The Pennsylvania State University
The Graduate School
College of Information Sciences and Technology

XML ACCESS CONTROL IN NATIVE AND RDBMS-SUPPORTED XML DATABASES

A Dissertation in
Information Sciences and Technology
by
Bo Luo

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2008

The dissertation of Bo Luo was reviewed and approved* by the following:

Dongwon Lee
Assistant Professor of Information Sciences and Technology
Dissertation Advisor, Chair of Committee

C. Lee Giles
David Reese Professor of Information Sciences and Technology

Peng Liu
Associate Professor of Information Sciences and Technology

Wang-Chien Lee
Associate Professor of Computer Science and Engineering

John Yen
University Professor of Information Sciences and Technology
Associate Dean for Research and Graduate Programs

*Signatures are on file in the Graduate School.

Abstract

As the eXtensible Markup Language (XML) has emerged as the *de facto* standard for storing and exchanging information in the Internet Age, the needs for efficient yet secure access of XML data naturally arise. It becomes increasingly important to be able to tailor information in XML data for various users and applications, while preserving confidentiality. In this dissertation, we ask how fine-grained XML access control can be supported when underlying (XML or relational) DBMS does not provide any security features for XML data.

We first present *deep set operators* for XML as an extension of conventional set operators, and use them to algebraically describe XML access control. We introduce a general framework to capture design principles and operations of existing XML access control mechanisms across centralized and distributed environments.

In the native XML environment (XDB), we advocate an efficient, view-free, Non-deterministic Finite Automata (NFA) based access control enforcement mechanism, called QFilter. It supports fine-grained XML access control and works independently from the underlying XML engine, thus provides great flexibility. In RDBMS-supported XML database systems (XRDB), we first introduce object and operation equivalency as a bridge between relational and XML data models. Then we present theoretical results on how one can (or cannot) support fine-grained XML access control using relational access control features. We also show implementation choices and the required security features from underlying RDBMS. Finally, we implement our approach and exhibit its superior performance against native XML DBMS.

Table of Contents

List of Figures

List of Tables

List of Symbols

Q/Q_X	User's input query in XPath expression
Q'	Re-written query from Q
Q_R	Relational query (SQL) converted from Q
R/R_X	An individual XML access control rule
R_R	An individual relational access control rule converted from R_X
R^+/R^-	A positive/negative access control rule
ACR	$[R_1, ..., R_n]$, a set of access control rules
ACR_X/ACR_R	Access control rule set in XML or relational model
D/D_X	XML document
$Q\langle D\rangle$ or $\langle Q\rangle$	The answer of the query Q against data D (D is omitted if clear).
$\overset{D}{\cup}, \overset{D}{\cup}_X, \overset{D}{\cup}_R$	Deep union operator, XML deep union operator, relational deep union operator
$\overset{D}{\cap}, \overset{D}{\cap}_X, \overset{D}{\cap}_R$	Deep intersect operator, XML deep intersect operator, relational deep intersect operator
$\overset{D}{-}, \overset{D}{-}_X, \overset{D}{-}_R$	Deep except operator, XML deep except operator, relational deep except operator

Acknowledgments

It has been a long journey towards the Ph.D. degree. When I look back all these years, there are a lot of people who I am thankful for.

First of all, I would like to take this chance to express my heartily thanks to my Ph.D. committee chair, Dr. Dongwon Lee, for his patient and professional advising in the past five years. He has not only provided me with valuable ideas, insights, and comments, but also has taught me the way of thinking and doing researching. He has not only given me advices in research, but also coursework, graduate student life, and even job hunting. It is a great pleasure and fortune to have him as my advisor.

Second, I would like to express my thanks to Dr. Peng Liu. He has been involved in all my research projects and publications. He is like a co-advisor to me, who also helped me in all the aspects of professional and personal life. I would also like to thank Dr. Wang-Chine Lee, who was in the group of QFilter research, which builds the foundation of almost all my publications. The meetings at Pond Lab opened the door of database security research for me. In additions, I took his course *CSE542 Database Systems*, which was a great benefit. I would also like to thank Dr. Lee Giles. I have learned a lot in his *IST511* class; and it's really a great honor to have him on my Ph.D. committee.

Also, I want to express my thanks to all the members of the Pike group: Dr. Byung-Won On, Dr. Seog-Chan Oh, Max Chang, Ergin Elmacioglu, Yoojin Hong, Hyunyoung Kil, Hung-sik Kim, Haibin Liu, Wonhong Nam, and Su Yan. We together create a friendly and competitive research atmosphere, from which we all benefit a lot!

There are a lot more people that I am truly thankful for: all my teachers ever since primary school, especially my Bachelor's advisor Dr. Nenghai Yu at University of Science and Technology of China, and Master's advisor Dr. Xiaoou Tang at the Chinese University of Hong Kong; my managers and mentors at IBM Silicon Valley Lab: Steve Chen, Dr. Guogen Zhang, Dr. Mengchu Cai, Sophia Perl, Dr. Rick Chang, and Dr. Dongmei Ren; many of my classmates, and colleagues

in various projects. I really cannot list everybody's name here. But I want to say "thank you" to all!

Last but not least, my parents have given me infinite love ever since I was born. Without their care and support, I would never be who I am. I also have to thank my wife Fengjun, for all the love and understanding in the past 12 years.

Dedication

To My Family.

Chapter 1

Introduction

The eXtensible Markup Language (XML) [12] has emerged as the *de facto* standard for storing and exchanging information in the Internet Age. As the distribution and sharing of information over the Web becomes increasingly important, the needs for efficient yet secure access of XML data naturally arise. It is necessary to tailor XML documents for various user and application requirements, while ensuring confidentiality and efficiency at the same time.

Current access control research can be categorized into two groups: access control modeling and access control enforcement mechanisms. Table 1.1 illustrates the current development of access control model research. First row refers to (research-oriented) access control models developed for XML and relational data models, respectively, while second row refers to state-of-art open-source or commercial products for each model. In general, not all the features proposed by modeling research community (first row) are implemented in existing access control enforcement approaches. For instance, to our best knowledge, most industrial or open source XML database products do not have any support for fine-grained XML access control yet.

	XML	**Relational**
Models	XML Access Control Models (e.g., [16], [4])	Relational Access Control Models (e.g., [33])
Products	XML Databases (e.g., Galax [67], Tamino [65])	Relational Databases (e.g., Oracle, DB2, SQL Server)

Table 1.1. The overview of XML and Relational access control model supports.

For native XML database systems, most available commercial or open-source products support either no access control, or document level access control. Controlling access at the document level (e.g., using file systems) is not suitable for today's XML applications, where data access is typically performed at element and attribute levels. To remedy these shortcomings, recently, various proposals in support of fine-grained XML access control have appeared. Most of them can be categorized as either the *view-based* (e.g., [4, 16, 19]) or *DBMS-based* (e.g., [15, 52][1]) approach. The view-based approach provides fast access to the authorized data (especially when views are materialized), but needs to deal with issues of view maintenance. On the other hand, the DBMS-based approach is efficient to maintain, but requires security support from underlying (XML or relational) databases. However, to our best knowledge, there are no XML databases that provide security-related features yet[2]. Therefore, our first research problem is looking for a better way to support XML access control. More specifically:

> **How can we implement a fine-grained XML access control enforcement mechanism (i.e. to support the upper-left quadrant of Table 1.1), which is non-view, and independent from the engine?**

In the first part of this dissertation, we analyze and examine three different classes of solutions for access control, namely, *primitive*, *pre-processing* and *post-processing*. In particular, we advocate a practical and scalable pre-processing solution, called QFilter, as an external component to the database engine. QFilter checks incoming XML queries against access control rules, and rewrites them such that parts violating access control policies are pre-pruned. Since QFilter does not use views, it entirely avoids the issues of high storage and maintenance costs. Furthermore, since QFilter does not rely on security-related features of underlying databases (e.g., GRANT/REVOKE in RDBMS), it can work with any off-the-shelf databases (as long as they can process XML queries). This property makes QFilter a very "practical" solution.

[1] [52] is a preprocessing approach, however, it requires assistance from underlying DBMS under the presence of predicates in the query.

[2] None of the recent development in [4, 16, 19, 15, 52] are adopted to commercial XML database products yet.

In the scenario of RDBMS-backed XML database systems (hereafter *XRDB*), XML data is shredded into relations and stored in RDBMS; query-answering is conducted through a conversion layer so that users interact with the system as if it is native XML. In the scenario of XML publishing, relational data is compiled into XML format for distribution and exchange; users receive documents as if they were originated from XML model. For both scenarios, we enjoy the benefit of XML model while taking advantage of the maturity of the off-the-shelf RDBMS. In both scenarios, it is desirable to natively specify access controls on the XML side (upper-left quadrant of Table 1.1), but they need to be enforced on the RDBMS side (lower-right quadrant). We believe that current XML access control enforcement mechanism research is in a sense re-inventing wheels without utilizing existing relational access control models (i.e., upper-right quadrant) or leveraging on security features that are readily available in relational products (i.e., lower-right quadrant). Therefore, our second research question is:

> **When is it (not) possible to support the upper-left quadrant of Table 1.1 (i.e., fine-grained XML access control) using the lower-right quadrant (i.e., RDBMS)? Why? How?**

The major challenges of supporting XML access controls in XRDB systems stem from the inherent discrepancy of XML and relational data models. Relational data model features a structure of two-dimensional table, while XML features a hierarchical data model. When XML data are shredded into relational data model by some transformation algorithms, not all transformation algorithms can fully preserve structural properties of XML model [2]. Therefore, the inherent incompatibility of two data models leads to the fundamental discrepancy between two access control models. Second, relational access control policies define authorized actions of "cells," where each cell is an impartible element and whose accessibility is explicitly expressed. However, XML nodes are hierarchically nested, and XML data model inherently takes "answer by subtree model" (e.g., querying for `//foo` yields the whole subtree rooting at node `<foo/>`). Therefore, for any XML node, an action could be: authorized (or unauthorized) to the whole subtree, or partially authorized. The later case does not occur in relational access control model. Finally, in XML model, we can control the access right of each individual node.

In traditional relational model, the smallest granularity that one may control is a column via GRANT/REVOKE. Therefore, one needs to employ more recent developments of RDBMS access controls (e.g., Oracle VPD) to enable row/cell level access control.

In the second part of this dissertation, we propose a generic analysis to XML access control. We first analyze access control models to propose a formal description of XML access control using deep set operators. Then we articulate the problem of XML access control in XRDB as essentially the problem of XML/Relational object and operation equivalency and conversion. We show that, equivalent counterparts of deep set operators in relational model are needed to fully implement XML access control in XRDB. We analyze the definition and semantics of each operator, and show how they can be converted to XRDB through two lemmas. Although detailed conversion implementation is connected with the specific X2R conversion algorithm used in XRDB, we propose an algebraic description of these operators.

Moreover, we study possible implementations of XML access control in XRDB. We categorize them into three approaches, and formally describe the semantics of each approach using deep set operators. We also discuss the features and considerations of each approach. Finally, we show the validity of our approaches using experiment results.

We have carefully explored the problem space, proposed theocratical solutions, and discussed implementation approaches. However, there are still open questions in XRDB access control, especially the questions connected with particular implementation methods. E.g. how to enforce XML access control with minimal overhead and alternation upon underlying RDBMS? We leave these as our our future research topics.

Key contributions.

1. To our best knowledge, this work is the first one to algebraically formalize XML access control in both native XML (XDB) and RDBMS-supported XML (XRDB) environment.

2. We introduce a general framework to capture the design principles and operations of existing (and future) XML access control mechanisms. Inherent

pros and cons of each approach is intensively analyzed [44].

3. We developed *deep set operators* as extensions of regular set operators defined in XPath and XQuery. With deep set operators, we are able to formally describe XML access control at algebraic level.

