Computers & Security (2004) **23**, 591—605

ELSEVIER

Computers & Security

www.elsevier.com/locate/cose

Filtering XPath expressions for XML access control

Jae-Myeong Jeon[a], Yon Dohn Chung[b,*], Myoung Ho Kim[a], Yoon Joon Lee[a]

[a]*Division of Computer Science, Korea Advanced Institute of Science and Technology, 373-1, Kusung-dong, Yusung-gu, Daejon 305-701, Republic of Korea*
[b]*Department of Computer Engineering, Dongguk University, 26, 3 ga, Pil-dong, Chung-gu, Seoul 100-715, Republic of Korea*

Received 24 November 2003; revised 4 May 2004; accepted 18 June 2004

KEYWORDS
XPath;
XML;
Query processing;
Access control;
Security

Abstract XPath is a standard for specifying parts of XML documents and a suitable language for both query processing and access control of XML. In this paper, we use the XPath expression for representing user queries and access control for XML. And we propose an access-control method for XML, where we control accesses to XML documents by filtering query XPath expressions through access-control XPath expressions. For filtering the access-denied parts out of query XPath expressions, set operations (such as, intersection and difference) between the XPath expressions are essential. However, it is known that the containment problem of two XPath expressions is coNP-hard when the XPath expressions contain predicates (or branch), wildcards and descendant axes. To solve the problem, we directly search *XACT* (XML Access Control Tree) for a query XPath expression and extract the access-granted parts. The *XACT* is our proposed structure, where the edges are structural summary of XML elements and the nodes contain access-control information. We show that the query XPath expressions are successfully filtered through the *XACT* by our proposed method, and also show the performance improvement by comparing the proposed method with the previous work.

* Corresponding author.
E-mail addresses: jmjeon@dbserver.kaist.ac.kr (J.-M. Jeon), ydchung@dgu.edu (Y.D. Chung), mhkim@dbserver.kaist.ac.kr (M.H. Kim), yjlee@dbserver.kaist.ac.kr (Y.J. Lee).

0167-4048/$ - see front matter © 2004 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cose.2004.06.009

Introduction

XML (eXtensible Markup Language) (W3C, October 2000) is defined as a standard of data representation and transmission on Internet by W3C (World-Wide Web Consortium). Owing to its independence of applications, self-description and extensibility, the application areas of XML grow rapidly. And the problems of XML security are also being issued recently. Much research on securing XML document is in progress (http://www.nue.et-inf.uni-siegen.de/geuerpoll-mann/xml_security.html; OASIS, February 2003; OASIS (SAML); W3C, March 2001; W3C, February 2001; W3C (XKMS), March 2001; W3C, February 2002). Especially, among XML security issues, the digital cryptography and the server-side access control have received much attention. The digital cryptography deals with how to securely encrypt XML, how to transmit them via WWW and how to decrypt them to the authorized users (OASIS (SAML);W3C, March 2001; W3C, February 2002; W3C (XKMS), March 2001; W3C, February 2001). On the other hand, the server-side access control provides users the domain-based access control of XML according to their access rights on XML documents (Bertino et al., 2001; Damiani et al., 2000, 2002; Kudo and Hada, 2000; OASIS, February 2003). When the server-side XML security provides an element-level access control with the multi-user support, it takes up much time. In this paper, we propose a strong and efficient server-side XML access-control method that is applicable to complicate XPath expressions and reduces access-control processing time for queries.

XPath (W3C, 2003) is a standard for addressing parts of XML documents. With the XPath expressions, we can locate specific patterns at a particular context, and perform additional operations on the context of the located nodes. With this regard, the XPath expressions are considered suitable for both query processing and access control of XML data (Damiani et al., 2000; Jajodia et al., 2001; Kudo and Hada, 2000; OASIS, February 2003). In this paper, we use the XPath for representing user queries and access-control information of XML documents. (We use a 'query *XPE*' (XPath Expression) to denote a query represented by an XPath expression, and an 'access-control *XPE*' to denote an access-control information represented by an XPath expression.)

We assume the following environment:

- the user submits a query *XPE* to the server
- the access-control information (in the server) is represented (and also stored) in *XPE*
- the server compares the query *XPE* (by the user) with the access-control *XPE*, and produces a set of *XPEs* which represents only authorized parts of the XML document
- the result *XPEs* are evaluated in the server, and the result data are delivered to the user

In this paper, we propose a strong and efficient access-control method that produces the access-granted *XPEs* from the query *XPE* by using the access-control *XPEs*. The method builds XML Access Control Tree (*XACT*), where the edges are structural summary of XML elements and the nodes contain access-control *XPEs*. When a query *XPE* is submitted, the method compares the query and the access-control *XPEs* in the *XACT*, and produces the *Result XPEs*.

The remainder of this paper is organized as follows. In the next section, we describe the background of this work, including XML, XPath, access-control policies of XML documents and related work on XML access control and XPath expression properties. In section 'Problem definition', we define the problem that takes place when we extract access-granted *XPEs* from query and access-control *XPEs*. We describe the *XACT* structure and propose an XML access-control method that filters access-granted parts from the user's query *XPE* using the *XACT* in section 'The proposed access-control method'. We analyze the performance and compare it with some previous work in section 'Performance analysis'. In last section 'conclusions', we conclude this paper.

Background

XML

An XML document is composed of a sequence of nested elements and each element contains sub-elements, texts, or attributes. The sub-elements and texts exist between start-element '<tag-name>and end-element </tag-name>', and the attributes are defined inside the start-element '<tag-name>'. XML tags represent formatting rules, data description or data relationship of an XML document. XML allows the tailoring of the tags to describe the specific structure of data, which supports a flexible structure to the data. Fig. 1 shows an example of XML document and its graph representation which shows structural properties of the XML document.

```
<order num="b392-323">
  <customer_info>
    <name>Jeon</name>
    <phone>016-211-2815</phone>
    <addr>
      <city>Daejeon</city>
      <zipcode>301-754</zipcode>
    </addr>
    <credit_card type="Master"
      exp_date="2006-12"/>
  </customer_info>
  <order_info>
    <title>DATABASE SECURITY</title>
    <publisher>Addison Wesley</publisher>
    <price>25.00</price>
    <addr>
      <city>Daejeon</city>
      <zipcode>301-754</zipcode>
    </addr>
  </order_info>
  <order_info>
    <title>Data on the web</title>
    <ISBN>1-55860-622</ISBN>
    <price>39.95</price>
    <addr>
      <city>Seoul</city>
      <zipcode>435-651</zipcode>
    </addr>
  </order_info>
</order>
```

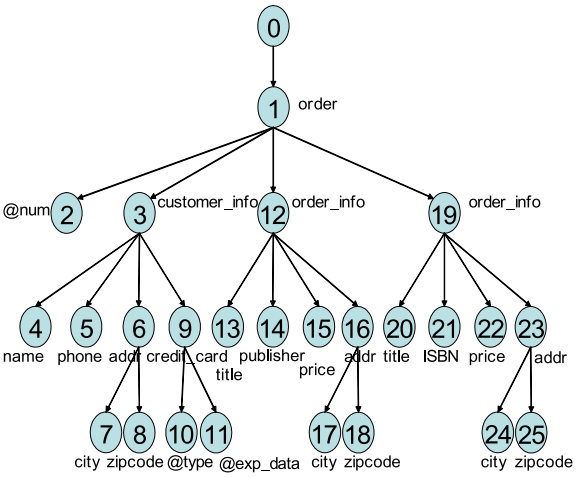


Figure 1 An XML document and its tree representation.

XPath

The primary purpose of XPath is to address parts of an XML document (W3C, 2003). XPath represents the location of data in an XML document correctly and efficiently, which makes XPath a suitable language for both XML query (XQuery W3C, May 2003; XSLT W3C, November 1999, etc.) and XML access control (Damiani et al., 2000; Kudo and Hada, 2000). XPath Expression (*XPE*) is a structural pattern which can be matched to nodes in an XML document tree. *XPE* is a sequence of location steps which specifies elements and their relationships by several operators: the parent–child('/') and ancestor–descendant('//') operators represent a single step and one or more steps, respectively; wildcard operator('*') denotes any element name; and so on. Also, each location step can contain predicates (enclosed between '[' and ']') to refine the set of selected nodes. The order and position among sibling nodes can be predicates of *XPE* but we do not consider them because they are scarcely used as a predicate in XML data and documents. In this paper, we use *XPEs* for describing query, access-granted and access-denied parts of an XML document.

