A Scalable Access Control Model for XML Databases

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

Abstract XML databases involving sensitive information introduce a new challenge specific to scalable and efficient access control for data protection. However, related approaches have suffered from scalability problems because they tend to work on individual documents. In this paper, we propose a novel approach to XML access control through Java-based rule functions. A rule function is an executable code fragment that encapsulates the access control rules of a specific user or group, and is shared by all documents of the same document type. At runtime, the rule functions corresponding to the access request are loaded to the main memory and executed to determine the accessibility of document fragments. Moreover, this approach enables an efficient real-time update when the rules are updated. Using synthetic and real data, we show the scalability of the scheme by comparing the accessibility evaluation cost.

Keyword XML, XML database, Access control, Security

1. Introduction

The Extensible Markup Language [6] is widely used for data presentation, integration, and management because of its rich data structure. In applications such as business transactions and medical records, sensitive data may be scattered throughout an XML document and access control on node-level (element or attribute) is required to ensure that sensitive data can only be accessed by authorized users. Access control must be expressive and be able to support rules that select data based on the location and value(s) of data. In practical applications, such as electronic libraries, and credit card companies, the number of access control rules is on the order of millions, which is a product of the number of document types (in 1000's) and the number of user roles (in 100's). It is obvious that such applications call for high scalability and performance from the underlying access control system.

Ideas to efficiently perform expressive access control have been proposed in [2, 7, 10]. They are effective in searching for access controlled nodes [2, 10], or eliminating unnecessary accessibility checks at runtime [7]. These research efforts have managed to improve the efficiency of expressive access control; however, since they generally focus on document-level optimizations basis on the access control rules, rule updates may incur unacceptable costs. In our previous research [26], a rule-based matching tree is constructed to achieve high expressiveness which managed to handle 760,000 rules. And in another previous research work [27], the rule number is extended to 2,000,000 which obviously improves scalability. The two previous approaches also share the point that they both focus on uni-subject case in which each user is specified to one subject. However, in many real applications, a user may belong to multiple groups or domains that involve more complicated and costing decision making for the multi-subject environment. For example, when Alice is an employee and a manager at the same time, both rules for the employee group and the manager groups should be used to decide whether Alice can access the department information.

In this paper, we extend the access control model introduced in [27] and present an access control system with highlight on multi-subject decision making. The novelty of this access control model is the high scalability and the high performance. The key idea is to encode the access control rules of each subject as a rule function, and then further group the rule functions to Java classes on the basis of the relationship between subjects. Only the corresponded class is required to reside in the memory and the rule function(s) are executed for actual access evaluation. In addition, the related rule function is executed only once for accessibility decision of the subject. As far as we know, our Java-based approach is the first one that is capable of supporting millions of access control rules efficiently.

The rest of this paper is organized as follows. After reviewing some preliminaries in Section 2, we introduce our Java-based access control model in Section 3. Section 4 describes how to generate rule functions from the access control policy. In Section 5, the limitations and optimization of rule function grouping are introduced. Runtime access control enforcement is introduced in Section 6, and experimental results are reported in Section 7. In Section 8 we summarize our conclusions and future work.

1.1 Related Work

Many approaches for enforcing XML access control have been proposed. Some of them [14, 20] support full XPath [8] expressions to provide expressiveness by creating the projection of the policy on a DOM [16] tree. However, these approaches incur massive costs when handling a large policy or a deeply layered XML document. The mechanisms proposed in [1, 3, 9, 10] perform more efficiently but also encounter the same problem since the node-level access control on a DOM-based view can be expensive when processing large numbers of XML documents.

To overcome this problem, several efficient access control models have been proposed [22, 25]. Qi et al. [25], our previous research, presents a method performing in near-constant time regardless of the number of rules. This is achieved by using an access condition table which is generated from the access control rules independent of the XML data. However, this approach places limitations on the XPath expression, and does not provide an efficient runtime evaluation mechanism for value-based conditions. We also proposed a policy matching tree in [26] to achieve expressiveness and scalability. Though this approach can handle almost 760,000 rules, we developed another approach [27] supporting 2,000,000 rules by converting the policy to a group of rule functions in Java. Related Java classes are compiled before runtime, then loaded to the main memory as necessary and executed for the accessibility decision of a subject. However, the approach does not support access control enforcement when multiple subjects are involved for an accessibility decision.

