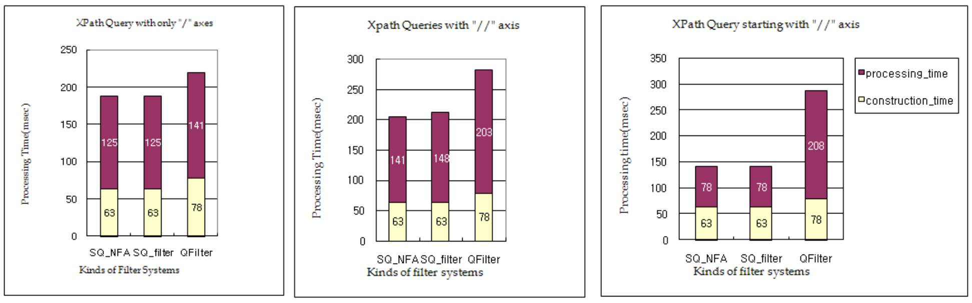


Fig. 12. The processing time of random 100 queries with 12 *ACRs* for each query type.

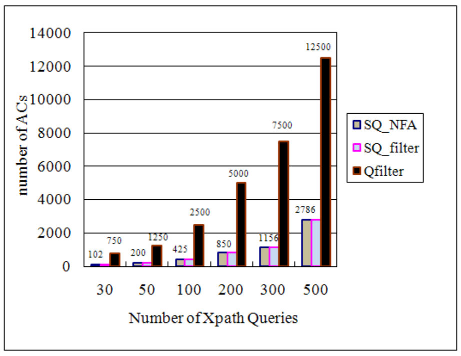


Fig. 13. The total number of *ACRs* used.

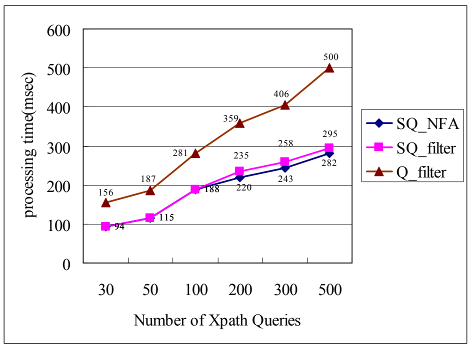


Fig. 14. The processing time of the security check on XPath queries.

Third, we measured the total number of *ACRs* used, and the average processing time per 30, 50, 100, 200, 300, and 500 random queries. For the verification of the *ACR*-F algorithm which eliminates unnecessary *ACRs*, we made more *ACRs* (7 positive and 18 negative) for each query. The QFilter uses all *ACRs* per each query, so the total number of *ACRs* is 750 per 30 queries. By the *ACR*-F in section 4.3, however, the *SQ*-Filter and the *SQ*-NFA use 102 *ACRs* per 30 queries as shown in Fig. 13. By the QE using the PPS metadata as shown in Fig. 1 (c), the *SQ*-Filter and the *SQ*-NFA can rewrite queries with the "*" wildcard and the "//" axis faster than the QFilter. The result is shown in Fig. 14. Each processing time includes the construction time as shown in Fig. 11.

6.3 Experiment III: Handling Predicates

We measured the processing time of handing predicates for the outputs of the *SQ*-Filter system. The result is shown in Fig. 15. Each processing time excludes the construction time as shown in Fig. 11. From this, we can also see that the simple handling predicates method does not account for much.

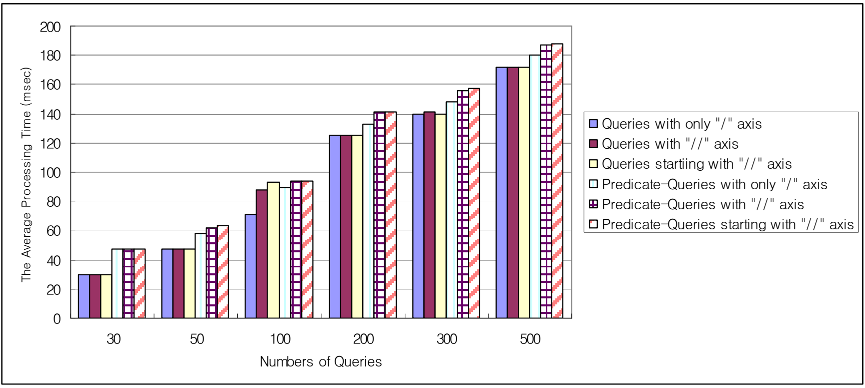


Fig. 15. The processing time of handing predicates in the *SQ*-Filter system.

7. CONCLUSION

The *SQ*-Filter system for XML access control enforcement described in this paper exploits the tree properties encoded in the PRE/POST plane to eliminate unnecessary access control rules for a user query and to reject unauthorized queries ahead of rewriting. In addition, the *SQ*-Filter system exploits the simple hash tree of a DTD to find an actual node of a "*" node and a parent node of a node with the descendant-or-self axis ("//"), and to rewrite an unsafe query into a safe one by receiving help from operations combining the two sets of node. By the *QUERY ANALYZER* component and the *Prune_TN algorithm* we could completely filter the rejection queries. By the *ACR FINDER* component we could eliminate all the unnecessary *ACRs* for a query. By the *QUERY-EXECUTOR* component we could make a rewritten safe query by extending/eliminating query tree nodes and combining a set of nodes by the set operations.

Our mechanism is a pre-processing one, which is independent from the underlying XML database engine. Thus, the techniques proposed in this paper could be built on top of any XML DBMS. In the future, we will think over complex predicate optimization. Since our approach is not entirely suitable for system specification, we plan to extend our techniques to handle a dynamic well-formed XML.