Definition 1. Let p_i and p_j be *XPEs*, and $n\{p_i\}$ denote a set of target nodes specified by p_i in an XML document tree

- $p_i \cup p_j$ is the *XPE* (or *XPEs*) representing the nodes specified by p_i or p_j, that is, $n\{p_i \cup p_j\} = n\{p_i\} \cup n\{p_j\}$
- $p_i \cap p_j$ is the *XPE* (or *XPEs*) representing the nodes specified by p_i and p_j, that is, $n\{p_i \cap p_j\} = n\{p_i\} \cap n\{p_j\}$
- $p_i - p_j$ is the *XPE* (or *XPEs*) representing the nodes specified by p_i but not p_j, that is, $n\{p_i - p_j\} = n\{p_i\} - n\{p_j\}$

Example 1. Let p_i = "order/customer_info//" and p_j = "order/customer_info/name", where p_i and p_j are *XPEs* for the XML document in Fig. 1. Then the result *XPE* of the set operations between p_i and p_j are as follows:

(a) $p_i \cup p_j$ = "order/customer_info//"
(b) $p_i \cap p_j$ = "order/customer_info/name"
(c) $p_i - p_j$ = "order/customer_info/phone"; "order/customer_info/addr//"; "order/customer_info/credit_card//"

Access-control policies

Some access-control policies are studied in Jajodia et al. (2001). The *propagation policy* determines whether the access control to a node is propagated to its descendants or not. The *conflict resolution policy* resolves the situation that a node is defined as 'access-granted' and 'access-denied' at the same time. And the *decision policy* determines when a node is neither access-granted nor access-denied. For exact and efficient access control of XML documents, the policies are applied simultaneously. In this paper, we use "most specific overrides" for the

propagation policy that prefers access control to the most specific node of an XML document tree, and use ''denials take precedence'' for the conflict resolution policy to keep data in safety. For the decision policy, we use the ''closed'' for the same reason, that is, we deny a query for a node whose access-control information is not defined.

Related work

Many approaches have been studied for XML document security. Some of them are as follows: Damiani et al. (2000) and Bertino et al. (2001) proposed an early model of XML access control, which builds a DOM tree (Document Object Model) for a query *XPE* and labels the tree nodes as ''*access-granted*'' (or ''*access-denied*'') sign according to the access rights of a user. The access-denied part of the tree is pruned and the access-granted part is supplied for the query. Yu et al. (2002) have improved the approach of Damiani et al. (2000) and Bertino et al. (2001) from the viewpoint of access time for a query. The method builds optimal access-control look-up data for each user and access type. When a query is submitted, it refers to the access-control data and selects the access-granted part of the query. However, a query *XPE* with '//' or '*' is not applicable to the method because it requires the complete path from the root to a node. Kudo and Hada (2000) proposed a provisional authorization model that compares a query with XACL (XML Access Control Language) for authorization of the query. The XACL includes security data such as authorization, non-repudiation, confidentiality, and audit trail. But the model does not deal with the coNP-hardness problem that may occur when comparing between the query *XPE* and *XPE* in the XACL. Cho et al. (2002, 2003) focused on finding optimal and safe rewritings of queries that ensure secure evaluation given a DTD graph of XML document.

Murata et al. (2003) used automata to represent and compare queries, access-control policies and schemas. It created automata for query, access control and schema, and produced results by intersection of the automata. However, as they mentioned, the method has some limitations processing predicates and recursive queries. Blyth et al. (2003) identified and quantified the ways where XML processing can be subverted into leaking, corrupting or denying XML documents. And Damiani et al. (2002) presented a fine-grained authorization model for SOAP requests.

The research on comparing *XPEs* is carried out in the field of XML query processing. Benedikt et al. (2003) studied the closure properties (union, intersection and complementation) between fragments of *XPEs*, which showed that the union and intersection are closed but the complementation is not. Miklau and Suciu (2002) investigated containment and equivalence problems between *XPEs* which contain branching, label wildcards and descendant relationships. It showed that the containment problem between *XPEs* is coNP-hardness in general and PTIME in special cases (Miklau and Suciu, 2002).

Problem definition

Let Θ and Δ be access-granted and access-denied *XPEs* for an XML document, and Π be a query *XPE*. Is the following operation computable?

$$\text{Result } XPE = (\Theta \cap \Pi) - \Delta$$

It is difficult to extract the *Result XPE* in the above set operation when they have partially common set of data. Furthermore, the *Result XPE* depends on the structure of target documents. Therefore, in order to compute the *Result XPE*, we must be informed of the structure of the document in advance. Miklau and Suciu (2002) proved that the containment problem for *XPE* $\mathbf{P}^{\{[],*,//\}}$ expression is coNP-hard, where [], * and // denote predicate (or branch), wildcard and descendant axis, respectively. Hence, the intersection and difference between *XPEs* are also coNP-hard. To solve the problem, Miklau and Suciu (2002) suggest 'canonical models' that replace descendant axis with 'z'-labeled nodes by referring the XML document. But the model is not applicable to the set operations between *XPEs*, because the difference of two *XPEs* with 'z'-labeled nodes is not closed (Benedikt et al., 2003). Damiani et al. (2000) extract the access-granted data from the DOM tree denoted by access-granted *XPEs* and processes a query *XPE* based on the extracted data. However, when queries access several XML documents on small scales, the method is not efficient.

In this paper, we propose a method that produces the *Result XPE* of ''$(\Theta \cap \Pi) - \Delta$'' without accessing XML documents or DOM trees. For efficient processing of set operations between the *XPEs*, we define a structure, called the XML Access Control Tree (*XACT*), where the edges are structural summary of XML elements and the nodes contain access-control information. When a query *XPE* is submitted, our method evaluates the access rights of its target nodes in the *XACT*. If a node is granted, which means the node is an object node of both the query and the access-granted *XPEs*, the

path of the node is appended to the *Result XPE*. If no node is granted, the result of authorization becomes null.

The proposed access-control method

In this section, we propose an efficient access-control method that compares query *XPEs* with access-control *XPEs*, and extracts access-granted *XPEs* from the query *XPEs*. Fig. 2 shows an architectural overview of the proposed method. First, '*XACT Constructor*' creates and updates an *XML Access Control Tree* (*XACT*) that consists of edges (which are structural summary of an XML document) and nodes (which contain access-control data). The access-control data are converted from an *Access Control List* (*ACL*) and stored into the nodes each of which corresponds to the *XPEs* of the *ACL*. Details about '*XACT Constructor*' are described in the following section. In the figure, we show in 'node 10' that an *XACT* node contains access-control data. Second, '*Query Filter*' compares *Input Query* with the *XACT*. When the *XPE* of *Input Query* finds its corresponding node in the *XACT*, 'ID' and 'access mode' of the *Input Query* are compared with the access-control data of the node. If the *Input Query* is granted, the path of the node(s), and access information ('ID' and 'access mode') of the query are appended to the *Authorized Query*. Otherwise, the *Input Query* is ignored. Details about '*Query Filter*' are described in section 'The access control algorithm using the *XACT*'. Then the *Authorized Query* is delivered to '*Query Processor*' instead of the *ACL* and the original *Input Query*.

The *XACT* construction

We construct the *XACT* to compute the *result XPE* of "$(\Theta \cap \Pi) - \Delta$", where Θ and Δ are access-granted and access-denied *XPEs* and Π is a query *XPE*. First of all, we store the access-control information "$(\Theta - \Delta)$" in *XACT* nodes to which they are matched. When a query *XPE* is submitted, we search the *XACT* nodes which are matched by the query *XPE* and are *accessible* to the user. Then the result nodes become "$(\Theta - \Delta) \cap \Pi$".

The '*XACT Constructor*' generates *XACT* from an XML document and an *Access Control List*. The *Access Control List* is defined as follows.