4. We propose QFilter as an efficient, view-free, NFA-based XML access control enforcement mechanism, whose performance is superior to existing approaches [45]. Moreover, QFilter is independent from underlying XML engine, which brings great flexibility and application potential [41] [45] [1].

5. This work takes the first steps to define the *equivalent objects* and *equivalent operations* between native XML and XRDB systems. With this concept, we can migrate all the exciting features of native XML systems into XRDB by converting the atomic operations into equivalent relational counterparts. In this dissertation, we take the feature of fine-grained XML access control for a pilot study, and the results are encouraging [43].

6. This work shows for the first time that the "security" of XRDB can be achieved by finding the "equivalent" relational operators for three specific deep-set operators. This finding provides a viable way to build secure XRDB systems.

The rest of this dissertation is organized as follows: in Chapter 2, we introduce the background . In Chapter 3, we present *deep set operators*, which we implemented as an extension of regular set operators in XQuery and XPath. In Chapters 4 and 5, we answer our first research question. We first identify the need and potential of non-view based XML access controls, and examine three different approaches to implement XML access control enforcement mechanisms (Chapter 4). Then we present the design and implementation of QFilter using Non-deterministic Finite Automata (NFA); we conduct extensive performance evaluation on QFilter and other approaches (much more extensive than what we did in [45]). Results show that QFilter is very efficient and scalable. In Chapter 6, we formally introduce RDBMS-supported XML database systems. We describe the internal process in such systems at algebra level, and formulate the access control problem in XRDB.

In Chapters 7 and 8, we give solutions to this problem at two levels: theoretically, we show the whether and why fine-grained XML access control could (or could not) be enforced using RDBMS security features; practically, we show how to implement the enforcement methods. Finally, we wrap-up this dissertation, and address possible future works.

Chapter 2

Background and Related Work

2.1 XML Model and Native XML Database Systems

Extensible Markup Language (XML) 1.0 [10] and 1.1 [11] are currently W3C recommendations. XML is a general-purpose specification for creating markup languages. Quoting W3C XML Specification [11], its design goals are:

> *XML shall be straightforwardly usable over the Internet.*
>
> *XML shall support a wide variety of applications.*
>
> *XML shall be compatible with SGML.*
>
> *It shall be easy to write programs which process XML documents.*
>
> *The number of optional features in XML is to be kept to the absolute minimum, ideally zero.*
>
> *XML documents should be human-legible and reasonably clear.*
>
> *The XML design should be prepared quickly.*
>
> *The design of XML shall be formal and concise.*
>
> *XML documents shall be easy to create.*
>
> *Terseness in XML markup is of minimal importance.*

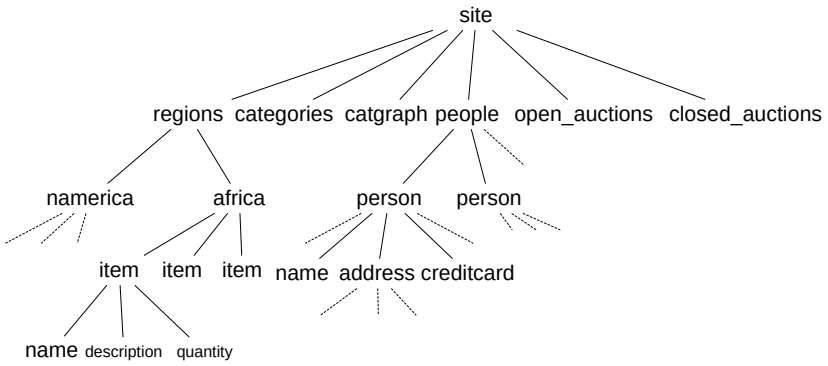


Figure 2.1. XMark DTD.

Although XML is usually regarded as a markup language, it is indeed a specification for creating other markup languages. XML documents are plain texts, including tags and data values. Internally, they represent tree-structured hierarchical data. XPath [3], XSLT [51] and XQuery [8] are developed to access nodes in XML documents. More specifically, XPath is a language used to query elements in XML trees. XQuery is a more complicated query language that provides flexible data manipulation functions. It is said that XQuery is to XML what SQL is to database tables. However, since XML model is considerably more powerful than relational model, XQuery is much more complicated than SQL. Due to the fact that XQuery uses XPath to access XML data, in XML access control research, people focus on XPath.

The volume of XML data explodes as XML model becomes the *de facto* standard in Internet information sharing and exchanging. Storing XML documents in text files is certainly not efficient or effective. Hence, XML database systems naturally appear. Native XML database system store XML in its own data structure. In [32], TAX algebra is proposed for tree-structured data, in accordance with the relational algebra for relational data. TIMBER [56] is developed based on this algebra. Among other native XML database systems such as Sedna [1], eXist [2], Tamino[65], Galax [67] is one of the complete implementation of XQuery.

[1] http://modis.ispras.ru/sedna/
[2] http://exist.sourceforge.net/

```
<site>
 <people>
  <person id="person0">
   <name>Ayonca Vijaykrishnan</name>
   <emailaddress>mailto:Vij……</emailaddress>
   <phone>(477) 63141558</phone>
  </person>
  <person id="person1">
   <name>Vidar Reinsch</name>
   <emailaddress>mailto:Vid……</emailaddress>
   <address>
    <street>70 Zlatev St</street>
    <city>Greenville</city>
    <country>United States</country>
    <zipcode>22021</zipcode>
   </address>
   <creditcard> 8814 4441 4702 6117</creditcard>
  </person>
  ……
 </people>
 …...
</site>
```

Figure 2.2. Part of the original XMark document.

Throughout the rest of this dissertation, we use the online auction DTD of XMark [64] as the exemplar schema, shown in Figure 2.1. It simulates an e-commerce scenario: online auction. It is used to store category, people, item, and auction information. Part of the XML document is shown in Figure 2.2. As we can see, XML documents are pure text files with tags. XML elements are nested, to represent a tree structure, as shown in Figure 2.3.

2.2 RDBMS-supported XML Database Systems

As we introduced, great amount of XML data still resides in well-developed relational database management systems (RDBMS). We call RDBMS-backed XML database systems *XRDB*. As illustrated in Figure 2.4, in an XRDB system: XML documents (D_X) are first converted into relations (D_R) using some conversion algorithm (C). D_R is stored and managed in RDBMS. In this scenario, any RDBMS could be used; user issues XML query Q_X (XPath or XQuery) using published

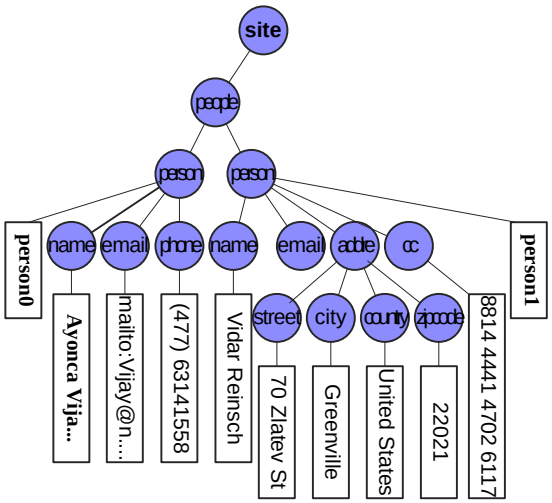


Figure 2.3. XMark document.

XML schema; Q_X is then converted into Q_R (in SQL) and evaluated against D_R. Relational answer A_R is finally converted back to XML answer A_X and returned to user.

These systems take advantage of the availability and maturity of existing RDBMS. We only need to develop query and data conversion mechanism. Data storage, query evaluation, transaction management and many other issues are handled by underlying RDBMS. Most relational database vendors implement this approach to provide XML support.

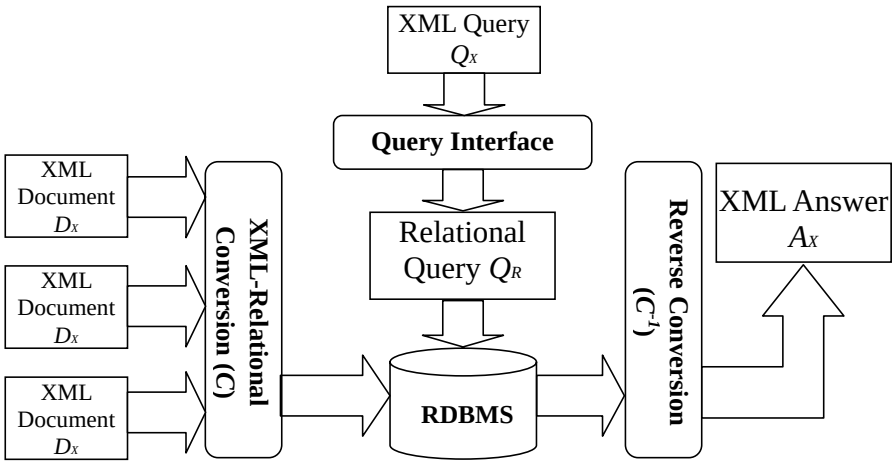


Figure 2.4. Overview of XRDB system architecture.

2.2.1 XML and Relational Conversion Algorithms

Toward conversion between XML and relational models, an array of research has addressed the particular issues lately. On the industry side, database vendors are busily extending their databases to adopt XML types. Shredding and non-shredding are two major pathes that followed by commercial products. Oracle provides both un-shredded (CLOB) and shredded storage options [54]. Microsoft supports XML shredding and publishing through mid-tier approach in SQL Server 2000, and adds CLOB storage in SQL Server 2005 [61]. IBM proposes the first native XML storage in DB2 9, but shredded XML storage (through schema decomposition) is still kept as an important feature [55, 6].

On the research side, various proposals have been made recently, mainly either schema-based (e.g., [20, 66, 39]) or schema-oblivious (e.g., [26, 72]) approaches. In terms of access control, some commercial products apply existing column level access control of RDBMS on XML data stored in CLOB columns. None of these approaches supports or discusses fine-grained access control. Finally, to our best knowledge, the only work that is directly relevant to our proposal is [69]. [69] proposes an idea of using RDBMS to handle XML access controls, in a rather limited setting. In our vision paper [40], we addressed some issues and challenges of enforcing XML access control atop RDBMS. We provide the algebraic analysis and explore practical solutions in this dissertation.

2.3 XML Access Control

XML access control is to ensure that only authorized users could access data they are allowed to access. It is critical to have access control in XML database systems. Figure 2.5 gives an example of XML access control on the XMark document, which we have shown in Figures 2.2 and 2.3. In this example, managers are able to see everything; sales are able to see `<people>` nodes but not `<creditcard>` nodes; and normal users can only see `<name>` and `<email>` tags.

There are two major research areas in XML access control – *models* and *enforcement mechanisms*. The focus of our work is on the latter. However, we first survey existing access control models.

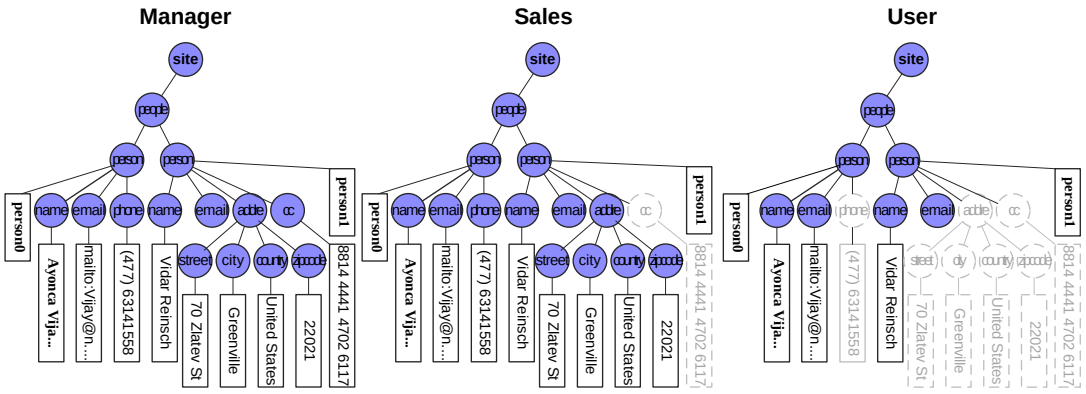


Figure 2.5. Example of XML access control.