Murata et al. [22] optimizes pre-processing steps by minimizing the number of runtime checks for determining the accessibility of nodes in a query with automata. However, the mechanism is limited to XQuery [5] and cannot handle other query languages and primitive APIs such as DOM.

A different approach with document-level optimizations is also proposed by Yu et al. [28]. Their scheme enforces efficient access control with an accessibility map which is generated by compressing neighboring accessibility rules to improve cost efficiency. However, since the maps are generated on a document-level, document updates or policy updates may trigger expensive re-computations especially for a large XML database.

Optimizations are also done in a number of research efforts on XML query languages (e.g., XPath and XQuery). The methods include query optimization based on (i) the tree pattern of queries [7, 11, 24], (ii) XML data and XML schema [13, 18, 19, 21]; and (iii) the consistency between integrity constraints and schemas [12]. However, these perform efficient data selection usually on document-level and require indices. Therefore, in a large XML database, for instance, a database with 10,000 document collections and 10,000 documents for each document collection, such optimization mechanisms may consume a prohibitive amount of space. Moreover, these technologies are designed for XQuery and they cannot handle other query languages and primitive APIs such as DOM.

2. Access Control Policy

Various access control policy models have been proposed. We use the one proposed by Murata et al. [22] in which an access control policy contains a set of 3-tuple rules with the syntax[1]: <Subject, Permission Action, Object>.

The subject has a prefix indicating the type such as *uid*, *role*, and *group*. Permission '+' stands for a grant rule while '-' means denial. The action value can be either read (r or R), update, create, or delete. Due to lack of space, we focus on read in this paper, as the others can be implemented with the same mechanism. The rule with +R or –R propagates read permissions (grant or denial) downward to the entire sub-tree, while +r grants a permission on the selected node only. As an example, (uid:Alice, +r, /a) specifies that user Alice is allowed to access /a but access to /a/b is implicitly specified since grant is not propagated down to the descendants of /a. Moreover, according to the *denial downward consistency* principle [22] which stipulates that the descendants of an inaccessible node are also inaccessible, accessibility dependence exists between ancestors and descendants. Therefore, it is obvious that –r and –R are equivalent, and thus we specify denial rules only with –R in this paper. In addition, in order to maximize data security, we (i) resolve access conflicts with the *denial-takes-precedence* principle [22], and (ii) apply the default action of denying permission on paths that have no explicit and implicit access control specified.

[1] The policy syntax can be represented in XACML [23]. In this paper, we use the above syntax for simplicity.

3. Scalable Access Control Model

In this section, we describe a scalable access control model that consists of a rule function, a mapping table, evaluation results, and an evaluation algorithm. As an extension of the previous work, we add a process in which multiple returned values of the rule functions are combined to the final accessibility result. The process could be defined regarding to the application requirements.

3.1 Rule Functions

A rule function is the basic fragment in Java that performs access control on a specific subject. The rule function receives the accessed path and returns an evaluation result to the caller.

If a user owns multiple subjects, when Alice participating in multiple user groups for example, multiple rule functions are bound for accessibility evaluation.

In our previous work [27], two types of rule functions are introduced that one is indexed by the object, and the other one is indexed by the subject. However, the former model is not able to support the multi-subject environment since the accessibility decision mechanism may be different regarding to the application requirements when multiple subjects involve.

3.2 Evaluation of Accessibility

Given a requested path of a request, the rule function returns an evaluation result in accordance with both the action permission and the propagation property of the appropriate rule. There are four types of evaluation results as Table 1 shows.