Definition 2. The *Access Control List* (*ACL*) for an XML document is a set of triples of the form $\langle o, s, a \rangle$, where $o \in O$, $s \in S$, $a \in A$ such that

- O is a set of *XPEs*
- S is a set of subjects i.e., $S = \{ID_1, ID_2, ID_3, \ldots\}$, where ID_i is the identifier of a user or a group of users

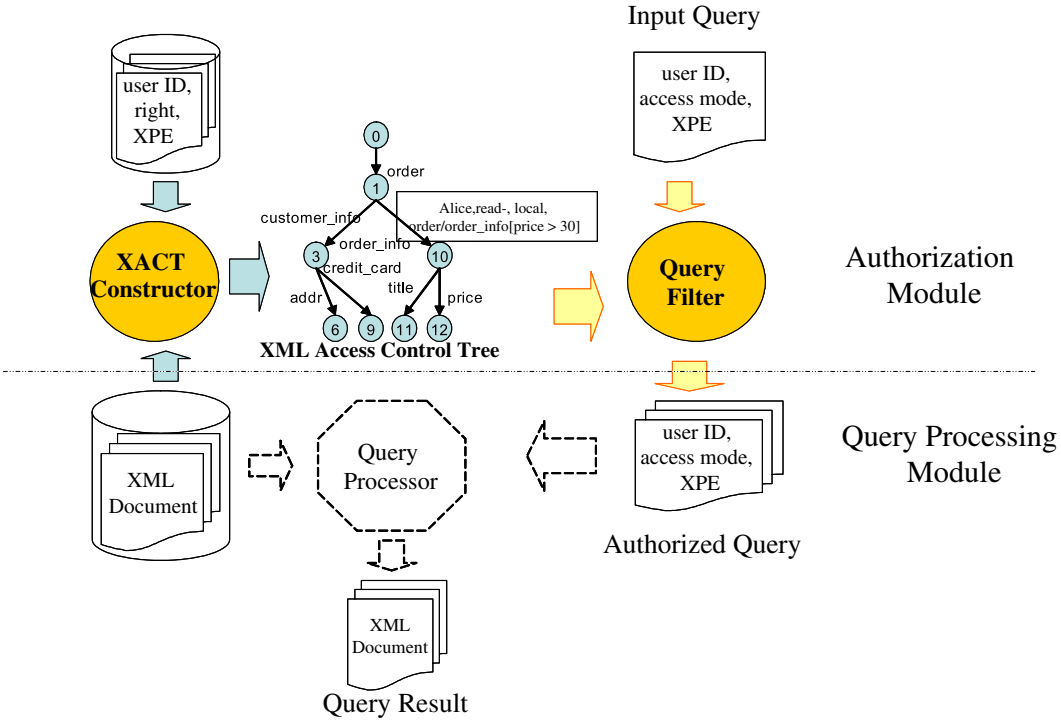


Figure 2 The architecture of the proposed method.

- A is a set of authorized modes for the *XPEs* i.e., $A = \{\text{read}+, \text{read}-, \text{write}+, \text{write}-\}$

The *ACL* is stored into each node of *XACT* which is specified by the *XPE*. An *XPE* can be divided into a path to a node and its predicates which refine the node. Here, we store only the predicates of the *XPE* to the matched node instead of *XPE*, because the path of the node represents the location where the *XPE* should be stored. An *XPE* terminated with '//' or '*' (e.g., order/order_info//or order/order_info/*) means that the scope of access control is 'recursive' or 'child', respectively.

Definition 3. The *XML Access Control Tree* (*XACT*) consists of edges E, which are elements of an XML document, and nodes N each of which contains a quadruple $\langle s, a, c, f\rangle$ where $s \in S$, $a \in A$, $c \in C$, $f \in F$ such that

- Each node n (except the root) has an edge e that points to the node, where $n \in N$ and $e \in E$
- The path $path(n)$ is a sequence of edges from the root to node n, and there is no duplicate path in the *XACT* i.e, $path(n_i) \neq path(n_j)$ if $n_i \neq n_j$
- S is a set of subjects i.e., $S = \{ID_1, ID_2, ID_3, \ldots\}$, where ID_i is the identifier of a user or a group of users
- A is a set of authorized modes for the *XPEs* i.e., $A = \{\text{read}+, \text{read}-, \text{write}+, \text{write}-\}$
- C is the scope of authorization to the node n for subject S i.e., $C = \{\text{local, child, recursive}\}$, where local, child, and recursive denote that an authorized mode is applied to ''node n'', ''child nodes of n'', and ''node n and all its descendants'', respectively
- F is a set of predicates of the *XPE* to node n, which is the condition of node n to be accessible by subject S e.g., $F = \{\text{price} > 30, \text{name} = \text{''Bob''}\}$

The *XACT* consists of a reduced XML tree and access-control data. The reduced XML tree means a tree composed of elements without duplicate paths. Thus, each node of *XACT* has a unique path. Access-control data are transformed from ACL in Definition 2. The example of *XACT* is shown in Fig. 2. An edge of the *XACT* is an element of an XML document and a path is composed of a sequence of edges from the root to a node. To transform ACL to access-control data in the *XACT*, we search a sequence of edges in the *XACT* which are matched by an *XPE* in the *ACL*. If we find the sequence of *XACT* edges for the *XPE*, we transform the *ACL* in Definition 2 to access-control data in Definition 3 and store it to the node to which the most specific edge points. S and A in Definition 3 are the same as in Definition 2. C in Definition 3 is decided by the scope of *XPE* in the *ACL*. If the *XPE* is terminated with '//' or '*', the scope C is 'recursive' or 'child', respectively. Otherwise, the scope C is 'local'. F in Definition 3 is the predicate of the *XPE* which is the condition of node n. Therefore, predicate F is adapted to the path of its target nodes and stored into them. Because the path of the *XACT* is a sequence of elements, non-element data in *XPE*, such as attributes, texts and so on, cannot be matched to any edge in the *XACT*. So, non-element data have to be transformed to some predicates of elements which contain them.

Fig. 3 shows the process of '*XACT Constructor*'. Note that the edges of the *XACT* are composed of XML elements. We omit the algorithm for building the edges because it is straightforward. After building the edges, we search edges in the *XACT* with an *XPE* in the *ACL*. When an edge is matched to the *XPE* without a predicate, the scope of the *XPE* is verified and stored to the node to which the edge points. If the *XPE* contains predicates, each predicate has to be examined whether it includes '//' or '*'. When a predicate contains '//' or '*', the predicate is transformed into a set of predicates without '//' or '*' by referring the *XACT* edges. After the examination and transformation, the set of predicates is stored to the node to which the edge points. Then, the scope of the *XPE* is verified. If the *XPE* is terminated with '//' or '*', the scope is 'recursive' or 'child', respectively. Otherwise, the scope is 'local'. After storing the access-control information, we search another edge in the *XACT*. If an edge is matched to the *XPE*, the above procedure is repeated; otherwise we search edges in the *XACT* with another *XPE* in the *ACL*. If there remains no *XPE* to search for, the '*XACT Constructor*' process ends.

Example 2. Suppose there is an *ACL* as follows:

(1) (Bob, read+, ''order//'')
(2) (Alice, read+, ''order/order_info//'')
(3) (Bob, read−, ''//credit_card/'')
(4) (Alice, read−, ''//order_info[price > 30]/addr'')

Then, the *ACL* is transformed and stored into the *XACT* (in Fig. 4) as follows: (1) and (2) are transformed like *A* and *B*, and stored in node '1' and '10', respectively. Their scopes are '*recursive*' because the *XPEs* are terminated with '//'. Then, (3) is transformed like *C* and stored in node '9'.

Algorithm *XACT*_Constructor
Input : ACL(ID, right, *XPE*), XML document
Output : *XACT*
(1) Build *XACT* edges of XML document
(2) **for** (each *XACT* node which is matched by the *XPE*)
(3) **if** (the *XPE* contains predicates)
(4) Current_node.predicate := Predicate_Search(node, *XPE*)
(5) Current_node.ID := ID
(6) Current_node.right := right
(7) **if** (the *XPE* terminated with //) Current_node.scope := 'recur'
(8) **else if** (the *XPE* terminated with *) Current_node.scope := 'child'
(9) **else** Current_node.scope := 'local'

Algorithm Predicate_Search
Input : node, *XPE*
Output : predicate
(1) **for** (each predicate)
(2) **if** (the predicate contains '//' or '*') // search exact set of predicates
(3) **for** (each path of the predicate) predicate_path := predicate_path ∪ path
(4) **else** predicate_path := predicate
(5) predicates := predicates ∪ predicate_path
(6) **return**(predicate)

Figure 3 The process of *XACT Constructor*.