2.3.1 Access Control Models

Most XML access control models inherit the framework of either role based access control [63], in which users are assigned with roles and thus can exercise certain access rights characterized by their roles, or credential (attribute)-based access control, where each user features a set of attributes and access rights are denoted based on the value of attributes. As a whole, the difference between role-based access control and credential-based access control is mainly the way they identify *subjects*, i.e., users. However, this is not closely related to our topic, since we focus on access control enforcement, which mainly considers *objects*, i.e. XML data.

Recently, several authorization-based XML access control models are proposed. In [19], a specific authorization sheet is associated with each XML document/DTD expressing authorizations. In [16], the model proposed in [19] is extended by enriching authorization types supported by the model, providing a complete description of the specification and enforcement mechanism. Among comparable proposals, in [4], an access control environment for XML documents and some techniques to deal with authorization priorities and conflict resolution issues are proposed. In terms of XML data objects, [27] propose a framework to normalize data object specification in XML access control using XPath. They also capture access control policy specification in existing literature with the proposed framework. Moreover, languages for access control policy are developed in such efforts as XACL by IBM [37] and XACML by OASIS [29]. While these are languages to specify access controls, what we propose here is a method to enforce access controls. Finally, the

use of authorization priorities with propagation and overriding are related to similar techniques studied in OODB [24, 60]. The above XML access control models can specify the authorizations of a subject against an XML data object without ambiguity. While an XML access control model can be enforced in various ways, the model cannot tell which enforcement mechanisms are better ones.

We will be using RDBMS access control features in the second part of this thesis, thus we quickly summarize relational access control. Relational access control models can be classified into three categories: *multilevel security models* [35, 70, 62], *discretionary security models (DAC)* and *role-based security models (RBAC)*. Most real world database systems implement a table/column level DAC similar to the one implemented in System R [30]. View-based approaches is the traditional method to enable row-level access control, while Oracle's VPD is the most recent development. Finally, some advanced access control models (e.g., [33, 34]) are proposed in a more theoretical manner.

2.3.2 XML Access Control Enforcement.

XML access control enforcement mechanisms in native XML environment have been intensively studied in recent years. Generally speaking, they are categorized into four classes:

1. engine level mechanisms implement node-level security check inside XML database engine; they tag each XML node with a label [17, 15, 71] or an authorization list [73, 36], and enforce security check at query processing. Figure 2.6 gives an intuitive example of node tagging. During query processing, XML engine traverses the subtrees of all candidate answers, eliminates all inaccessible nodes from the final answer. This traversal seriously slows down query processing.

2. view based approaches build security views that only contain access-granted data [68, 23, 38]. Note that, when a view-based approach implements virtual views without materializing them, it is inherently a pre-processing approach. Figure 2.7 gives an example of views built for *manager*, *sales* and *user* roles as introduced in Figure 2.5. Since views are pre-built, during run time, users'

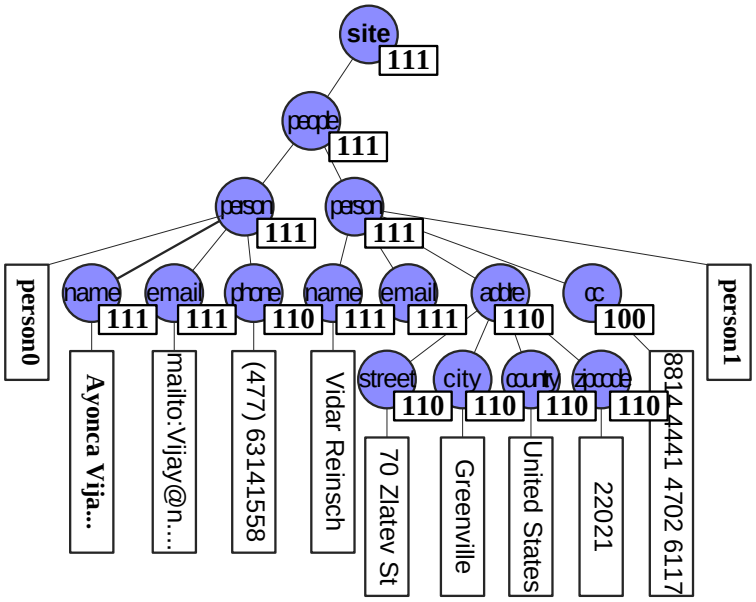


Figure 2.6. Example of engine level access control.

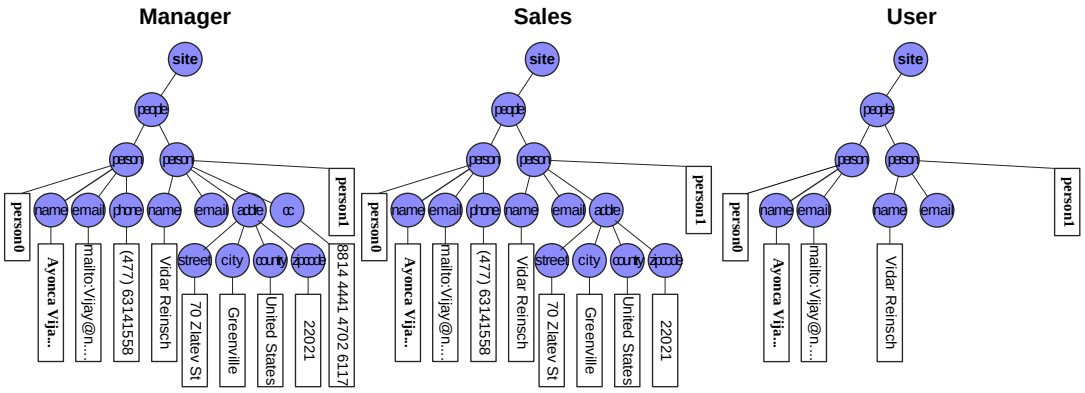


Figure 2.7. Example of view-based access control.

queries are evaluated directly against views, providing fast access. However, it is challenging to maintain a large number of (frequently updating) views for a large number of security roles. Especially, storage and view synchronization are two major concerns.

3. pre-processing approaches check user queries and enforce access control rules before queries are evaluated, such as the static analysis approach [52, 53], QFilter approach [45], function-based approach [59], access condition table approach [57] policy matching tree[58], secure query rewrite (SQR) approach

[49], etc.

4. [9] considers access control of streaming XML data and apply security check at client side, using a filtering mechanism. [25, 48].

Moreover, [5, 14] focus on XML access control policies and enforcement as well as encryption issues in information pushing or brokerage systems. More recently, protecting the privacy and security associated with XML tree structure (instead of content) becomes an emergent topic [25, 50].

To our best knowledge, *Static Analysis* [52] is the first attempt of non view-based XML access control. It first converts an input query q to an NFA M_q and access control rules r to another NFA M_r. Then, it (1) accepts q if $M_q \subseteq M_r$ (i.e., q asks for data that are "entirely" authorized), or (2) rejects q if $M_q \cap M_r = \emptyset$ (i.e., what q asks for is "entirely" prohibited). Therefore, when the pair of q and r belongs to one of two cases, access controls can be determined immediately. However, the static analysis method cannot handle the remaining cases where q and r partially overlap, and depends on the security features of underlying databases to determine its access controls. As shown in Section 5.4, since the majority of q and r belong to the partial-overlapping cases, the performance of [52] often severely suffers. Therefore, our QFilter approach, first appeared in [45], extends [52] so that the partial-overlapping case can also be handled without relying on underlying databases. By de-coupling the link between XML access controls and underlying databases entirely, therefore, QFilter becomes more practical and more efficient than [52].

2.4 Deep Set Operators

We proposed *deep set operators* in [46]. There are several related work that bear the same term but different semantics. In [13], "deep union" and "deep update" operators are proposed to process semi-structured data. This operator takes two edge-labeled tree-structural documents as input and merges/updates them based on their structural similarities. In [47], a deep-equal function is introduced to check the equality of two sequences. It checks if the arguments contain items that

are equal in values and positions. Despite the same name, their deep operators (functions) are different from ours in semantics or underlying operation objects.

2.5 Preliminaries

As we introduced, an XML document can be represented as a hierarchy of nested nodes (i.e., elements and attributes), so that fine-grained access controls at node level are established. A node-level authorization specified via 5-tuple access control rules (ACR) was proposed in [16], while many XML access control literature take similar ACR formats [52, 9]. In our model, users are assigned with roles and thus can exercise certain access rights characterized by their roles.

Node-level authorization in our study is specified via 4-tuple access control rules (ACR) = {**subject**, **object**, **action**, **sign**}, where (1) *subject* is to whom an authorization is granted (i.e., role); (2) *object* is part of an XML document specified by an XPath expression; (3) *action* consists of read, write, and update; (4) *sign* $\in \{+, -\}$ refers to either access "granted" or "denied", respectively. When a node does not have either explicit or implicit authorization, it is considered to be "access denied." It is possible for a node to have more than one relevant access control rule. If conflict occurs between "+" and "−" rules, "−" rules take precedence.

Compared with the 5-tuple ACR used in many related works, we do not have the "type" field. The 5-tuple ACR is usually represented as: $ACR = \{\mathbf{subject}, \mathbf{object}, \mathbf{action}, \mathbf{sign}, \mathbf{type}\}$. Particularly, *type* $\in \{LC, RC\}$ refers to either local check (LC) where authorization is applied to only attributes or textual data of nodes in context – "`self::text() | self::attribute()`" or recursive check (RC) where authorization is applied to current nodes and propagated to all their descendants – "`descendant-or-self::node()`", respectively.

A traditional approach is to convert ACR with RC type to a combination of three rules with LC type, as proposed in [52]. For instance, `/x` with RC type is semantically equivalent to three expressions: `/x`, `/x//*`, `/x//@*` with LC type. Therefore, by re-writing all rules with RC type into equivalent ones with LC type, we can focus on the construction and execution of rules with only LC type. However, in our model, "RC" type is enforced by default, i.e. access control specified on a node affects the whole subtree rooting at that node. This setting

Table 2.1. Example rules.

```
R1:  (role1, /site/categories//*, read, +)
R2:  (role1, /site/regions/*/item/location, read, +)
R3:  (role1, /site/regions/*/item/quantity, read, +)
R4:  (role1, /site/regions/*/item/name, read, +)
R5:  (role1, /site/regions/*/item/description, read, +)
 R5':  (role1, /site/regions/*/item[quantity>0]/location, read, +)
R6:  (role1, /site/people/person/name, read, +)
R7:  (role1, /site/people/person/address/*, read, +)
R8:  (role1, /site/people/person/emailaddress, read, +)

R9:  (role1, /site/regions/asia/item/location, read, -)
R10:  (role1, /site/regions/africa/item/location, read, -)
```

complies with the XML semantics, where a querying for a node yields the whole subtree. In other words, according to XML semantics, nodes are by default "RC". If a rule only applies to the text child of the context node, "`/text()`" is appended to the end of the XPath expression (object). In this way, we exactly follow XPath specification to identify XML nodes.

Although XQuery [8] is more powerful, like the other XML access control approaches, we choose XPath [3] for the specification of queries as well as the identification of nodes. XML data (nodes) covered by positive rules and not covered by any negative rule are considered *safe data*. XML query that only requests safe data is called *safe query*; and the answer is *safe answer*.