Table 1 Evaluation results of a rule function

Access effect	Evaluation Result
+r	GRANT_ON_NODE
+R	GRANT_ON_SUBTREE
-R	DENY
Nothing	UNDECIDED

As Table 1 shows, positive authorization is defined by GRANT_ON_NODE, and by GRANT_ON_SUBTREE which propagates access permissions to the sub-tree. When access is denied, the evaluation result of the descendants is also DENY due to the denial downward consistency principle. UNDECIDED is returned when the accessed path is not covered by any rule. Access is denied by default if no further access control is specified by other rules.

3.3 Mapping Tables

A mapping table is the key component that connects an access request, which contains one or multiple subjects, to the appropriate rule functions for accessibility evaluation. We define a mapping table which holds subject as key and the 3-tuple of package name, class name, and function name as its value.

In addition, when handle a large-scaled policy, we can construct a hierarchical mapping table to achieve scalability that each Java package contains a single mapping table.

3.4 Evaluation Algorithms

Based on the defined components (rule functions, evaluation results, and mapping tables), our model computes a decision result of an access request based on the following algorithms. The requirement of our approach is: The access control policy must satisfy the consistency that all of the ancestors of a descendant specified with grant accessibility should be accessible as well.

Given the path and the subjects of the access request, the corresponding entry of each subject is looked up in the mapping table. If the corresponding entry exists, the rule function with the name found in the table is invoked and executed. Since rule functions for multiple subjects may return different Boolean results, *true* or *false*, for the same request. The mechanism to combine all of the returned values for a final accessibility result may be different regarding to the application requirements. Therefore, we separate the combination procedure away from the evaluation mechanism.

3.5 Result Combination Process

When a user is bound to multiple subjects, since multiple run functions may result in different Boolean results, *true* or *false*, to combine the results with *denial-take-precedence* principle or *grant-take-precedence* principle should be decided by the application requirements.

The relationship between the concerned subjects can be joined. For example, Alice is both the employee and the manager. Therefore, Alice's access request should be evaluated with the rules specified for the employee group and the manager group. Since manager usually has the higher privilege than employee, if the access to a path is denied by the employee group but granted by the manager group, as a final accessibility result the access is granted. In this case, *grant-take-precedence* principle is used to combine the results.

However, *denial* may override *grant* in some applications to guarantee the maximize security. It is obvious that different with other components of this access control model, the result combination process is application dependant.

4. Rule Function Generation

A policy is converted to a group of rule functions. During the conversion, both simple paths and paths involving // or predicates are coded as rule functions as well. Since various rules are specified for the same subject, the execution order of the rules in a rule function directly decides the accessibility result for the subject with the consideration of the *denial-take-precedence* and the propagation mechanism. As a consequence, the rules are converted into the code fragments in the order of *–R*, *+R*, and then *+r*. The details of the conversion order and the conversion algorithm are introduced in [27]. In this section, we briefly present some rule function examples.

4.1 Policy with Simple Path

In the rule function model, the access control policy is first sorted by subjects. For each subject and subset of rules corresponding to that subject, a rule function is produced containing code that implements those rules. Inside a rule function, the rules are distinguished by the object, and the evaluation result is coded as the return value when the accessed path satisfies the object condition. As an example, we have a rule subset of the access control policy P1 contains

(1) *Rule(Alice, +r, /a)* (2) *Rule(Bob, +R, /a)*.

Figure 1 shows the rule function for (1) and (2).

```
// the rule function for Alice
1  static public Integer sf_Alice(String path) {
2      if (path.equals("/a"))                    // for (1)
3          return GRANT_ON_NODE;
4      else return UNDECIDED;
5  }
// the rule function for Bob
6  static public Integer rf_Bob(String path) {
7      if (path.startsWith("/a"))                // for (2)
8          return GRANT_ON_SUBTREE;
9      else return UNDECIDED;
10 }
```

Figure 1 Example for P1

4.2 Policy with //

A path expression containing // selects nodes whose position in the data structure is not known precisely and therefore the path may map to multiple specific paths. Matching in a rule function is performed by regular expressions and the java.util.regex package. For instance, the third rule in P1 is (3) *Rule(Bob, -R, /a//d)*. The rule function for Bob is generated as Figure 2 shows. Note that -R rule is converted first, then coming up with +R.

```
1  static Pattern p1 = Pattern.compile("/a/.*/d|/a/.*/d/.*");

2  static public Integer rf_Bob(String path) {
3      Matcher m1 = p1.matcher(path);
4      if (m1.matches())                         // for (3)
5          return DENY;
6      else if (path.startsWith("/a"))           // for (2)
7          return GRANT_ON_SUBTREE;
8      return UNDECIDED;
9  }
```

Figure 2 Example handing //

4.3 Policy with Predicates

An object containing a predicate(s) is first pre-processed by separating the predicate(s) *pred* and the path *p*, then both *pred* and *p* are programmed in the rule function for the subject.