REFERENCES

1. Extensible Markup Language (XML) 1.0, 2nd ed., http://www.w3.org/TR/2000/REC-xml-20001006.
2. E. Damiani, S. Vimercati, S. Paraboachk, and P. Samarati, "Design and implementation of access control processor for XML documents," *Computer Network*, Vol. 33, 2000, pp. 59-75.
3. E. Damiani, S. Vimercati, S. Paraboachk, and P. Samarati, "A fine-grained access control system for XML documents," *ACM Transactions on Information and System Security*, Vol. 5, 2002, pp. 169-202.
4. E. Bertino, S. Castano, E. Ferrari, and M. Mesiti, "Specifying and enforcing access control policies for XML document sources," *WWW Journal*, Vol. 3, 2000, pp. 139-151.
5. E. Bertino, S. Castano, and E. Ferrai, "Securing XML documents with author-x," *IEEE Internet Computing*, Vol. 5, 2001, pp. 21-31.
6. E. Bertino and E. Ferrari, "Secure and selective dissemination of XML documents," *ACM Transactions on Information and System Security*, Vol. 5, 2002, pp. 237-260.
7. A. Gabillon and E. Bruno, "Regulating access to XML documents," in *Proceedings of the IFIP WG11.3 Working Conference on Database Security*, 2001, pp. 299-314.
8. A. Stoica and C. Farkas, "Secure XML views," in *Proceedings of the IFIP WG11.3 Working Conference on Database Security*, 2002, pp. 133-146.
9. V. Gowadia and C. Farkas, "Tree automata for schema-level filtering of XML associations," *Journal of Research and Practice in Information Technology*, Vol. 38, 2006, pp. 109-121.
10. M. Murata, A. Tozawa, and M. Kudo, "XML access control using static analysis," *ACM Transactions on Information and System Security*, Vol. 9, 2006, pp. 292-324.
11. B. Luo, D. W. Lee, W. C. Lee, and P. Liu, "QFilter: Fine-grained run-time XML access control via NFA-based query rewriting," in *Proceedings of ACM Conference on Information and Knowledge Management*, 2004, pp. 543-552.
12. P. Ayyagari , P. Mitra , D. Lee, P. Liu, and W. C. Lee, "Incremental adaptation of XPath access control views," in *Proceedings of the 2nd ACM Symposium on Information*, *Computer and Communications Security*, 2007, pp. 105-116.
13. W. Fan, C. Y. Chan, and M. Garofalakis, "Secure XML querying with security views," in *Proceedings of ACM SIGMOD International Conference on Management of Data*, 2004, pp. 587-598.
14. W. Fan, C. Y. Chan, and M. Garofalakis, "SMOQE: A system for providing secure access to XML," in *Proceedings of International Conference on Very Large Data Bases*, 2006, pp. 1227-1230.
15. J. M. Jeon, Y. D. Chung, M. H. Kim, and Y. J. Lee, "Filtering XPath expressions for XML access control," *Computers and Security*, Vol. 23, 2004, pp. 591-605.
16. J. G. Lee, K. Y. Whang, W. S. Han, and I. Y. Song, "The dynamic predicate: integrating access control with query processing in XML databases," *The VLDB Journal*, Vol. 16, 2007, pp. 371-387.
17. E. Damiani, M. Fansi, A. Gabillon, and S. Marrara, "A general approach to securely querying XML," *Computer Standards and Interfaces*, Vol. 30, 2008, pp. 379-389.
18. XQuery 1.0 and XPath 2.0, http://www.w3.org/TR/xquery-full-text/.

19. F. Rabitti, E. Bertino, W. Kim, and D. Woelk, "A model of authorization for next-generation database systems," *ACM Transactions on Database Systems*, Vol. 126, 1991, pp. 88-131.
20. S. Mohan, A. Sengupta, Y. Wu, and J. Klinginsmith, "Access control for XML – A dynamic query rewriting approach," in *Proceedings of ACM Conference on Information and Knowledge Management*, 2005, pp. 251-252.
21. A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse, "The XML benchmark project," Technical Report INS-R0103, CWI, April 2001.
22. T. Grust, "Accelerating XPath location steps," in *Proceedings of ACM SIGMOD Conference on Management of Data*, 2002, pp. 109-120.
23. T. Grust, M. J. Rittinger, and J. Teubner, "Pathfinder: XQuery off the relational shelf," *IEEE Data Engineering Bulletin*, Vol. 31, 2008, pp. 7-14.

Changwoo Byun (邊昶又) received the B.S., M.S. and Ph.D. degrees in the Department of Computer Science and Engineering from Sogang University, Seoul, Korea, in 1999, 2001 and 2007 respectively. Since Sep. 2007, he has been working in the Department of Computer Systems and Engineering of Inha Technical College. His areas of research include data stream management system, role-based access control model, access control for distributed systems, access control for XML data, XML transaction management, and Ubiquitous security.

Seog Park (朴錫) is a Professor of Computer Science at Sogang University. He received the B.S. degree in Computer Science from Seoul National University in 1978, the M.S. and the Ph.D. degrees in Computer Science from Korea Advanced Institute of Science and Technology (KAIST) in 1980 and 1983, respectively. Since 1983, he has been working in the Department of Computer Science and Engineering, Sogang University. His major research areas are database security, real-time systems, data warehouse, digital library, multimedia database systems, role-based access control, Web database, and data stream management system. Dr. Park is a member of the IEEE Computer Society, ACM and the Korean Institute of Information Scientists and Engineers.