Example 2.1. We use the XMark schema of Figure 2.1 and XML access control rules of Table 2.1 for running examples. The schema demonstrates an online auction scenario. Rules `R1` to `R8` say that users of `role1` are permitted to access all "`categories`" information, some of "`item`" information, and some of "`person`" information. Initially, we only consider positive, who have no predicate in the XPath expressions of their *object* field. Then, rule `R5'` is referred when we demonstrate how predicates are processed. Rules `R9` and `R10` are used when we discuss negative rules.

Chapter 3

Deep Set Operators

3.1 Motivation

Before going to the details of XML access control mechanisms, we first introduce deep set operators, which we developed to formally describe XML access control.

XQuery [7] was developed by two W3C working groups to serve as the standard XML query language. In [7] and [22], three *set operators* are defined, namely `union`, `intersect` and `except`. In [7], they are defined as:

- *"The union and | operators are equivalent. They take two node sequences as operands and return a sequence containing all the nodes that occur in either of the operands."*
- *"The intersect operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in both operands."*
- *"The except operator takes two node sequences as operands and returns a sequence containing all the nodes that occur in the first operand but not in the second operand."*

According to [22], the set operators of XQuery are defined using the notion of node-IDs – unique (conceptual) ID per XML node. Therefore, for instance, the "union" of two node sequences are the union of node-IDs from both sequences. On the other hand, XQuery uses XPath [3] to locate nodes. As defined in XPath, once the query is processed and the final node-IDs, say {5, 7}, are found, the answer to

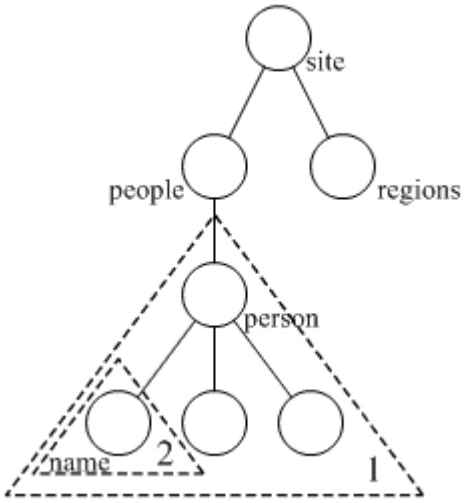


Figure 3.1. Tree structure of an XML document.

be returned to the user is the entire subtree rooted at the node with IDs 5 and 7. For instance, Figure 3.1, an XPath expression "//person" represents all the subtree rooted at <person> node (subtree 1), and "//person/name" represents the whole subtree rooted at "<name>" node (subtree 2), which is hierarchically nested as a subtree in subtree 1. In this way, all these operators are at node level, which treat nodes (node-IDs) as impartible unites and ignore the children or descendants of the elements.

In addition, the set operators defined in XQuery only require two operands to be "node sequences" without any further requirements on their comparability. Therefore, two operands may be sequences of nodes at different level, or even nodes from mixed levels. For instance, in the expression "//person `union` //name", two operands contain different nodes, <person> and <name>, and are thus incomparable regarding their semantics. In the relational model, this kind of union is not valid because of incompatible domains. However, XQuery accepts this query and would return a sequence of mixed nodes, <person> and <name>. Consequently, the "regular" set operators defined in XQuery are more flexible than their counterparts in the relational model, but they sometimes generate confusing semantics. In particular, we have observed that some XML applications would have been benefited greatly if there are "novel" set operators with different semantics in XQuery. We take XML access control as an example.

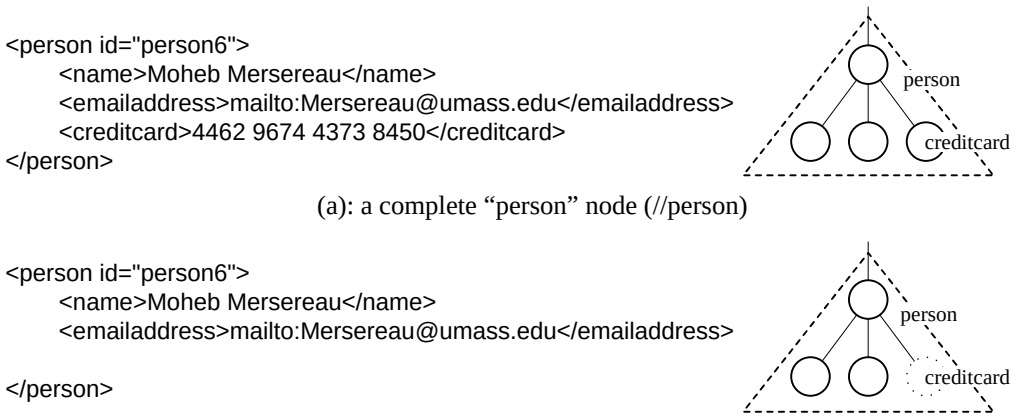


Figure 3.2. An example of `deep-except` semantics

Example 3.1 (XML Access Control)**.** In [45], we proposed an XML access control enforcement method that re-writes users' incoming query Q to a safe query Q' such that all fragments in Q that are asking for illegal data access are pruned out. In this context, conceptually, the safe query is either (1) "intersect" of Q and what users are granted to access (i.e., positive access control rules), or (2) Q "except" what users are prohibited to access (i.e., negative access control rules). That is, if Q is //person, but a positive access control rule grants only the data matching //person/name[age>18], then users must have an access to the data that are the intersect of //person and //person/name[age>18]. However, the expression using the regular `intersect` operator, "//person `intersect` //person/name[age>18]" would return NULL since no node-IDs from //person and //person/name[age>18] match. What is desirable here is, thus, a novel "intersect" operator that compares operands to check their structural overlap in a deep manner, and returns the overlapped region. That is, "//person `deep-intersect` //person/name[age>18]" should return the nodes matching //person/name[age>18] since it is completely nested in //person.

Symmetrically, for negative access control rules, we need novel `deep-except` operator. For instance, if a user issues a query //person but a negative access control rule prevents her from accessing the data matching //person/creditcard, then what she can really access is the data matching the expression "//person `deep-except` //person/creditcard". Again, the regular `except` operator, if used,

would have resulted wrong semantics. Figure 3.2 illustrates the above semantics of `deep-except`: (a) shows two <person> nodes in their original form, while (b) shows the expected output of the "deep" query "//person `deep-except` //person/creditcard".

Example 3.2 (Database as a Service)**.** In recent proposal to use database as a service model [31], query and data are delivered over to the database site which processes the query and returns answers back to users. Furthermore, XML data may be gathered and stored in a non-replicating fashion. A small company A that has branches in LA and NY may store different but partly overlapping XML data in both branches. When analysis needs to be done, the company ships query and two snapshots of XML data to 3rd party company B that provides database-as-a-service. For instance, the company A wants to gather aggregated statistics over items that were sold in 2004, and may request names of all items in the LA branch to be sent to B, while requesting complete item information of Northern America to be sent to B. That is, what B will receive is the "merged snapshot" of //item/name and //namerica/item. Like Example 3.1, this cannot be handled by the regular `union` operator, and can only be coped by introducing the novel `deep-union` operator as follows: "//item/name `deep-union` //namerica/item".

3.2 Formal Definitions

We first give precise formal definitions of three novel deep set operators, followed by illustrative examples.

3.2.1 Definitions

First, we denote node sequences as $P = \{p_1, ...p_n\}$ and $Q = \{q_1, ...q_n\}$, where p_i and q_i are XML nodes, identified by node-IDs according to XQuery semantics [22]. And the enumeration of the nodes and all their descendant nodes as:

$$\begin{aligned} P_d &= P/\texttt{descendant} - \texttt{or} - \texttt{self} :: * \\ Q_d &= Q/\texttt{descendant} - \texttt{or} - \texttt{self} :: * \end{aligned}$$

Definition 3.1 (deep-union)**.** `deep-union` operator ($\overset{D}{\cup}$) takes two node sequences P and Q as operands, and returns a sequence of nodes (1) who exist as a node or as a descendant of the nodes in "either" operand sequences, and (2) whose parent does not satisfy (1). Formally,

$$\begin{aligned} P \overset{D}{\cup} Q \;\; = \;\; & \{r | (r \in P_d \vee r \in Q_d) \;\wedge \\ & (r :: parent() \notin P_d \wedge r :: parent() \notin Q_d)\} \end{aligned}$$

In the above definition, condition (1) represents the foundamental semantics of deep union operator: compare not only nodes in operand node sequences but also their descendants; (2) serves as a supplement: when a node satisfies condition (1), all its descendants also satisfy condition (1), thus we wanted to eliminate the descendants and keep the "greatest common node" only [1]. Condition (2) is directly expressed as:

$$! \; (r :: parent() \in P_d \vee r :: parent() \in Q_d)$$

According to De Morgan's Law, it is equal to:

$$(r :: parent() \notin P_d \wedge r :: parent() \notin Q_d)$$

Similarly, we have

Definition 3.2 (deep-intersect)**.** `deep-intersect` operator ($\overset{D}{\cap}$) takes two node sequences P and Q as operands, returns a sequence of nodes (1) who exist as a node or as a descendant of the nodes in "both" operand sequences, and (2) whose parent does not satisfy (1). Formally,

$$\begin{aligned} P \overset{D}{\cap} Q \;\; = \;\; & \{r | (r \in P_d \wedge r \in Q_d) \;\wedge \\ & (r :: parent() \notin P_d \vee r :: parent() \notin Q_d)\} \end{aligned}$$

To formally define `deep-except`, we first need to define the `deep-except-node` operator. W3C XPath [3] standard limits the connection of any two nodes be

[1] According to XML standards, when this node is projected to the document, the whole subtree rooted at this node is returned.

in one (or more) of the twelve axis. For any two given nodes, we can further categorize their relationship into three classes: (1) they are identical, (2) they are ancestor-descendant, or (3) there is no overlap between them (including sibling, etc.). In other words, two nodes cannot be "partly overlapped". Then we define the `deep-except-node` operator as:

Remark 1. `deep-except-node` operator takes two nodes as operands, processes them according to the following conditions: (1)when the first node is equal to the second node, or is a descendant of the second node, return **null**; (2) when the second node is a descendant of the first node, remove it from the subtree of the first node and return the remaining; (3) otherwise, when there is no overlap between the first and second nodes, return the first node.

In addition to Remark 1, we extend the second operand to a "node sequence" to define `deep-except-nodeseq`:

Remark 2. `deep-except-nodeseq` operator takes one node as the first operand and one node sequence as the second operand, process them according to the following conditions: (1)when the first operand is equal to any node in the second operand, or is a descendant of any node in the second operand, return **null**; (2) when any node(s) of the second operand is descendant(s) of the first operand, remove it(them) from the first operand and return the remaining; (3) otherwise, when there is no overlap between the first and second operands, return the first operand.

Finally, `deep-except` operator is defined as follows.

Definition 3.3 (deep-except)**.** `deep-except` operator $(\overset{D}{-})$ takes two node sequences as inputs, and conducts `deep-except-nodeseq` operation between each node in the first operands vs. the second operand, and combine the outputs. Formally,

$$P \overset{D}{-} Q \;=\; \{r | r \in (p_i \texttt{ deep} - \texttt{except} - \texttt{nodeseq } Q)\}$$

Here we can see that the definition of `deep-except` operator appears to be different from the other two deep set operators. The differences are further discussed and explained in Section 3.3.