A predicate is a condition comparing XML data or a conjunction of such conditions. The comparisons are performed through the mathematical operators, =, <>, <, ≤, >, and ≥. These mathematical operators are translated to Java code to enforce predicate evaluation. Since predicate evaluation requires XML data stored in the XML database, the data referenced in the predicate needs to be retrieved before the evaluation. Therefore, an API, *retrieveData*, is provided to retrieve the required data. *retrieveData* has two parameters: the requested node name *n*, and the path expression *p* of the node

imposed by the predicate. Since at runtime the DBMS has the position information *pos* of the requested node, the position of *n* can be found by traversing from *pos* to *p*. For example, we have a *Record* document type which contains order details. Each order is identified with a unique customer ID, the *CustKey* element. To guarantee that a customer can only see their own order details, the access control policy P2 can be

Rule(role:customer, +r, /Orders)
Rule(role:customer, +R, /Orders/Order[CustKey=$custID])

In P2, *$custID* is a system variable which is automatically set when a customer logs in. The value of *$custID* is obtained by *obtainSystemData* which is another API returning the value of the system global variables. Assuming that the object *DataCenter* implements both APIs, the Java program fragment for handling the predicate evaluation is as shown in Figure 3.

If the rule contains multiple predicates, the Java code for the conjunction of the predicates is encoded.

```
String value0 = DataCenter.retrieveData("CustKey", "/Orders/Order");
String value1 = DataCenter.obtainSystemData("custID");
if (value0.equals(value1))
    return GRANT_ON_SUBTREE;
else
    return UNDECIDED;
```

Figure 3 Code fragment for predicate *CustKey=$custID*

5. From Rule Functions to Java Classes

5.1 Size Limitations

Owing to Java-based implementation, we group rule functions to Java classes to gain easy memory management provided by the Java Virtual Machine. The simplest way is to construct a Java class holding all of the rule functions. However, since in the Java Virtual Machine, the total sizes of heap and stack cannot exceed 65,536, which means the maximum number of rule functions, or subjects, is 65,536, and either the method size or the class size cannot exceed 65,536 bytes at meanwhile.

We checked 22 types of XML-based applications to see the number of paths in XML instances, and then we found the number is no more than 300 and most of them are below 100. If we specify a rule on each path for each subject, and each rule usually costs less than 100 bytes, then we can say that each rule function is less than 10,000 bytes. It is clear that in most cases we can group the rules sharing the same subject into a single rule function without violating the size limitation.

One more important factor in rule function grouping that affects the runtime performance is the file number in a Java package. For example, in Windows operation system, time for looking up a specific file turns costing when the file number in a directory exceeds 1024. As a result, we group at most 1024 Java classes in each Java package.

However, the number of the subjects may reach to several millions in some cases. For instance, a credit card company may prepare access control rules for each client. Therefore, rule functions, whose number may reach about millions, should be grouped into multiple Java classes. Besides that, the number of rule functions in each Java class should be decided also based on the rule number for each subject. From the experimental results shown in experimental results, we found the memory cost is independent of the rule number for each Java class, but time for loading from disk to main memory is directly influenced by Java class size.

5.2 Group Optimization

The multi-subject environment requires multiple rule functions to be executed for the accessibility result of an access request. To make the number of the classes on memory as few as possible, rule functions may be executed for an access request should be grouped into the same Java class as much as possible.