Lastly, (4) is transformed like *D* and stored into node '15'. Note that, the predicate is modified based on the node's path. For example, in (4), the predicate '//order_info[price > 30]' is transformed to 'order/order_info [price > 30]' based on the path of node '15'.

A unique path from the root to an element in an XML tree is transformed into a sequence of edges in the *XACT* and duplicate paths are eliminated. Hence, an XML document and its corresponding *XACT* satisfy the following property.

Property 1. *If an XPE p without predicates matches a sequence of edges in the XACT, p also matches a set of elements in the XML document that corresponds to the set of edges in the XACT and vice versa.*

Proof. Let *edge-path*$_{XACT}$(*n*) be a sequence of edges from the root to node *n* in the *XACT*, and *element-path*$_{XML}$(*e*) be a sequence of elements from the root element to element *e*, which is corresponding to node *n*, in the XML document. By Definition 3, since each node n_i in *element-path*$_{XML}$(*e*) corresponds to an edge e_j, in *edge-path*$_{XACT}$(*n*), they have the same sequence of ancestors. Hence, the *XPE p* matches a sequence of edges leading to node *n* (i.e., *edge-path*$_{XACT}$(*n*)) in the *XACT*, *p* also matches one (or more) sequences of elements leading to element *e* (i.e., *element-path*$_{XML}$(*e*)) in the XML document, where *e* is the element corresponding to node *n*. (Note that there may be several elements *e* in the XML document for a single node *n* of the XACT because the XACT eliminates duplicate paths in the XML document.) □

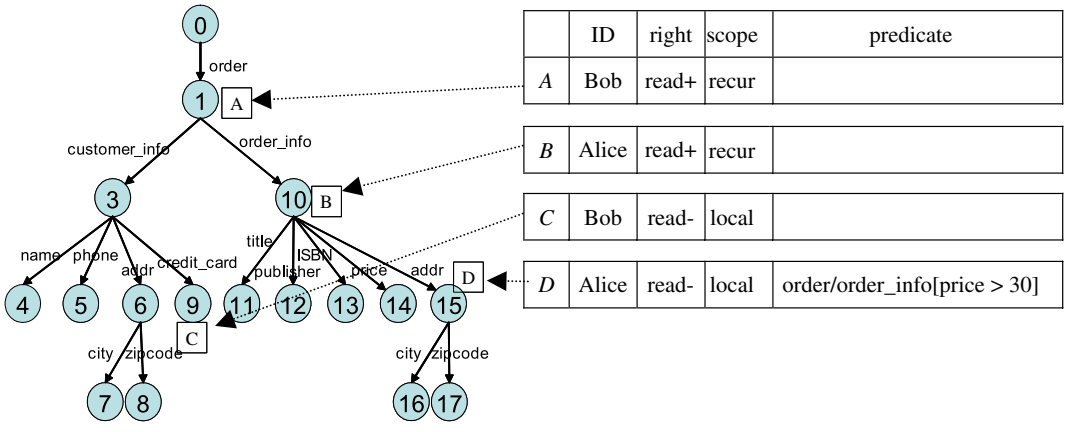


Figure 4 *XACT* and its access-control data.

However, if an *XPE* p includes predicates, the set of elements for p on an XML document might be different from the set of elements which is corresponding to the edge set for p on the *XACT*.

Property 2. *Let p be an XPE with predicates. And, let $n_{XML}\{p\}$ be the set of nodes for p in an XML document tree and $n_{XACT}\{p\}$ be the set of nodes whose incoming edges are matched by p in the corresponding XACT. Then $n_{XML}\{p\} \subseteqq n_{XACT}\{p\}$.*

Property 2 occurs when there exist two nodes that have the same sequence of ancestors and have different set of child nodes in an XML document tree. By Definition 3, one of the duplicate paths is eliminated in the XACT and the two sets of child nodes will become a set of sibling nodes in the XACT. Hence, for any node n, the sibling set of node n in the XACT includes that of node n in the XML document tree.

We show an example. An *XPE* 'order/order_info[ISBN]/publisher' does not match any node in the XML document, but it matches node '12' in the *XACT* as shown in Fig. 5. In this case, $n_{XML}\{p\} \subset n_{XACT}\{p\}$. In some other cases, $n_{XML}\{p\}$ may be equal to $n_{XACT}\{p\}$.

Now we consider a special case, where an *XPE* contains a predicate which includes a negation e.g., 'order/order_info[not(ISBN)]/publisher'. It matches node '14' in the XML document tree, but it does not match any node in the *XACT*. To prevent these errors, we treat the negation predicates in a special manner in the *Query Filter* algorithm described in Fig. 6. (In the algorithm, we do not search *XACT* nodes with an *XPE* predicate which contains 'not' operation but consider the predicate as an attribute and store it into its parent node. The predicate should be referred when an *Input Query* specifies the node.) Consequently, the set of elements for *XPEs* (with predicates) on an XML document will be the same to those on the XML document via *XACT*.

Definition 4. We call two *XPEs* p_j and p_k are *equivalent*, denoted by $P_j \equiv_{D_i} P_k$, when their sets of target nodes $n\{p_j\}$ and $n\{p_k\}$ are the same in an XML document tree D_i.

Theorem 1. $\mathbf{P} \equiv_{D_i} \mathbf{P}'$ *w.r.t the XML document D_i, where* $\mathbf{P}$ and $\mathbf{P}'$ *are sets of XPE and* $\mathbf{P}'$ *is transformed from* $\mathbf{P}$ *within the XACT.*

Proof. Let $n\{\boldsymbol{P}\}$ be a set of nodes for a set of *XPEs* $\boldsymbol{P}$ and g be a function from $\boldsymbol{P}$ to $n\{\boldsymbol{P}\}$ in the XML document tree D_i, $g(\boldsymbol{P}) = n\{\boldsymbol{P}\}$. Also, let f be a function from $\boldsymbol{P}$ to $\boldsymbol{P}'$ in the *XACT*, $f(\boldsymbol{P}) = \boldsymbol{P}'$. For $^{\forall}p$, where $p \in \boldsymbol{P}$, and its transformed *XPE* p', where $p' \in \boldsymbol{P}'$, if $g(f(p)) = g(p)$, then $f(p) = p$ and $p \equiv_{D_i} p'$. For an *XPE* p without predicates and its transformed *XPE* p', $g(p) = g(p')$ by Property 1, i.e., $f(p) = p$. Hence, $p \equiv_{D_i} p'$. However, for an *XPE* p with predicates, four different cases are possible: (1) $g(f(p)) = n\{p\}$ and $g(p) = n\{p\}$. There exists p that $f(p) = p'$ and $g(p') = g(p) = n\{p\}$. Then $p \equiv_{D_i} p'$. (2) $f(p) = \emptyset$ and $g(p) = \emptyset$. There is no node for p both in the XML document tree and in the *XACT*. Thus, $p \equiv_{D_i} p'$. (3) $g(f(p)) = \emptyset$ and $g(p) = \emptyset$. There exists p, where $f(p) = p'$, in the *XACT*, but $g(p') = \emptyset$ in the XML document node. Even though there exists a node for p in the *XACT*, the result of function g is empty, thus, $p \equiv_{D_i} p'$. (4) $f(p) = \emptyset$ and $g(p) = n\{p\}$. There is no node for p in the *XACT*, but there exists