3.2.2 Examples

Consider the following XML fragment:

```
<a> <b> <c/> </b> <d/> </a>
```

Query "/a `union` //b" yields both <a> and <b> nodes. When projected on the document, the answer would be:

```
<a> <b> <c/> </b> <d/> </a>, <b> <c/> </b>
```

On the other hand, query "/a `deep-union` //b" yields <a> nodes only. When projected on the document, the answer would be:

```
<a> <b> <c/> </b> <d/> </a>
```

Query "/a `deep-intersect` //b" yields <b> nodes:

```
<b> <c/> </b>
```

Finally, query "/a `deep-except` //b" yields newly constructed <a> nodes, which is different from original <a> nodes:

```
<a> <d/> </a>
```

As another example of deep set operators, we compare deep set operators and regular set operators at micro level: given two nodes (i.e. only one item in each node sequence as operand), what could be the productions of deep set operations, as well as regular set operations?

As we pointed out, the relationship between two nodes A and B can only be one of the following: (1)they are the same (A=B); (2) A is an ancestor of B (//A//B)[2]; or (3) they are not related (no overlap between them). Table 3.1 summarizes the results of conducting regular set and deep set operators on two nodes of each category.

As an example, if we take a node (e.g. //person[@id='1']) and one of its grandchild (e.g //person[@id='1']/address/zip) as operands to conduct three set operations defined in XQuery, they will generate the results as shown in Figure 3.3. As we can see, regular set operators compares the IDs of two nodes and found them different. Thus union operation returns both nodes, intersect operation returns NULL, and except operator returns the first operand (grandparent node).

[2]For case "A is a descendant of B" (//B//A), we can simply swap token A and B, thus it is still categorized as case 2

	$\cup$	$\overset{D}{\cup}$	$\cap$	$\overset{D}{\cap}$	-	$\overset{D}{-}$
A=B	A	A	A	A	$\emptyset$	$\emptyset$
//A//B	{A, B}	A	$\emptyset$	B	A	partial content
no overlap	{A, B}	{A, B}	$\emptyset$	$\emptyset$	A	A

Table 3.1. Comparison between set operators and deep set operators

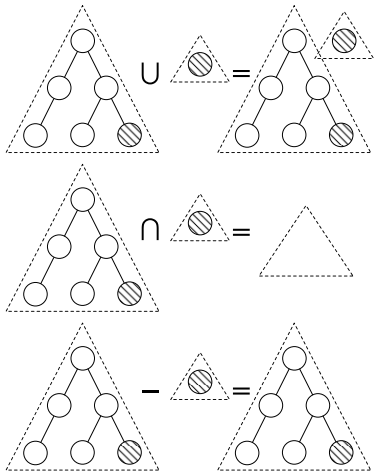


Figure 3.3. set operators defined in XQuery

On the other hand, if we take the above nodes and conduct deep-set operations, different results are generated, as shown in Figure 3.4:

- **deep-union** operator detects that the node of the second operand is a descendant of the first operand, so it returns only the ancestor node.
- **deep-intersect** operator also finds that the second operand is descendant of the first one, then their `deep-intersect` is the second operand.
- **deep-except** operator detects the second operand is a descendant of the first operand. In this way, the output of `deep-except` operator is not equal to any existing nodes. Rather, it is partial content of the ancestor node, with one of the descendant node been removed.

The above example is case 2 in Table 3.1. Comparing Figure 3.3 with Figure 3.4, we can observe the essential difference between regular set operators and deep set operators: the regular set operators only compare and process node(s) in operands, while deep set operators compare and process node(s) as well as their descendants.

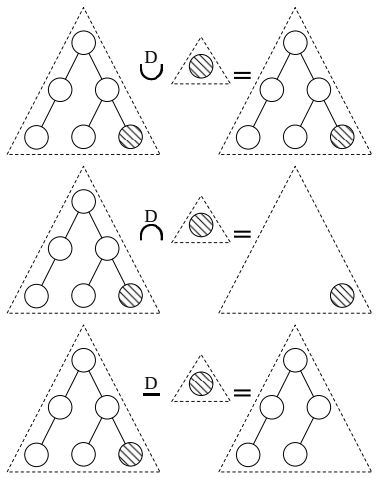


Figure 3.4. Deep set operators

3.3 Properties

3.3.1 Arithmetic Properties

The following properties of deep set operators are most fundamental and can be easily proved by their definitions (here we omit the proof due to space limit). They are similar to the properties of regular set operators.

Commutativity

$$P \overset{D}{\cup} Q = Q \overset{D}{\cup} P$$
$$P \overset{D}{\cap} Q = Q \overset{D}{\cap} P$$
$$P \overset{D}{-} Q \neq Q \overset{D}{-} P, \quad \text{unless} \quad P = Q$$

Associativity

$$\begin{aligned}(P \overset{D}{\cup} Q) \overset{D}{\cup} R &= P \overset{D}{\cup} (Q \overset{D}{\cup} R)\\ (P \overset{D}{\cap} Q) \overset{D}{\cap} R &= P \overset{D}{\cap} (Q \overset{D}{\cap} R)\end{aligned}$$

Distributivity:

$$(P \overset{D}{\cup} Q) \overset{D}{\cap} R = (P \overset{D}{\cap} R) \overset{D}{\cup} (Q \overset{D}{\cap} R)$$

In addition, we would like to point out one essential difference between `deep-except` operator and the other two. As we can see, both `deep-union` and `deep-intersect` operators return a sequence of nodes that are originated from the given XML tree, i.e. they do not create any new nodes. In this way, the production of these two operators are available for any other operations defined in XQuery that accepts nodes as operands, e.g. union, intersect, etc. On the other hand, whenever `deep-except-nodeseq` (case 2 of Remark 2) operation is conducted for `deep-except` operator, new nodes are constructed. Therefore, the production of `deep-except` operator may not be existing node in the XML tree. In this way, we should identify that although three operators are named "deep set operators" together, they are actually operators of different properties. `deep-except` operator is constructing new nodes while the other two operators return nodes of the original XML tree, which is similar to the regular union and intersect operators.

As an example, "//person `deep-except` //person/name" returns "person" nodes. However, the returned "person" nodes are not the same as "person" nodes that reside in original XML document: the "person" nodes produced by `deep-except` operation do not have "name" child. As a result, the newly constructed "person" nodes have new nodeIDs, which are different from original "person" nodes. Therefore, with the production of `deep-except` operator, we have to be careful when conducting further operations with existing "person" nodes in the XML document, such as union, intersect etc. Moreover, we cannot use

$$//person \overset{D}{-} //person/name \overset{D}{-} //person/age$$

although it looks fine in semantics. Instead, we have to use:

$$//person \overset{D}{-} (//person/name \cup //person/age)$$

3.3.2 Comparison with Set Operators

As we described above, `deep-union` and `deep-intersect` operators return nodes of the original document (probably node-IDs in actual applications). Here we provide two theorems to further describe the output of these two operators, especially their relationships with regular set operators.

Lemma 3.1. ***deep-union*** *of two node sequences is subset of their union production:*

$$(P \overset{D}{\cup} Q) \subseteq (P \cup Q)$$

Proof. Lemma 1 is equivalent to:

$$\text{if} \quad r \in P \overset{D}{\cup} Q, \quad \text{then} \quad r \in P \cup Q. \tag{3.1}$$

Suppose we have an r that $r \in P \overset{D}{\cup} Q$, according to Definition 1:

$$r \in P_d \quad \text{or} \quad r \in Q_d.$$

Consider the equivalency of P and Q, we can assume

$$r \in P/descendant - or - self :: * \tag{3.2}$$

According to Definition 1, we also have

$$r :: parent() \notin P/descendant - or - self :: *$$

which means

$$r \notin P/descendant :: * \tag{3.3}$$

Comparing (2) and (3), we have

$$r \in P, \quad \text{thus} \quad r \in P \cup Q,$$

which proves Equation 1. □

Lemma 3.2. ***deep-intersect*** *of two node sequences is subset of their union production:*

$$(P \overset{D}{\cap} Q) \subseteq (P \cup Q)$$

On the other hand, it is not possibly subset of their intersect production:

$$\exists P, Q \quad that \quad (P \overset{D}{\cap} Q) \nsubseteq (P \cap Q)$$

This theorem is also described as : if $r \in P \overset{D}{\cap} Q$, then $r \in P \cup Q$, but not always $r \in P \cap Q$.

Lemma 3.3. *`Deep-intersect` of two node sequences is superset of their intersect production, unless both operands contain ancestor-descendant nodes:*

$$(P \cap Q) \subseteq (P \overset{D}{\cap} Q)$$

The proofs of the above two theorems are similar to that of Lemma 1, and thus omitted.

3.4 Preliminary Implementations

We have implemented the deep set operators through user-defined functions of XQuery. With this implementation, these operators are executable in any XML engine that supports XQuery. On the other hand, as a drawback, this engine-independent implementation may not be as efficient as implementations at lower level (say, engine level).

As XQuery's user-defined functions, our implementations take node sequences as inputs, including XPath expressions and other forms of node sequences (e.g. products of set operators). In addition, as described above, the results of both `deep-union` and `deep-intersect` operations are XML node sequences and are available to further XQuery operations. Our implementation also supports this property.

3.4.1 `Deep-union` Operator

According to theorem 1, product of `deep-union` operator is a subset of regular union operation, i.e. each output node must originally resides in at least one operand. Following this theorem, `deep-union` operator, as shown in Algorithm 1, enumerates the nodes in each operands, referring to requirements of `deep-union` and return the satisfied ones.

Algorithm 1: `deep-union`

Input: input node sequences P and Q
foreach *node* P_i *of* P **do**
  if *empty*($P_i \cap Q//*$) *&* *empty*($P_i \cap P//*$) **then**
    P_i
foreach *node* Q_i *of* Q **do**
  if *empty*($Q_i \cap (P \cup P//*)$) *&* *empty*($Q_i \cap Q//*$) **then**
    Q_i

3.4.2 `Deep-Intersect` operator

`deep-intersect` is implemented in a similar way as `deep-union`. According to theorem 2, each output node of `deep-intersect` operator must originally resides in at least one operand. To enhance the readability of the algorithm, we divide it into three steps: first extract the regular "intersect" of two operands, remove the possible ancestor-descendant relationship that may exists, and output the remaining. Then, for each of the operand, enumerate the node items, output it if it is a descendant of the other operand (some exceptions are eliminated). Algorithm 2 shows how `deep-intersect` works.

Algorithm 2: `deep-intersect`

Input: input node sequences P and Q
foreach *node* r *in* $(P \cap Q)$ **do**
  if *empty*($r \cap (P \cap Q)//*$) **then**
    r
foreach *node* P_i *of* P **do**
  if *empty*($P_i \cap P//*$) *&* *!empty*($P_i \cap Q//*$) *&* *empty*($P_i \cap (P \cap Q)$) **then**
    P_i
foreach *node* Q_i *of* Q **do**
  if *empty*($Q_i \cap Q//*$) *&* *!empty*($Q_i \cap P//*$) *&* *empty*($Q_i \cap (P \cap Q)$) **then**
    Q_i

3.4.3 `Deep-except` Operator

For `deep-except` operator, as we have described in Section 3.1, it is different from the other two deep set operators. We implement it in a recursive manner: for each

node in the first operand, (1) if it has no overlap with any node in the second operand, output it; (2) if it is ancestor of any node(s) in the second operand, construct a new node with the same name, attributes and text, and enumerate all the children to conduct `deep-except` with the second operand; (3) otherwise, the node is "covered" by nodes in the second operand, eliminate it. The general algorithm is shown in Algorithm 3.