Therefore, except the access control policy, we introduce the subject specification to the system. In the subject specification, the subjects that possibly bound to the same user are list up. For instance, {{*Sub1, Sub2, Sub5*} {*Sub1, Sub3, Sub4*}} shows two subject groups that a user may belong to *Sub1*, *Sub2*, and *Sub5*, or *Sub1*, *Sub3*, and *Sub4*. When group rule functions to Java classes, the rules functions for *Sub1*, *Sub2*, and *Sub5* are constructed to a Java class while *Sub1*, *Sub3*, and *Sub4* for another. It is clear that the rule function for *Sub1* appears in two Java classes.

6. Function-based Access Control System

6.1 Access control system

The proposed function-based access control system is constructed through Access Control Modeling and Model Deployment as Figure 4 shows.

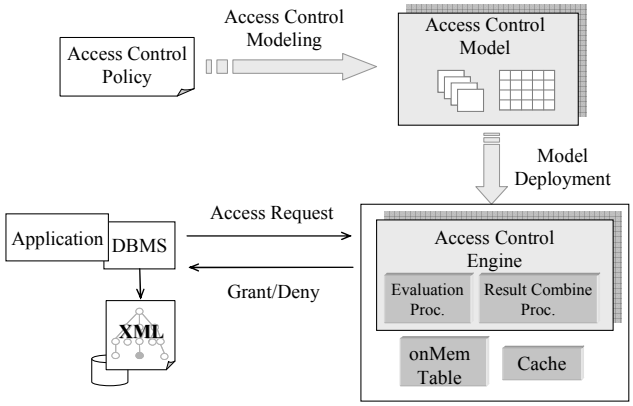

Figure 4 Function-based access control system

In Access Control Modeling, the access control policy is converted to Java classes where each class represents the access control rules for a certain number of subjects or objects. During this modeling, the corresponding mapping table is generated as well. In addition, the Java code fragments are compiled into bytecode so that they can be executed.

Another process, Model Deployment, initially loads the Java mapping table into main memory, and then prepares another empty system global table, *onMemTable*, that stores in main memory Java object instances that are required to process an access request. Since each function group has a unique name, *onMemTable* uses the group name as a key and associates it with the corresponding object instance.

At runtime, given an access request containing subjects and a simple path expression, the access control system runs the evaluation to get the decision result. The rule function for each subject is executed individually by Evaluation Process, and

then the results are combined by Result Combine Process for a final accessibility result. If the accessibility result returns DENY, the access denied response will be sent back to the user without data retrieval from the XML database. Otherwise, the output is generated after retrieving the data value from the database and returned to the user. Caching can be employed in both the access control system and the DBMS to reduce cost of the accessibility evaluations.

This system separates the access control system from the database engine so that security-related support is not required from the underlying database. This enables any XML DBMS, even off-the-shelf products, to provide scalable access control.

6.2 Policy update

The Java-based implementation loads the required classes into memory and so any updates to the classes residing on disk will not automatically change those in memory. But the class reload mechanism provided by Java can be utilized when we wish to update memory-resident classes. Class reloading provides a simple and low-cost solution to supporting policy updates.

When the policy is updated at runtime, generally we update the system in four steps: 1) generate a new Java class or multiple new Java classes for updated rules, 2) compile the generated Java classes, 3) update the mapping table, and 4) remove the affected entries in the accessibility cache. In step 3, if the corresponding entry exists in the mapping table, the system updates the entry with a new package name, a new class name, and a new method name. In addition, if the rules for new subjects are added to the policy, new entries are inserted into the mapping table. In case the class file affected by the policy update resides in memory, we remove the corresponding entry from *onMemTable* which enables the system to reload the class instance.

Easy updating is also a significant improvement from our previous approaches [25, 26] in which policy update leads to a re-computation of the entire access control table or the entire policy matching tree. If rules are updated while rule functions are being executed, one can either re-execute the rule functions after the update or wait until the current evaluation finishes before updating the rule functions. The appropriate strategy depends on the system configuration and requires further investigation.

7. Experiments

In this section, we describe our experiments to evaluate the performance of our function-based access control mechanism for XML documents. All of the experiments were conducted on a machine with 1.8GHz Pentium 4 CPU, 1.5GB of main memory, and IBM JDK1.4.2.