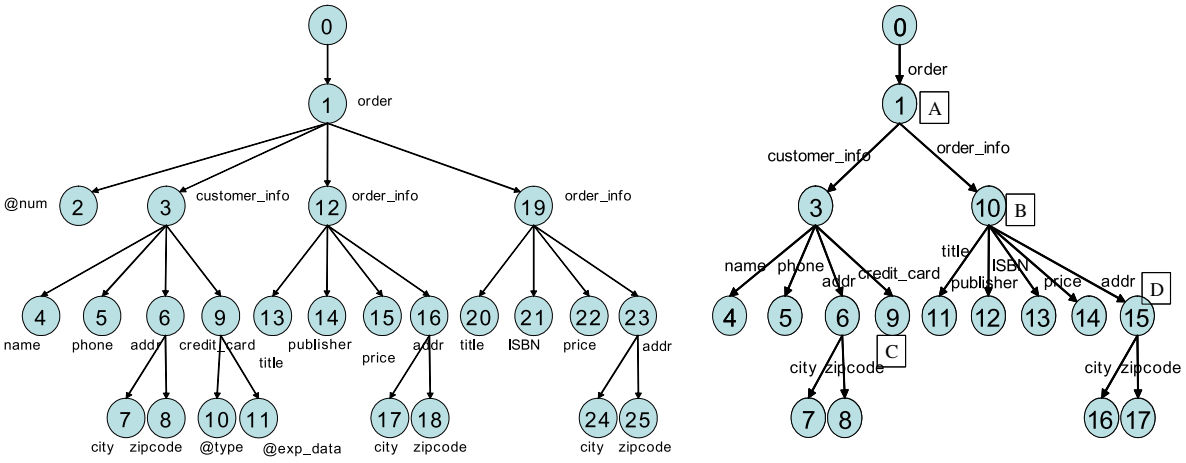


Figure 5 An XML document tree and its *XACT*.

```
Algorithm Query_Filter
Input : Input Query(ID, access mode, XPE), XACT
Output : Authorized Query(ID, access mode, XPE)
(1)  for (each node of XACT ∈ XPE)
(2)      if (current_node is access-denied) ∧ (current_node has a predicate)
(3)        ancestor_result := Ancestor_Check(current_node, ID, access_right)
(4)        if (ancestor_result = 'grant')
(5)          not_predicate := Predicate_Comparison(current_node, Input Query)
(6)          if (predicate_XPE != NULL)
(7)                predicate_XPE   := concat('not(', not_predicate, ')')
(8)      else if (current_node is access-denied)
(9)      else if (current_node is access-granted) ∨ (current_node is access-undefined)
(10)         if (current_node is access-undefined)
(11)          ancestor_result := Ancestor_Check(current_node, ID, access_right)
(12)         if (ancestor_result = 'grant') ∨ (current_node is access-granted)
(13)           if (XPE is terminated with '//' or '*')
(14)             descendant_XPE := Descendant_Check(current_node, XPE)
(15)           if (XPE contains predicate)
(16)             for (each descendant_XPE)
(17)               predicate_XPE := Predicate_Comparison(current_node,
(18)                                Query(ID, access mode, descendant_XPE))
(19)     if (the XPE contains predicate) ∧ (predicate_XPE != NULL)
(20)                Auth_XPE := Auth_XPE ∪ predicate_XPE
(21)     else if (the XPE does not contain predicate) ∧ (descendant_XPE != NULL)
(22)                Auth_XPE := Auth_XPE ∪ descendant_XPE

Algorithm Descendant_Check
Input: current_node, XPE
Output: access-granted XPE
(1)  if (XPE terminated with '//')
(2)      if (all descendants are granted) grant_XPE := XPE
(3)      else
(4)        for (each descendant of XPE)
(5)          if (descendant is granted) grant_XPE := grant_XPE ∪ descendant
(6)  else if (XPE terminated with '*')
(7)        if (all child nodes are granted) grant_XPE := XPE
(8)        else
(9)          for (each child of XPE)
(10)           if (the child is granted) grant_XPE := grant_XPE ∪ child

Algorithm Ancestor_Check
Input: current_node, ID, access_right
Output: access right of current_node
(1)  ancestor_node := parent_of_current_node
(2)  if (ancestor_node is access-granted and scope is 'child') return 'grant'
(3)  for (each ancestor_node)
(4)      if (ancestor_node is access-granted and scope is 'recur') return 'grant'
(5)      else if (ancestor_node is access-denied or scope is 'local') return 'denied'
(6)      else ancestor_node := parent_of _ancestor_node     // when ancestor_node is access-undefined

Algorithm Predicate_Comparison
Input : current_node, Input Query(ID, access mode, XPE)
Output : XPE with access-granted predicate
(1)  grant_XPE := path of current_node
(2)  for (each predicate of XPE)
(3)      if ( predicate contains '//' or '*' )
(4)            for (each path of predicate)
(5)              if(path is access-granted) predicate_path := predicate_path ∪ path
(6)      else predicate_path := current_predicate
(7)      for (each access-granted predicate_path)
(8)            node_predicate := node_predicate ∪ predicate_path
(9)      if (node_predicte = NULL) return null
(10)     grant_XPE := node_predicate ∪ grant_XPE
(11) return grant_XPE
```

Figure 6 The algorithm of *Query Filter*.

at least one node for p in the XML document tree. By Properties 1 and 2, $g(p) \subseteqq f(p)$. So, case (4) cannot occur. Therefore, for $^{\forall}p \in P$, and its transformed *XPE* $p' \in \boldsymbol{P}'$, $p \equiv_{Di} p'$. Hence, $\boldsymbol{P} \equiv_{D_i} \boldsymbol{P}'$. □

Theorem 1 says that sets of *XPEs* $\boldsymbol{P}$ and $\boldsymbol{P}'$ are equivalent w.r.t any XML document D_i. $\boldsymbol{P}'$ is a set of *XPEs* which, if applied to the *XACT*, produces the same set of nodes produced by $\boldsymbol{P}$. That is, for the XML document D_i, the set of target nodes of *XPEs* $\boldsymbol{P}$ and those of $\boldsymbol{P}'$ are the same.

Corollary 1. *The XPEs are closed under the union and intersection operations.*

Proof. see Benedikt et al. (2003). □

Corollary 2. *The XPEs are closed under the difference operation based on XACT.*

Proof. Let P_i, P_j and P_k be sets of *XPEs*, and P'_i, P'_j and P'_k be sets of *XPEs* transformed from P_i, P_j and P_k in the *XACT*, respectively. And let $n\{P'_i\}$, $n\{P'_j\}$ and $n\{P'_k\}$ be sets of *XACT* nodes specified by P'_i, P'_j and P'_k, respectively.Assume that $P_i = P_j - P_k$, then $P'_i = P'_j - P'_k$ by Theorem 1. Then we can specify $n\{P'_i\}$ by sets of *XPEs* in the *XACT* without *difference* operation (i.e., $\{P'_i\} = p'_{i1} \cup p'_{i2} \cup \cdots \cup p'_{il}$), where $n\{P'_i\} = n\{P'_j\} - n\{P'_k\}$. For $^{\forall} n\{p'_m\}$, $1 \le m \le l$, $n\{p'_m\} \subseteqq n\{p'_j\}$ and $n\{p'_m\} \cap n\{p'_k\} = \emptyset$. Also, $P'_i = p'_1 \cup p'_2 \cup \cdots \cup p'_l$ is closed by Corollary 1. Therefore, a set of *XPEs* P_i' is closed under the *difference* operation based on *XACT*. □

The access control algorithm using the *XACT*

In this section, we describe '*Query Filter*' algorithm that produces *Authorized Query* from the *Input Query* by comparing with the access-control data in the *XACT* nodes. Let Θ and Δ be sets of access-granted and access-denied *XPEs*, respectively, and Π be a set of query *XPEs*. Our goal is to solve the problem '$(\Theta \cap \Pi) - \Delta'$ and to produce *Authorized Query*. As we described in previous section, the access-control data are stored into the *XACT* nodes on which the query *XPE* is evaluated.

The algorithm of '*Query Filter*' is described in Fig. 6. We traverse the *XACT* and decide whether the target node of *Input Query* is granted to the user or not. When there is an edge matched to the *Input Query XPE* ((1) of **Algorithm *Query_Filter***), we check the access right in the node, which can be one of '*access-denied*', '*access-undefined*' or '*access-granted*'.