Algorithm 3: `deep-except`

```
Input: input node sequences P and Q
foreach node P_i in P do
    if empty(P_i//* ∩ Q) and empty(P_i ∩ (Q ∪ Q//*)) then
        P_i
    if ! empty(P_i//* ∩Q) then
        construct element{
        element_name=name(P_i);
        element_attributes=P_i/@*;
        element_text()=P_i/text();
        deep-except(P_i/*,Q);
        }
    else
```

3.4.4 Complexity

Although the above preliminary implementations may not be fully optimized, we can still estimate the computation of deep set operators. The computation of `deep-union` and `deep-intersect` operators are both $O(n_c * n_s)$, where n_c denotes the total number of nodes in the sequences of operands, and n_s denotes the total size of the subtrees rooted at these nodes. On the other hand, the computation of `deep-except` operator (P `deep-except` Q) is denoted as $O(n_{cq} * n_{sp} * i)$, where n_{cq} denotes number of nodes in P, n_{sp} denotes the size of subtrees rooted at nodes in P, i denotes the maximum depth of these subtrees. This appears to be more expensive than the other two, since recursive function call is employed in the implementation. It could be greatly optimized if implemented at XML engine level.

Chapter 4

XML Access Control Enforcement Mechanisms

In this section, we introduce a general framework, which capture the design principles and operations of existing XML access control mechanisms. Under this framework, we observe that most existing XML access control mechanisms share the same design principle with slightly different orderings of underlying building blocks (i.e., data, query, and access control rule). Furthermore, according to the framework, we identify four plausible approaches to implement XML access controls, namely built-in, view-based, pre-processing and post-processing. For each approach, we algebraically describe it with deep set operators.

4.1 System Architecture - a General Framework

We view the XML access control mechanism as the interplay of three building blocks – *data*, *query*, and *access control rule* as follows:

- **Data (D)** indicates the XML data (or document) that contains the answers users are looking for. Often the data are stored in native XML engines or RDBMS, but the choice of storage system is irrelevant to the discussion of our discussion.

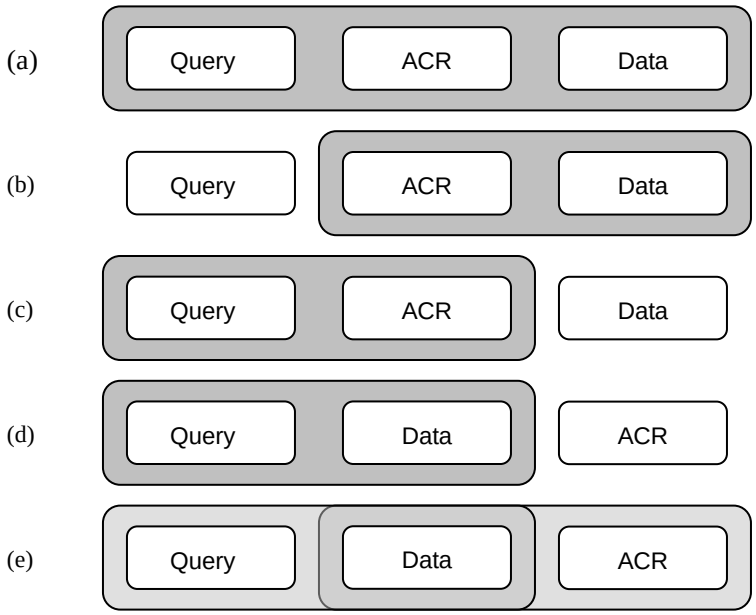


Figure 4.1. Different combinations of building blocks in the framework.

- **Query (Q)** describes the information that users want, and can be viewed as a conceptual pointer to the desired data in D. In XML domain, query is often written in either XPath or XQuery language. When a Q is issued by a user, Q has the same security role as what the user has.

- **Access Control Rules (ACR)** is a list of 5-tuple access control rule, describing the security policy of some roles. When a portion of data in D that does not violate policies of ACR are returned, it is a "safe" answer.

Note that D, Q and ACR are independent components, and thus can be located independently and processed separately. Figure 4.1 illustrates various combinations of the three building blocks, where gray box implies that building blocks in it are (1) co-located (in a spatial sense); and/or co-processed (in a temporal sense). For instance, (a) can be interpreted as: all three building blocks must be (1) co-located in a single system; and/or (2) processed at the same time. Below, we will consider both aspects of the framework.

- (a) indicates a scenario where all three building blocks are co-located in a single system. For instance, conventional RDBMS supports relational access

control via the embedded support of GRANT/REVOKE. In such a setting, Q is issued against both D and ACR which are stored together;

- (b) is a slight modification of (a) in that Q can be issued remotely while ACR must be stored together with D in a system. Typical example of this scenario includes the client-server model such as web-based database interface. On the other hand, from the temporal aspect, (b) illustrates the view-based XML access control mechanism where ACR and D are processed first (yielding a safe view), and then Q is evaluated against the view. Whichever case it is, the data provider must be able to support XML access control mechanism;

- In the spatial sense, (c) indicates a scenario where one party holds Q and ACR, while D is stored elsewhere. For instance, D is provided by a data provider while ACR is provided by a data mediator who connects end users with raw data sources with marginal fees. Once acquiring an adequate security role from the mediator by paying the fee, end users can issue a query to D. On the other hand, in the temporal sense, (c) implies that Q and ACR can be pre-processed prior to D. Therefore, for optimization, one can "merge" Q and ACR such that new output Q' can be processed against D more efficiently;

- (d) shows a scenario where only ACR is stored elsewhere. Since Q and D are stored together, conventional databases without access control support can be used to first evaluate Q against D. When ACR itself carries security-conscious information and has to be stored securely, this approach can be adopted; and

- Lastly, (e) is a conceptual merge of (b) and (d). Since the final "safe" answers are those data that can pass through constraints of Q as well as ACR, one can do intersection of two data sets – one from evaluating Q against D, and the other from enforcing ACR against D.

Now, we analyze popular approaches, e.g. (b), (c) and (d) in details.

4.2 View-based Approach

When access control is first enforced on XML documents to create *views*, it is the traditional view-based approach. This approach is adopted from RDBMS access control, and is intensively studied in the literature. View-based approach takes advantage of the fact that ACR and D are either co-located or co-processed. By processing $evalRule(ACR, D)$ first, this approach produces a set of data, SD_1, ..., SD_n for each role, thus creating a number of "views". Since each view contains only "safe" data for that particular role, query can be processed on this view without any further special care, making the query processing very efficient. The examples of view-based approaches recently proposed include [73, 16, 4], and is one of the most popular XML access control mechanisms. Depending on the details of the algorithms, the views can be maintained either physically or virtually.

In this approach, XML view V (or *safe document* SD) is constructed to capture:

$$V = [(\langle R_1^+ \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R_n^+ \rangle) \overset{D}{-}_X (\langle R_1^- \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R_m^- \rangle)]$$

And query is evaluated against the view

$$SA = Q\langle V \rangle = Q[(\langle R_1^+ \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R_n^+ \rangle) \overset{D}{-}_X (\langle R_1^- \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R_m^- \rangle)]$$

Since the I/O and space costs for constructing views are amount to evaluating $evalRule(ACR, D)$, it is dependent on the number of roles in ACR and the size of D. The view-based approach generally includes three steps: (1) view construction (2) query processing (3) view maintaining. However, often, the view construction is performed off-line, and thus the cost issue becomes less important. When the space cost becomes a major issue due to large number of views (e.g., million roles in Internet environment), then one may mitigate the problem using the compression-based techniques suggested in [73].

However, this approach still has to take extra burden to maintain the views. When update occurs to either ACR or D, synchronization must be performed to views. Overall, the view-based approach is fast in answering user queries but may have to pay high I/O and storage cost, and the extra complexity of view maintenance.

4.3 Pre-processing Approach

The idea of the pre-processing approach is to view both user's query and security policies written in ACR as two *constraints to satisfy*. Therefore, security enforcement is ensured by "merging" two constraints to form tighter constraints. For instance, for Example 2.1 and Table 2.1, consider a user of *role1*, "John", who wants to survey the items' location information. He submits a query, `Q://item/location`. The meta-semantics of Q and a positive rule $R+$ is that users are allowed to access the regions scoped by "`Q DEEP-INTERSECT R+`". Conversely, that of Q and a negative rule $R-$ is "`Q DEEP-EXCEPT R-`". Collectively, what "John" is allowed to read is then denoted as:

```
(Q DEEP-INTERSECT (R1 DEEP-UNION R2...DEEP-UNION R8))
DEEP-EXCEPT (R9 DEEP-UNION R10)
```

Algorithm 4: Primitive

Input: Q, ACR, D
Output: SD
$S \longleftarrow$ all rules in ACR having the same "role" as Q ;
$P \longleftarrow$ rules in S with + sign, P_1, ..., P_i;
$M \longleftarrow$ rules in S with - sign, M_1, ..., M_j;
$Q' \longleftarrow Q$ DEEP-INTERSECT (P_1 .. DEEP-UNION .. P_i) DEEP-EXCEPT (M_1 .. DEEP-UNION .. M_j);
$SD \longleftarrow Q'(D)$;

Note that only `R2`, `R9` and `R10` are related to the interest of "John". However, the primitive approach does not analyze the *object* field of rules to further distinguish corresponding rules. The formal algorithm, **Primitive**, is given in Algorithm 4. Primitive approach is built using deep set operators. This way, the final safe query Q' does not require any special security-related support from the underlying XML engine (i.e., deep-set operators are implemented as user-defined functions). The semantics and algorithm of the primitive approach is simple and clear, and thus can be easily implemented. On the other hand, the primitive approach may generate complex safe queries, especially when there are a large number of access control rules. Since such complex queries tend to be expensive to evaluate, the primitive approach is further improved in the pre-processing approach below.

One may improve the primitive algorithm by optimizing the modified query

Q' further. That is, instead of simply generating a complicated Q' with multiple deep-set operators interweaved, one may do some "pre-processing" by exploiting the specifics of XML model and XML access controls. For instance, if what users ask for are entirely prevented by ACR, then we can return null to users outright. Similarly, if users ask for data that are entirely granted, then no further security check is needed. Lastly, among the data that users ask for, if only part is granted by ACR, then it is beneficial to rewrite Q to Q' such that fragments asking for illegal data are pruned away.

In preprocessing approaches, *safe query* SQ is constructed as:

$$SQ = Q \overset{D}{\cap}_X [(\ R_1^+ \ \overset{D}{\cup}_X ... \overset{D}{\cup}_X \ R_n^+) \ \overset{D}{-}_X \ (R_1^- \overset{D}{\cup}_X ... \overset{D}{\cup}_X R_m^-)]$$

Safe answer is yielded by evaluating safe query against the original document: $SA = SQ\langle D\rangle$.

The **Pre-Processing** algorithm is shown in Algorithm 5 (the details of the function "Pre-Processing(Q,ACR)" are elaborated in Section 5).

Algorithm 5: Pre-Processing

Input: Q, ACR, D
Output: SD
$S \longleftarrow$ all rules in ACR having the same "role" as Q;
$Q' \longleftarrow$ Pre-Processing(Q,ACR);
$SD \longleftarrow Q'(D)$;

4.4 Post-Processing Approach

The post-processing approach extends regular query processing by going through a "post-filtering" stage, named as AFilter, to filter out un-safe answers. Despite their potential inefficiency for unnecessarily carrying unsafe data till the last step, this approach is simple to implement. Moreover, when ACR and data are stored separately in a distributed environment (e.g., database-as-a-service model), this approach can be useful. The formal algorithm, **Post-Processing**, is given in Algorithm 6. However, despite the simple look on the surface, its implementation needs to overcome the following technical issue. Let us again look at "John"'s query `Q: //item/location`: consider `Q` and `R9`, `R10`, which say that

"John" is not allowed to access location information of items in Asia or Africa. When Q is first evaluated against an XML document D, Q projects out only the tag `<location>` without its ancestor tags. Therefore, in the post-filtering stage, when `R9` and `R10` are to be enforced against these intermediate answers having only `<location>` tags, they cannot check whether the `<location>` satisfies `/site/regions/africa/item/location` or not. However, if underlying XML database can produce `<location>` as well as all its "ancestor" tags (e.g., using a recursive function of XQuery), then the post-processing approach by AFilter can be applied without any further security support from databases.