To demonstrate the scalability of the system, we examine the memory cost when a large access control policy is loaded into main memory, and the access control processing time when a large XML document is processed. To show the expressiveness of the access control specification, we also run experiments involving predicates and //. In addition, we show the performance gains achieved with the accessibility cache.

7.1 Results of the Experiments

Scalability to large access control policies

The main purpose of this experiment is to see whether the function-based model can support large access control policies. For simplicity, we specified 2,000,000 access control rules for 80,000 users for the *Orders* document type. Each user is associated with a set of 25 access control rules specified with simple path expressions and +r.

We varied the group size from 50 to 100; hence, 2,000,000 rules were translated into 800 to 1,600 Java classes. In the test, we managed to load all of the rules into the main memory in a random order without any Java garbage collection (GC) triggered. Memory cost was independent of group size and is close 58MB.

Scalability for large-scaled XML document

In many systems, XML-formatted documents for record retention may be several hundred megabytes in size. In this experiment, we show the performance of the system by examining the total processing time when the XML documents shown in Table 2 are accessed.

Table 2 XML document information for experiments

		Size	Rulest	D-rate(%)
D1	Orders.xml	4MB	25	99.8
D2	standard.xml	111MB	514	99.97

For each subject, we specified 25 access control rules for *Orders.xml*, and 514 rules for *standard.xml*. All rules specified a +r permission. Both documents contain repeated sub-structures and so part of the access control is duplicated at multiple locations. In Table 2, the fraction of duplicated paths is shown as the *D-rate* and we can see that *standard.xml* has more duplicated sub-trees than *Orders.xml*.

We used the SAX API of the XML parser to parse the entire document, and checked the accessibility when encountering either an element or an attribute. The processing time includes XML parsing time, Java class loading time, access control time, and GC time if any GC occurs. In this experiment, we label the total time excluding parsing time as *AC Time*. We also measured the performance improvement achieved with caching. In Figure 5, the processing times of the entire documents are shown.

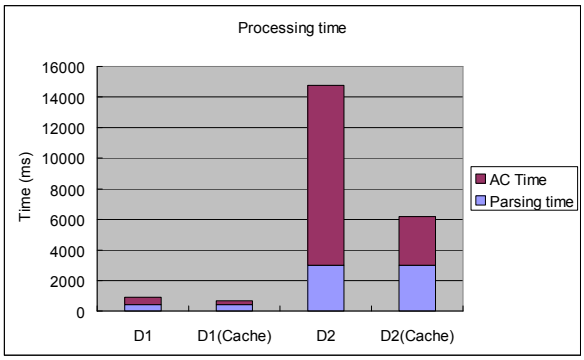

Figure 5 Processing time for D1 and D2

From the figures, it is clear that the accessibility cache makes a significant improvement in processing time. In the case of *Orders.xml*, the accessibility cache reduces the AC time by almost 50%. In the case of *standard.xml*, the AC time is reduced by 60-70%. Since the duplication rate of *standard.xml* is higher than that of *Orders.xml*, the accessibility cache is more effective when processing *standard.xml*.

Access control performance

The access control cost on a simple path is less than 3 microseconds per path. In the case of R, it cost around 2.2 microseconds per path, which is 20% less than for r. In our implementation, the performance for // depends on the performance of the java.util.regex package. The experimental results show the accessibility check involving // requires 5 to 5.5 microseconds per path compared to the 2.2 microseconds of R. Though // can be supported with reasonable performance, we recommend using R instead of // where applicable.

In our previous work [25], the access condition table driven access control model performs a little faster in that it takes approximately 2.0 microseconds per path. However, since the access control table is generated for the whole policy set, the memory consumption is massive when the policy is huge comparing to this rule function based approach. Moreover, in [25], the access control enforcement on predicates and // are not provided by the system that the user has to implement the enforcement by themselves.

Moreover, this approach performs better than the approach with a policy matching tree [26].