When a node, which has predicates, is '*access-denied*' ((2) of **Algorithm *Query_Filter***), we evaluate the ancestors whether their access rights are inherited to the node or not (**Algorithm *Ancestor_Check***, access right of node '10' is propagated to node '15' in Fig. 4). If an access-granted ancestor is inherited to the node, the *XPE* is granted to the node except the parts that belong to the predicates. For example, 'order/order_info[price $\le$ 30]addr' is granted to 'Alice' in Fig. 4. The access right of the predicate (e.g., 'order/order_info[price]') is verified in **Algorithm *Predicate_Comparison***. To exclude the predicate from the *XPE*, which is denied at the node, 'not' operation is inserted into the predicate ((7) of **Algorithm *Query_Filter***, e.g., 'order/order_info[**not**(price $>$ 30)]/addr'). If the node is '*access-denied*' without a predicate, the node is denied to access.

When the access right is '*access-undefined*', the access right of the node might be inherited from its ancestor nodes. Therefore, the ancestors are looked up from the parent to the root using the ''most specific overrides'' policy (**Algorithm *Ancestor_Check***). If the node is '*access-granted*' inherently, we regard it as an access-granted one. Therefore, when the access right is '*access-granted*' inherently or directly, we check whether the *XPE* is terminated with '//' or '*', which means the scope of *XPE* includes all descendants or all child nodes, respectively ((13) of **Algorithm *Query_Filter***). In these cases, we evaluate its descendants in **Algorithm *Descendant_Check*** and append the paths of the access-granted nodes to a set of access-granted *XPEs Grant_XPE* ((14) of **Algorithm *Query_Filter***). When the *XPE* is terminated with '//', we select the most inclusive nodes, where the nodes and all descendants are granted to access, and append '//' to the paths. If an access-granted node is a leaf one or its descendants are denied to access, only the node path is appended to *Grant_XPE*. And when *XPE* is terminated with '*' and all child nodes are granted to access, the *XPE* is granted. Otherwise, each child is verified and appended to *Grant_XPE* individually in **Algorithm *Descendant_Check***. If *XPE* has predicates, we evaluate the predicates in **Algorithm *Predicate_Comparison*** ((17) of **Algorithm *Query_Filter***). When a predicate contains '//' or '*', we assemble all paths of the predicate into a set of predicates. And then verify the access right of each predicate (or set of predicates). If one of the predicates (or set of predicates) is denied to access, the *XPE* is ignored in the node. However, a part of the set of predicates is denied, the part is removed from the set of predicates and the rest is granted to the user.

The path and predicates of the node are appended to *Grant_XPE* ((20) of **Algorithm *Query_Filter***). Lastly, *Grant_XPE* is appended to *Auth_XPE*, which is the result of '*Query Filter*'. After processing all *XACT* edges matched by the *XPE*, the '*Query Filter*' process ends.

Example 3. We show four user queries and their access-granted queries, where the *XACT* is based on Fig. 4.

Q1: (Bob, read, order/customer_info//) → (Bob, read, order/customer_info; order/customer_info/name; order/customer_info/phone; order/customer_info/addr//)

Q2: (Bob, read, //price) → (Bob, read, order/order_info/price)

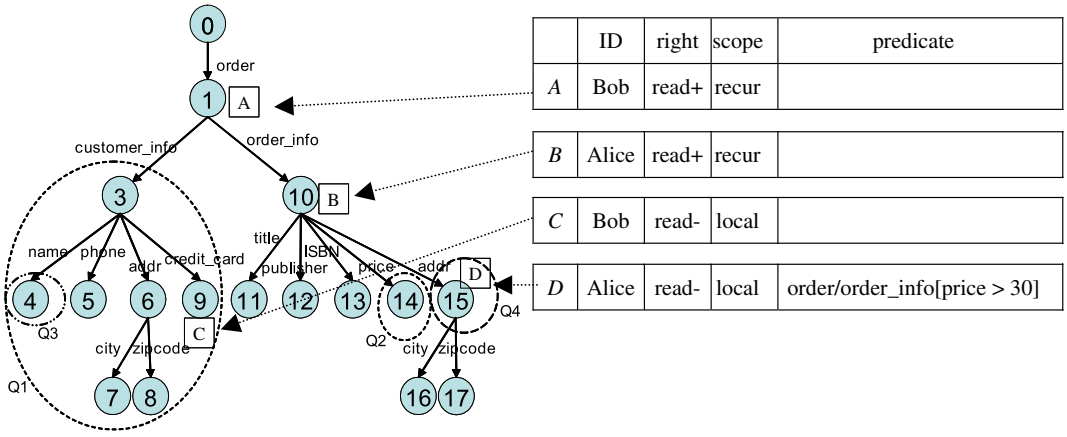


Figure 7 *XACT* and its target nodes denoted by the *Input Query* in Example 3.

Q3: (Alice, read, order/customer_info/name) → Ø
Q4: (Alice, read, //order_info[ISBN]/addr) → (Alice, read, order/order_info[ISBN and not(price > 30)]/addr)

The target nodes of Q1 are node '3' and its descendants in Fig. 7. All the target nodes except node '9' are *access-granted* to the user. So, we append them to *Authorized Query*. Q2 is corresponding to node '14' which is *access-granted* to the user, and Q3 is *access-denied* at node '4'. Q4 is *access-denied* at node '15' with a predicate to the user. Hence, the ancestors of the node are looked-up from the parent to the root according to ''most specific overrides'' policy. In Fig. 7, node '10' is *access-granted* recursively, and hence its accessibility is propagated to the node '15'. The node '15' is partially granted. Therefore, we generate an *access-granted XPE* for Q4 by adding a constraint for the predicate (i.e., [not(price > 30)]).

Because the *XACT* is a reduced XML tree, we do not know whether the objects for *Authorized Query* exist in actual XML documents. Hence, we regard that the results of the query always exist in the XML document.

The symbols for the following Lemmas and Theorem are described in Table 1 and 2.

Lemma 1. $(\Theta - \Delta) \equiv_{D_i} (\Theta' - \Delta')$ *w.r.t the XML document* D_i.

Proof. $\Theta \equiv_{D_i} \Theta'$ and $\Delta \equiv_{D_i} \Delta'$ by Theorem 1. Let $\mathbf{P} = (\Theta - \Delta)$ and $\mathbf{P}' = (\Theta' - \Delta')$, then $\mathbf{P}$ and $\mathbf{P}'$ are closed by Corollary 2. That is, $\mathbf{P} \equiv_{D_i} \mathbf{P}'$ (Theorem 1). Hence, $(\Theta - \Delta) \equiv_{D_i} (\Theta' - \Delta')$ w.r.t the XML document D_i. □

The difference between two *XPEs* is necessary when we subtract the *access-denied* part from the *access-granted XPEs*. Also, the intersection between two *XPEs* is required when we select the part denoted by both the *access-granted* and the query *XPEs*.

Lemma 2. $(\mathbf{P} \cap \Pi) \equiv_{D_i} (\mathbf{P}' \cap \Pi)$ *w.r.t the XML document* D_i.

Proof. $\mathbf{P} \equiv_{D_i} \mathbf{P}'$ by Theorem 1. $(\mathbf{P} \cap \Pi)$ and $(\mathbf{P}' \cap \Pi')$ are closed by Corollary 1. Hence, $(\mathbf{P} \cap \Pi) \equiv_{Di} (\mathbf{P}' \cap \Pi')$ w.r.t the XML document D_i. □

Theorem 2. $((\Theta - \Delta) \cap \Pi) \equiv_{D_i} \Pi'$ *w.r.t the XML document* D_i.