Access control through post-processing is described as:

$$SA = ACR\langle A\rangle = ACR\langle Q\langle D\rangle\rangle = [(\ R_1^+ \overset{D}{\cup}_X ... \overset{D}{\cup}_X\ R_n^+)\ \overset{D}{-}_X\ (R_1^- \overset{D}{\cup}_X ... \overset{D}{\cup}_X R_m^-)]\langle Q\langle D\rangle\rangle$$

Algorithm 6: Post-Processing

Input: Q, ACR, D
Output: SD
$S \longleftarrow$ all rules in ACR having the same "role" as Q ;
$P \longleftarrow$ rules in S with + sign;
$M \longleftarrow$ rules in S with - sign;
AFilter.constructAFilter(P,M);
$UD \longleftarrow Q(D)$;
$SD \longleftarrow$ AFilter.filter(UD);

4.5 A Qualitative Comparison

In this section, let us do a close examination on the above three (important) categories: view-based, pre-processing, and post-processing. We observe that typically an XML access control mechanism involves three separate operations: (1) off-line service preparation, (2) on-line query processing, and (3) service maintenance.

- **Off-line Service Preparation.** This step is typically devoted on tasks to help speed-up the subsequent query processing step, and done off-line. Obviously, view-based approach would need to generate views per roles in this step. Similarly, the pre-processing approach like QFilter or static analysis method spends this time on constructing needed data structures (e.g., NFA). For the post-processing approach, one can build up some kind of index on

ACR (e.g., given a "role", quickly retrieve all relevant rules from ACR) so that later post-filtering process can run faster. Note that in this stage, Q from users are not known, and both ACR and D are the sole resources. Therefore, often the cost for service preparation depends on the size of ACR and D. Moreover, when the preparation requires non-trivial probing of ACR such as QFilter case, the complexity of ACR also does affect the cost. However, overall, since these tasks are done off-line, they do not contribute much to the performance of whole XML access control mechanisms, and thus omitted in our experimental comparisons of Section 5.

- **On-line Query Processing.** Once Q is issued, the task of evaluating Q while ensuring security policies in ACR is done in this step, and must be done on-line (unless the submitted query is part of batch-process). The end output of this task must be the "safe answers". Thus, the end-to-end on-line query processing time is the time-line between Q and SA in Figure **??**.

  For the view-based approach, the query processing can be efficient since there is no need for additional security check (i.e., each view contains only safe data for the role, after all). For the pre-processing approach, the performance largely depends on the quality of the re-written query from the pre-processing. For instance, if the primitive method generates a re-written query Q' as "$s_1 \cap ... \cap s_n - t_1 ... - t_m$" ($n, m \gg 1$), then the evaluation of the Q' can be quite slow. Other pre-processing approaches like QFilter or static analysis method improve it drastically via early-pruning of access-full-granted or access-fully-denied cases and via improved query re-writing in $merge(Q, ACR)$. For the post-processing approach, the security check is pipelined after the query evaluation, and thus can be disadvantageous in terms of performance. Post-filtering time is highly dependent on the size of unsafe answer set.

- **Service Maintenance.** In general, any service preparations done off-line need to be maintained when update occurs. For instance, when D is changed (e.g., new sub-tree is inserted to D), view-based approach needs to (incrementally) re-construct relevant views. However, the changes to D

Approach	Preparation	Processing	Maintenance
View-based	Medium	Good	Medium
Pre-processing	Good	Medium/Good	Good
Post-processing	Good	Bad/Medium	Good

Table 4.1. Qualitative comparison of XML access control approaches.

do not affect the pre-processing or post-processing approach. On the other hand, when ACR is changed, it affects the pre-processing (e.g., an NFA needs to be updated) and post-processing approach (e.g., index on ACR needs to be updated).

The summary of the qualitative comparison of three scenarios of Figure 1 is summarized in Table 4.1. Note that the query processing cost of the post-processing approach heavily depends on the size of intermediate un-safe data and/or the complexity of rules in ACR.

Chapter 5

QFilter: An Implementation of Pre-Processing Approach

In this Chapter, we present our NFA-based implementation of the pre-processing approach, named QFilter. QFilter reads a query Q and access control rules ACR as input, and returns a modified query Q' as output:

$$Q' = \mathsf{QFilter}(Q, ACR)$$

This can be re-written by separating positive and negative rules:

$$\begin{aligned} Q' &= Q \overset{D}{\cap} (ACR^+ \overset{D}{-} ACR^-) \\ &= (Q \overset{D}{\cap} ACR^+) \overset{D}{-} (Q \overset{D}{\cap} ACR^-) \\ &= \mathsf{QFilter}(Q, ACR^+) \overset{D}{-} \mathsf{QFilter}(Q, ACR^-) \end{aligned}$$

That is, the **Pre-Processing** approach can be implemented by two invocations of QFilter function and a `DEEP-EXCEPT` operator.

Once QFilter is constructed from ACR, it "filters" out illegal fragments from incoming queries to produce only "safe" queries. In the filtering stage, in particular, QFilter has three types of operations: (1) **Accept**: If answers of Q are contained by that of ACR^+ (i.e., Q asks for answers granted by ACR^+) and disjoint from that of ACR^- (i.e., Q does not ask for answers blocked by ACR^-), then QFilter accepts the query as it is: $Q' = Q$; (2) **Deny**: If answers of Q are disjoint from

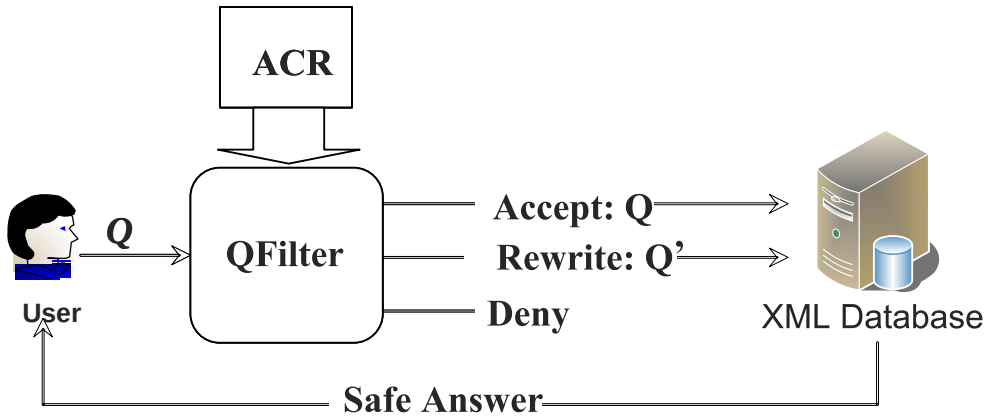


Figure 5.1. QFilter in a black box.

that of ACR^+ (i.e., no answers to Q are granted by ACR^+) or contained by that of ACR^- (i.e., all answers to Q are blocked by ACR^-), then QFilter rejects the query outright: $Q' = \emptyset$; and (3) **Rewrite**: if only partial answer is granted by ACR^+ or partial answer is blocked by ACR^- , QFilter rewrites Q into the ACR-obeying output query Q'.

Example 5.1. In Example 2.1, a user submits three queries:

```
Q1:/site/categories//*
Q2:/site/regions/asia/location
Q3:/site/regions/people/person/*
```

In comparing these with ACR of Table 2.1: (1) `Q1` is accepted by `R1`; (2) `Q2` is accepted by `R2` but rejected by `R9`, and is rejected since negative rules override positive rules; and (3) `Q3` is rewritten by `R6` and `R8` into:

$$\begin{array}{c} \texttt{/site/regions/people/person/name} \\ \overset{D}{\cup}\ \texttt{/site/regions/people/person/emailaddress}. \end{array}$$

5.1 QFilter Construction

In a nutshell, as shown in Figure 5.1, QFilter builds an Non-deterministic Finite Automata (NFA) from *Object* fields (in the form of XPath expressions) of ACR, and rewrites an input query Q according to one of the three operations. That

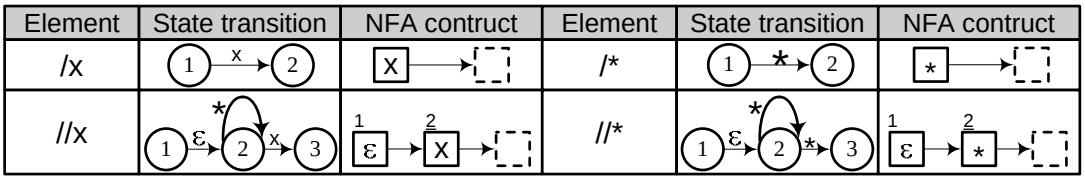


Figure 5.2. NFA element for each XPath building block

is, we view XPath expressions of ACR as compositions of "four" basic building blocks: `/x`, `/*`, `//x`, and `//*`. Complex XPath expressions with predicates (e.g., `/x[y='c']`) can also be handled and are further described in Section 5.2. The NFA element construction for each building block is illustrated in Figure 5.2. For a complete XPath expression, NFA fragments are constructed upon path elements and then linked in sequence. For a set of rules that form the ACR, NFA for each rule is constructed and all the NFAs are combined such that identical states are merged. The construction process is somewhat similar to that of regular NFA. As examples, Figure 5.3 is the state transition map (Left) and corresponding NFA (Right) of Example 2.1.

We first give a brief example of QFilter construction at the level of XPath steps, then we walk into details of QFilter data structure and construction algorithm. We construct the QFilter starting from R_1. For path step `/site`, we create state 0 and a transition on token "`site`" to state 1. Then a transition on token "`categories`" is created on path step `/categories`. For element `//*`, transition from state 2 to 3 and then 4 is created as shown in Figure 5.3 (left). Transition from state 3 to 4 requires at least 1 token after the ε-transition. We use the "next-token-driven ε-transition" in the NFA execution, thus state 3 and 4 could be merged in the NFA and set as acceptable state. The remaining access control rules are processed accordingly.

QFilter Data Structure

A regular NFA holds a state transition table at each state, mapping acceptable *tokens* to transition *states*. Since a QFilter captures *object* fields of access control rules in XPath, element names at each XPath step are identified as "tokens". Moreover, to capture each XPath step in one QFilter state, the predicates should also be captured. The data structure for QFilter (illustrated in Figure 5.4) consists

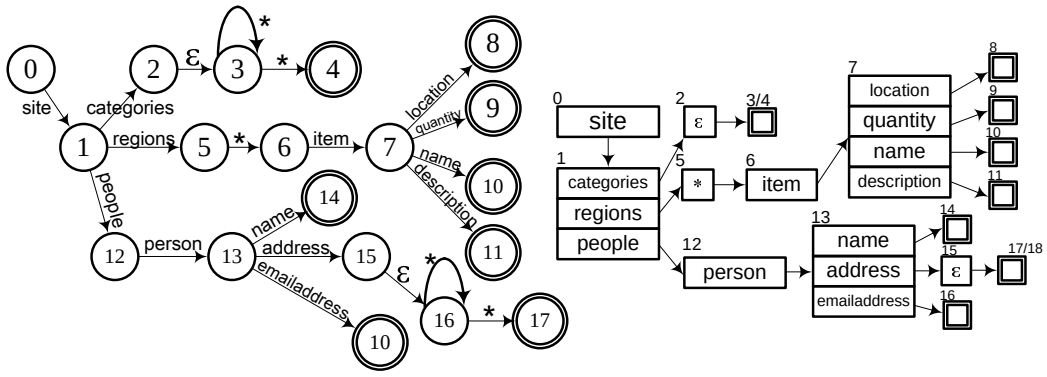


Figure 5.3. State transition map and NFA of the QFilter

of:

1. A state transition table called *stateTransitionTable* (implemented as a hash table), maps the element names of XPath to a *predicateTable*, that maps predicates to corresponding child *QFilterState*.