8. Conclusion and Future Works

In this paper, we have proposed a scalable access control model for providing expressive and efficient access control for XML databases. High scalability is achieved by grouping rule functions into Java classes and further organizing classes into packages. Each class is the unit for memory management and policy update. To improve performance, we enhance the access control system with a cache mechanism which eliminates the need for function invocation when the same path is accessed repeatedly by the same user. As an extended work, the system supports multi-subject environment.

In future work, we plan to explore deeper on the efficiency to see specific conditions for the model. We also plan to explore the generation of efficient rule functions by sharing more Java code inside the rule functions, which leads to less memory usage and more efficient class loading. We also plan to extend the predicate evaluation mechanism so that fewer database queries are made to retrieve the data values required for predicate evaluation.

9. References

[1] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti: Controlled access and dissemination of XML documents. ACM WIDM (1999) pp.22-27.

[2] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti: Specifying and Enforcing Access Control Policies for XML document Sources. World Wide Web Journal (2000), Vol. 3, No. 3, pp. 139-151.

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

[4] M. Bishop, L. Snyder. The transfer of information and authority in a protection system. Proc. 17th ACM Symposium on Operating Systems Principles, 1979.

[5] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon: XQuery 1.0: An XML query language, W3C Working Draft 12 November 2003. http://www.w3.org/TR/xquery/.

[6] T. Bray, J. Paoli, and C. M. Sperberg-McQueen: Extensible Markup Language (XML) 1.0. W3C Recommendation. http://www.w3g.org/TR/REC-xml (Feb. 1998).

[7] S. Cho, S. Amer-Yahia, L. V. S. Lakshmanan, and D. Srivastava: Optimizing the secure evaluation of twig queries. VLDB (2000) pp.490-501.

[8] J. Clark and S. DeRose: XML Path Language (XPath) version 1.0. W3C Recommendation. Available at http://www.w3g.org/TR/xpath, 1999.

[9] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati: Design and Implementation of an Access Control Processor for XML documents. WWW9 (2000).

[10] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati: A Fine-Grained Access Control System for XML Documents. ACM TISSEC (2002) pp.169-202.

[11] A. Deutsch and V. Tannen: Containment of regular path expressions under integrity constraints. KRDB (2001).

[12] W. Fan and L. Libkin: On XML integrity constraints in the presence of DTDs. Symposium on Principles of Database Systems (2001) pp.114-125.

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

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

[15] L. Gong: A Secure Identity-Based Capability System. Proc. IEEE Symposium on Security and Privacy, pp.56-65, 1989.

[16] A. L. Hors, P. L. Hegaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne: Document Object Model (DOM) Level 3 Core Specification. http://www.w3.org/TR/2004/PR-DOM-Level-3-Core-20040205 (2004)

[17] A. K. Jones, R. J. Lipton, and L. Snyder. A Linear Time Algorithm for Deciding Security. Proc. 17th Symposium on Foundations of Computer Science, Houston, Texas, pp. 33-41, 1976.

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

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

[20] M. Kudo and S. Hada: XML Document Security based on Provisional Authorization. ACM CCS (2000) pp.87-96.

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

[22] M. Murata, A. Tozawa, M. Kudo and H. Satoshi: XML Access Control Using Static Analysis. ACM CCS, 2003.

[23] OASIS. OASIS Extensible Access Control Markup Language (XACML), Feb. 2003. http://www.oasis-open.org/committees/xacml/docs.

[24] F. Neven and T. Schwentick: XPath containment in the presence of disjunction, DTDs, and variables. ICDT (2003) pp.315-329.

[25] N. Qi and M. Kudo: Access-condition-table-driven access control for XML databases. ESORICS (2004).

[26] N. Qi and M. Kudo: XML Access Control with Policy Matching Tree. ESORICS (2005).

[27] N. Qi, M. Kudo, J. Myllymaki, and H. Pirahesh. A

Function-based Access Control Model for XML Databases, ACM CIKM (2005).
[28] T. Yu, D. Srivastava, L. V. S. Lakshmanan, and H. V. Jagadish: Compressed Accessibility Map: Efficient Access Control for XML. VLDB (2002) pp.478-489.