Proof. In this paper, we take it for granted that the process of DOM tree labeling method (Damiani et al., 2000) is correct. Therefore, we show that process of DOM tree labeling method $(\Theta - \Delta) \cap \Pi$ and that of our method Π' are equivalent w.r.t the XML document.
$\Pi' = (\Theta' - \Delta') \cap \Pi$ by the 'algorithm of *Query Filter*'. $\Theta \equiv_{D_i} \Theta'$ and $\Delta \equiv_{D_i} \Delta'$ by Theorem 1. Also,

Table 1 Notations for *XPE*

Symbol	Description	Symbol	Description
$\mathbf{P}$	a set of access-control *XPEs*	$\mathbf{P}'$	a set of *XPEs* transformed from $\mathbf{P}$ by the *XACT* Constructor
Θ	a set of access-granted *XPEs*	Θ'	a set of *XPEs* transformed from Θ by the *XACT* Constructor
Δ	a set of access-denied *XPEs*	Δ'	a set of *XPEs* transformed from Δ by the *XACT* Constructor
Π	a set of query *XPEs*	Π'	a set of *XPEs* filtered from Π by the Query Filter

Table 2 Notations and their values for time analysis

Symbol	Description	Assumed value
N_e	The number of element nodes in the XML document tree	
N_n	The number of nodes in the *XACT*	$N_n^2 < N_e$
N_p	The average number of paths in the *XACT* corresponding to a complex *XPE*	10
N_d	The average number of duplicate paths for a unique path in an XML document	$(3/2)\sqrt{N_e}$
R_g	The rate of access-granted query	80%
R_n	The rate of searched nodes for a complex *XPE* in XML (or *XACT*)	30%
N_l	The average number of access-control *XPEs* per user in ACL	10

$\Theta - \Delta \equiv_{D_i} \Theta' - \Delta'$ by Lemma 1, and $(\Theta - \Delta) \cap \Pi \equiv_{Di} (\Theta' - \Delta') \cap \Pi$ and by Lemma 2.
Hence $(\Theta - \Delta) \cap \Pi \equiv_{D_i} \Pi'$. □

Theorems 1 and 2 show that the transformation of *ACL* into *XACT* and access control of queries with the *XACT* are correct. Therefore, the proposed method is correct.

Performance analysis

The performance analysis is carried out for the proposed method, DOM tree labeling method (Damiani et al., 2000) and CAM (Yu et al., 2002). We consider the number of node accesses for each method.[1] To simplify the analysis, we consider two kinds of *XPEs*: simple *XPE* (without '//' and '*'), and complex *XPE* (with '//' or '*'). Some symbols for the analysis are described in Table 2.

The number N_e of element nodes in the XML document varies in the type and the size of the XML document. However, the number N_n of nodes in the *XACT* is not so variable as the type and the size of its corresponding XML document. In general, when the number N_e is more than 500, $N_n^2 < N_e$. The average number N_d of duplicate paths in an XML document increases with the number N_e, we assume $N_d = (3/2)\sqrt{N_e}$ based on "The Plays of Shakespeare" OASIS, 2003. We analyze the performance of above methods in two phases; access control and query processing. In the following formulae, we describe them in the form of 'access control cost' + 'query processing cost'.

(1) Simple *XPE*

(i) The proposed method: $\log_2 N_n + R_g N_d \log N_e$ (for access to *XACT* and XML document)
(ii) CAM: $\log_2 N_n + R_g N_d \log N_e$ (for access to access-control look-up data and XML document)
(iii) DOM tree labeling method: $N_l N_d \log N_e + N_d \log (N_e R_g)$ (for labeling and access to access-granted XML document)

In the simple *XPE*, the performance is different according to its access-control method. In the proposed method, we search *XACT* with a simple *XPE* for the filtering of access-granted *XPE*. We assume that the average breadth of *XACT* is 2 and that of an XML document is 10, since the duplicate paths are eliminated in *XACT*. When an *XPE* is granted to access, the method searches all nodes in the XML document which are denoted by the *XPE*. The cost for processing a simple *XPE* is '$\log_2 N_n + R_g N_d \log N_e$', which is the same as that of the CAM. In the DOM tree labeling method, when a query is submitted, it labels the DOM tree with access-control *XPEs*, denoted N_l, and prunes the access-denied nodes. Then the query is submitted to the access-granted tree. Therefore, the cost for processing a simple *XPE* is '$N_l N_d \log N_e + N_d \log (N_e R_g)$'.

The performances of the proposed method and DOM tree labeling method are as follows:

$\log_2 N_n + R_g N_d \log N_e : N_l N_d \log N_e + N_d \log (N_e R_g)$
$\Rightarrow 3 \log_{10} N_n + R_g N_d \log N_e : N_l N_d \log N_e + N_d \log N_e$ (since $\log_2 N_n \approx 3 \log_{10} N_n$, and $\log(N_e R_g) \approx \log N_e$)
$\Rightarrow (3/2) \log_{10} N_e + R_g N_d \log N_e : N_l N_d \log N_e + N_d \log N_e$ (since $N_n^2 < N_e$)
$\Rightarrow (3/2) + R_g N_d : (N_l + 1) N_d$ (divide by $\log N_e$ on both side)
$\Rightarrow R_g : N_l + 1$ (divide by N_d on both side)
$\Rightarrow 0.8 < 11$ (since R_g: 0.8 and N_l: 10)

Therefore the performance of the proposed method is better than that of DOM tree labeling method for simple *XPEs*.

(2) Complex *XPE*

(i) The proposed method: $R_n N_n + R_g N_p N_d \log N_e$ (for access to *XACT* and XML document)

[1] Since a variety of storage approaches are discussed in XML research areas, we consider only the number of node accesses in the memory space.

(ii) DOM tree labeling method: $N_l R_n N_e + R_n(N_e R_g)$ (for labeling and access to access-granted XML document)

In the complex *XPE*, both the performances of access control and query processing are different. In the proposed method, a complex *XPE* is transformed into N_p access-granted paths during the comparison of its access right in the *XACT*. And each of the path searches N_d duplicate paths in the XML document. Therefore, the cost is '$R_n N_n + R_g N_p N_d \log N_e$'. Note that, CAM is not applicable to complex *XPE* queries because only simple *XPEs* can refer to the look-up data. In the DOM tree labeling method, the number N_l of access-control *XPEs* labels the XML document and a query *XPE* searches the access-granted XML document. Therefore, the cost is '$N_l R_n N_e + R_n N_e R_g$'.

The performances of the proposed method and DOM tree labeling method are as follows;

$R_n N_n + R_g N_p N_d \log N_e : N_l R_n N_e + R_n(N_e R_g)$
$\Rightarrow R_n \sqrt{N_e} + R_g N_p (3/2)\sqrt{N_e} \log N_e : N_l R_n N_e + R_n N_e R_g$ (since ${N_n}^2 < N_e$ and $N_d = (3/2)\sqrt{N_e}$)
$\Rightarrow R_n + R_g N_p (3/2) \log N_e : \sqrt{N_e}(N_l + R_g) R_n$ (divided by $\sqrt{N_e}$ on both side) (since $(\log N_e) < \sqrt{N_e}$, and R_n, R_g, N_p and N_l are constants)
$\Rightarrow (0.3 + 0.8 \times 10 \times (3/2) \log N_e) : \sqrt{N_e}(10 + 0.8)0.3$
$\Rightarrow 0.3 + 12 \log N_e : 3.24\sqrt{N_e}$
$\Rightarrow 0.3 + 12 \log N_e < 3.24\sqrt{N_e}$ (when $64 < N_e$)

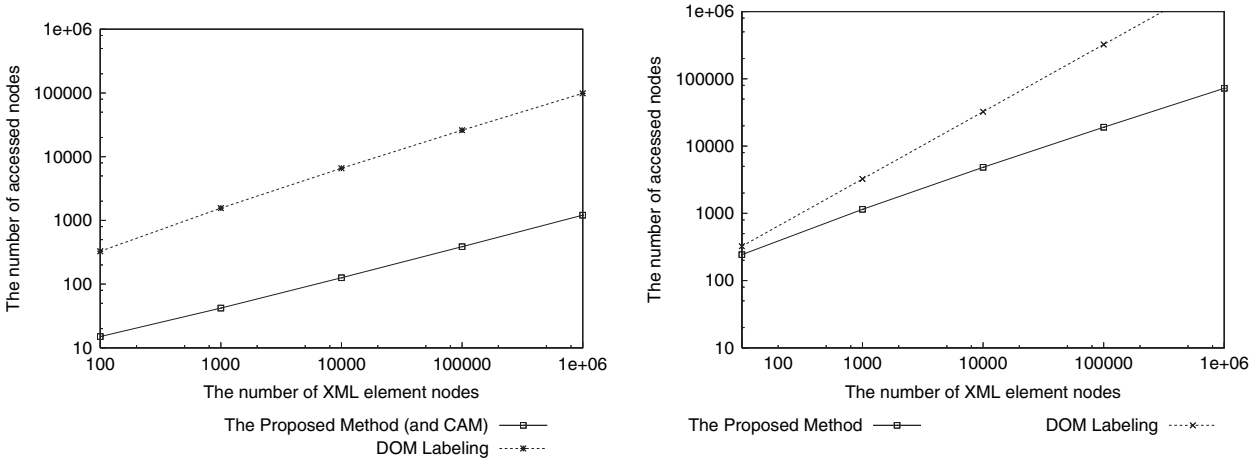
Therefore the performance of the proposed method is better than that of DOM tree labeling method for complex *XPE* when the number N_e of XML elements is greater than 64.

We simulated the performance of the proposed method and the DOM tree labeling for simple and complex *XPEs*, where the notations and their assumed values are described in Table 2. Since the formula of CAM is the same as that of the proposed method for simple *XPEs*, the performance results are identical. Fig. 8 shows the performance of the methods. The performance of the proposed method is superior to that of the DOM tree labeling method for both simple and complex *XPEs*, because it reduces the size of query XPath expressions by comparing the *XPE* with access-control *XPE*s before accessing XML documents.

Comparing with the existing methods (Bertino et al., 2001; Damiani et al., 2000; Yu et al., 2002), our proposed method dramatically reduces the number of node accesses for simple and complex *XPEs* compared with that of the DOM tree labeling methods (Bertino et al., 2001; Damiani et al., 2000). Second, even though the proposed method and CAM (Yu et al., 2002) have the same number of node accesses for simple *XPEs*, the proposed method can handle complex *XPEs* which are hardly applicable to CAM. Third, the proposed method supplements (Kudo and Hada, 2000), though it did not mention the matter of *XPEs* comparison, which requires the solutions of comparison between complex *XPEs*.

Conclusion

In this paper, we proposed an access-control method which is composed of 'XACT Constructor' and 'Query Filter' algorithms. The 'XACT Constructor' constructs XACT (XML Access Control Tree) which is a reduced XML tree with access-control



(a) simple *XPE* (b) complex *XPE*

Figure 8 The performance of the proposed method, CAM and DOM labeling.

information. And the 'Query Filter' filters access-granted XPath expressions from query XPath expressions. When a query is submitted, the method searches target nodes of the query XPath expression in the XACT and compares the access type and user information of the query with the access-control data in the XACT nodes. If the query XPath expression is granted in the XACT node, it is appended to the access-granted XPath expressions. We proved the correctness of our method and analyzed the performance of the algorithms with some previous work. The merits of our method are as follows: (1) it reduces the size of query XPath expressions by comparing the XPath expression with access-control XPath expressions before accessing XML documents; (2) it is suitable for complicated access-control system because the XACT supports access control and query XPath expressions with descendant axes, wildcards and conditions. The proposed method is also applicable to query processing of materialized views of XML documents.

Acknowledgement

This work was supported by the Korea Research Foundation Grant (KRF-2003-003-D00350).

References

Benedikt M, Fan W, Kuper G. Structural properties of XPath fragments. ICDT 2003, Italy; January 2003.

Bertino E, Castano S, Ferrari E. Securing XML documents with Author-X. IEEE Internet Computing 2001;5(3):21—31.

Blyth A, Cunliffe D, Sutherland I. Security analysis of XML, XML usage and XML parsing. Computer & Security 2003;22(6): 494—505.

Cho S, Amer-Yahia S, Lakshmanan L, Srivastava D. Optimizing the secure evaluation of twig queries. In: Proceedings of the 28th VLDB Conference. Hong Kong, China; 2002.

Cho S, Amer-Yahia S, Lakshmanan L, Srivastava D. LockX: a system for efficiently querying secure XML. In: Proceedings of the SIGMOD 2003 conference. San Diago, CA; 2003.

Damiani E, De Capitani di Vimercanti S, Paraboschi S, Samarati P. Securing XML documents. In: Proceedings of the 2000 international conference on extending database technology (EDBT2000). Germany; March 2000.

Damiani E, De Capitani di Vimercanti S, Paraboschi S, Samarati P. Securing SOAP e-services. International Journal of Information Security 2002;1:100—15.

Document Object Model, <http://www.w3.org/DOM/>.

Jajodia S, Samarati P, Sapino ML, Subrahmanian VS. Flexible support for multiple access control policies. ACM Transactions On Database Systems June 2001;26(2):214—60.

Kudo M, Hada S. XML document security based on provisional authorization. CCS 2000, Athens, Greece.

Miklau G, Suciu D. Containment and equivalence for an XPath fragment. ACM PODS. Wisconsin; June 2002.

Murata M, Tozawa A, Kudo M. XML access control using static analysis. CCS 2003. Washington, DC, USA; 2003.

OASIS. Cover pages, <http://xml.coverpages.org/xml.html>; October 2003.

OASIS. eXtensible access control markup language (XACML) version 1.0, OASIS standard, <http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml>; February 2003.

OASIS. Security assertion markup language (SAML) version 1.1, OASIS standard, <http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=security>.

W3C. Canonical XML version 1.0, W3C recommendation, <http://www.w3.org/TR/xml-c14n>; March 2001.

W3C. Extensible markup language (XML) 1.0. 2nd ed. <http://www.w3.org/TR/REC-xml>; October 2000.

W3C. SOAP security extensions: digital signature, W3C note, <http://www.w3.org/TR/SOAP-dsig/>; February 2001.

W3C. XML key management specification (XKMS), W3C note, <http://www.w3.org/TR/xkms/>; March 2001.

W3C. XML path language (XPath) version 2.0, W3C working draft, <http://www.w3.org/TR/2003/WD-XPath20-20030502/>; 2003.

W3C. XML-signature syntax and processing, W3C recommendation, <http://www.w3.org/TR/xmldsig-core/>; February 2002.

W3C. XQuery: a query language for XML. W3C working draft, <http://www.w3.org/TR/2003/WD-xquery-20030502/>; May 2003.

W3C. XSL transformations (XSLT) version 1.0, W3C recommendation, <http://www.w3.org/TR/xslt>; November 1999.

Yu T, Srivastava D, Lakshmann L, Jagadish H. Compressed accessibility map: efficient access control for XML. In: Proceedings of the 28th VLDB Conference, Hong Kong, China; 2002.

Jae-Myeong Jeon received his B.S. and M.S. degrees in Computer Science from Korea Air Force Academy, Cheongjoo, Korea, in 1988 and from Oklahoma State University, Stillwater, OK, in 1995, respectively. And he is currently a Ph.D. candidate in the Department of Computer Science at Korea Advanced Institute of Science and Technology (KAIST), Daejon, Korea. His research interests include security, XML and database systems.

Yon Dohn Chung received his B.S. degree in Computer Science from Korea University, Seoul, Korea, in 1994, and his M.S. and Ph.D. degrees in Computer Engineering from KAIST in 1996 and 2000, respectively. In 2003, he joined the faculty of the Department of Computer Engineering, Dongguk University, Seoul, Korea, where currently he is an assistant professor. Before joining the department, he worked in the Department of Computer Science of KAIST as a post-doctoral research associate and a research professor. His research interests include mobile databases, spatio-temporal databases, XML databases, data stream processing, and database systems.

Myoung Ho Kim received his B.S. and M.S. degrees in Computer Engineering from Seoul National University, Seoul, Korea, in 1982 and 1984, respectively, and his Ph.D. degree in Computer Science from Michigan State University, East Lansing, MI, in 1989. In 1989 he joined the faculty of the Department of Computer Science at KAIST, Taejon, Korea, where currently he is a professor. His research interests include database systems, OLAP, XML, mobile computing, transaction management, information retrieval, workflow and distributed processing. He is a member of the ACM and IEEE computer Society.

Yoon Joon Lee received his B.S. degree in Computer Science from Seoul National University, Seoul, Korea, in 1977 and his M.S. degree in Computer Science from KAIST 1979. And he received his Ph.D. degree in Computer Science from INPG-ENSIMAG, France, in 1983. In 1984 he joined the faculty of the Department of Computer Science at KAIST, Taejon, Korea, where currently he is a professor. His research interests include database systems, Internet, Web databases and distributed systems. He is a member of the ACM and IEEE computer Society.