2. A ε-transition child state. Transition to this child state is automatically triggered without any token matching.

3. A binary flag to indicate an accept state (e.g., states 8 and 9 of Figure 5.3)

4. A binary flag to indicate a double slash state, which recursively accepts tokens (e.g., states 3/4 and 17/18 of Figure 5.3).

Taking ACR as input, we construct QFilter from the root state, and hold this state for all future access (e.g., add a rule or filter a query). We first create an empty root state, then add each rule to the root state one by one. The general idea for adding each rule is to follow the existing NFA states as much as possible, until no identical path exists, then new states are constructed. Algorithm 7 shows QFilter construction (at state level) in details.

Example 5.2. Let us demonstrate the construction of QFilter using Example 2.1. First, let us add a positive rule `/site/categories//*` into QFilter. State 0 is first created for the XPath step `/site`. Then, state 1 is created for the step `/categories`. States 2 and 3/4 are created for `decedent-or-self()` step `//*`. Finally, state 3/4 is marked as "*DSState*" and "*Accept*", indicating that it takes

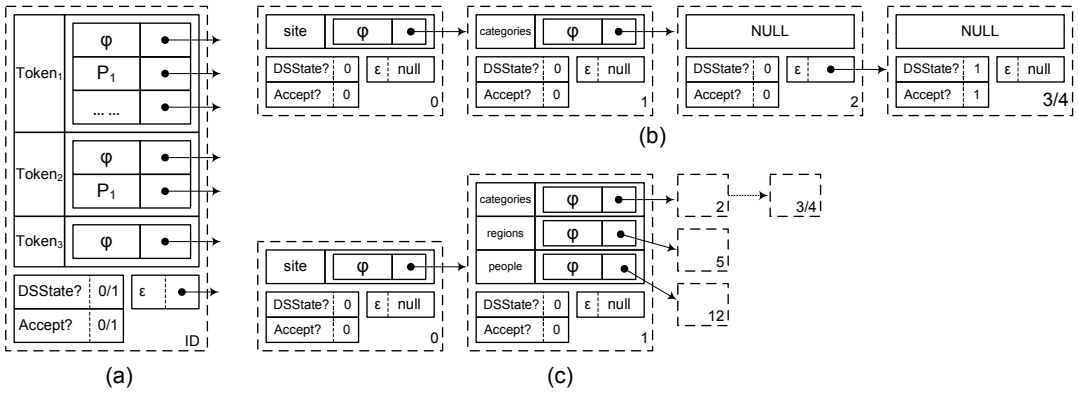


Figure 5.4. (a) Data structure of a QFilter state; (b) QFilter constructed for rule `/site/categories//*`; (c) QFilter constructed for more rules.

any input token (`//*`) and is an accept state. Next, we add another rule `R2: /site/regions/*/item/location`. For the first step "`/site`", since identical key is detected at state 0, it is reused. Then at state 1, element name "`regions`" is not in the state transition table. State 5 is created, and new entry is inserted into the state transition table at state 1. States 6, 7 and 8 are created in the same way. If we add all eight rules of Example 2.1 to this QFilter, state 1 is finally constructed as shown in Figure 5.4(c) (not all other states are shown in the Figure).

5.2 QFilter Execution

Given a query Q as input, QFilter filters unsafe fragments of Q to generate a safe query Q'. The filtering principle consists of: (1) If ACR allows all data that Q requests, keep Q as it is; (2) If what Q asks for is entirely prohibited by ACR, then reject Q outright; and (3) Otherwise, modify Q such that Q' returns a precise "deep-intersection" of Q and ACR. The filtering process becomes complicated when either Q or ACR has non-deterministic operators such as "//" and "*", which can match multiple branches in the NFA.

QFilter Execution Algorithm

At micro level, we first pass Q to the root state of QFilter to start NFA execution. Since queries with wildcards may go through several rules (being rewritten by each rule), the result of QFilter execution can be an array of *safe* XPath queries.

Algorithm 7: QFilterState.addRule

```
Input: XPath expression of Access Control Rule: R
1  if R.EOS() then { mark as accept state: acceptState = True;  return } ;
2  if (R.currentStep is "//") & (NOT R.doubleSlashProcessed) then
3      if ε − transitionChild does not exist then
4          Create ε − transitionChildState;
5          mark ε − transitionChildState as DSState;
6      R.doubleSlashProcessed ⟵ true;
7      ε − transitionChildState.addRule(R);
8      return
9  Token ⟵ R.elementName;
10 Predicate ⟵ R.predicate;
11 R.nextStep();
12 if DSState & Token = "∗" then { addRule(rule);
13 return;} ;
14 if NOT stateTransitionTable.hasKey(Token) then
15     stateTransitionTable.put(Token, emptyPredicateTable);
16 predicateTable ⟵ stateTransitionTable.get(Token);
17 if NOT predicateTable.hasKey(Predicate) then
18     create new filterState newState;
19     predicateTable.put(Predicate, newState) ;
20     stateTransitionTable.put(Token, predicateTable) ;
21     newState.addRule(R);
22 else (PredicateTable.get(predicate)).addRule(R);
```

Each element in the array reflects the rewritten branch of Q. Then, QFilter weaves the array of XPath queries through using $\overset{D}{\cup}$. Due to the design of access control rules, it is possible that one the the filtered branch is equal to the original input query, while some other branches reflect pruned input query. In this case, only the original query is kept. At the state level, the execution of QFilter is similar to that of regular NFA, except for its query re-writing process. The details, shown in Algorithm 8, are as follows:

- Starting from the root state, an element name of XPath and keys in the state transition table are compared to find a "match". When the "match" is found, keep the intersection of the element name and the key as the output of this state. For instance, when the"*" step in Q matches the key "`regions`" in the state transition table, their intersection becomes "`regions`". Predicates

from both ACR and Q are kept as the output of this state (details will be elaborated in Section 5.2).

- When a "match" is processed, QFilter locates the next state corresponding to the matched key and predicate from the state transition table. The input query also moves its pointer to the next step, and continues the execution.

- Q is accepted at the accept state. We then link the output of each state sequentially to obtain a final "filtered" output query. If a query is accepted at a state that is not "`//*`", "`text()`" is appended to the output, indicating that Q is accepted by an LC rule.

- At each step, multiple matches may exist (e.g., a "*" in Q matches all the keys in the state transition table). Then, QFilter execution is split into branches, and the final output of each branch (if not null) is put into the result array. On the other hand, when multiple predicates exist, QFilter execution is also split into branches. For multiple outputs (as array), they are connected by $\overset{D}{\cup}$.

Example 5.3. Let us use the QFilter in Example 5.1 to process a query:

`Q:/site/people/person/name`

The first step is accepted by state 0. Since the element name "site" matches the key "site", "`/site`" is put into the output, and execution continues at state 1. All steps of XPath continue through states 1, 12, and 13, with output:

`/site/people/person/name`

Finally, it is accepted at state 14. Since 14 is not a "`//*`" accept state, "`text()`" is attached. The final safe query generated from QFilter is:

`/site/people/person/name/text()`.

Algorithm 8: QFilterState.filter

Input: XPath query Q (with pointer to current step);
its filtered part: prefix
Output: String array of filtered query branches: $arrayQ'$

```
1  if Q.EOS() then
2      if is accept state then
3          return prefix;
4      else
5          return NULL;
6  if Q is double slash then
7      arrayQ' ⟵ QFilterState.DSFilter();
8      return arrayQ';
9  if ε − transitionChildState!=NULL then
10     Q' ⟵ ε − transitionChildState.filter(Q);
11     if Q' != NULL then
12         arrayQ'.insert(Q');
13 Token ⟵ Q.elementName;
14 Predicate ⟵ Q.predicate;
15 Q.nextStep() ;
16 if current state is a DSState then
17     prefix' ⟵ prefix+"/"+Token+Predicate;
18     Q' ⟵ filter(Q, prefix');
19     if Q' != NULL then
20         arrayQ'.insert(Q');
21 foreach match between Token and (key[i] in stateTransitionTable) do
22     if key[i]="*" then
23         Token' ⟵ Token
24     else
25         Token' ⟵ key[i];
26     predicateTable ⟵ stateTransitionTable.get(key[i]);
27     foreach predicate[i] in predicateTable do
28         prefix' ⟵ prefix + "/" + Token' + Predicate + predicate[i];
29         nextState ⟵ predicateTable.get(predicate[i]);
30         Q' ⟵ nextState.filter(Q, prefix');
31         if Q' != NULL then
32             arrayQ'.insert(Q');
```

Handling Predicates

Predicates such as "`[b=10]`" in "`//a/[b=10]/c`" frequently occur in Q or ACR. When Q has predicates in it, they are kept intact initially. Whenever the element name of an XPath step is accepted or re-written, then, the predicate (if any) is then attached to it. Otherwise, if a path token is rejected, the predicate is also rejected. For predicates from access control rules, from Figure 5.4(a), we can see that each element name (token) is mapped to a table, holding all the predicates affixed with it. During the execution of QFilter, when the XPath step of the incoming query matches an element name in the state transition table, we further process their corresponding predicates: (1) Rule predicate in the predicate table is attached to the output of the current step; (2) Multiple entries in the predicate table are not exclusive. That is, QFilter's execution is split into different branches, and each takes a different entry in the predicate table; and (3) For each branch, QFilter's execution continues at the next state, and maps to the predicate in the predicate table.

Example 5.4. Let us replace R5 of Example 2.1 by

```
/site/regions/*/item[quantity>0]/location
```

Then, the QFilter of Figure 5.3 (Middle) is re-constructed to Figure 5.5. Each none-leaf none-ε state carries an empty predicate processing state "φ", but omitted for simplicity.

Example 5.5. Suppose we process a query:

```
/site/regions/namerica/item/location
```

using the QFilter of Example 5.4. The query goes through states 0, 1, 5, and 6, with an output:

`/site/regions/namerica/item`[1]

at state 6. Then, QFilter execution splits into two branches at state 6p. Branch 1 goes to state 7.1 with output:

[1]Note, at state 5, "*" in the transition table matches with the current element name "namerica", and their intersection, "namerica", is kept in the output.

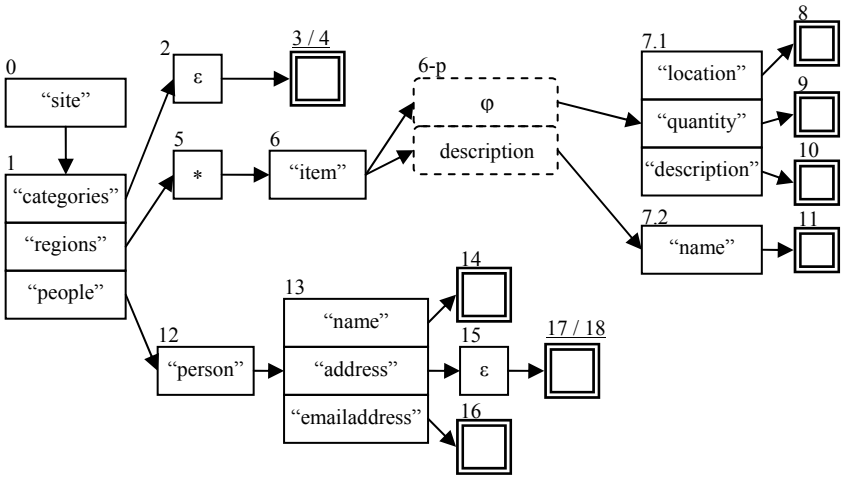


Figure 5.5. QFilter with predicate processing states.

```
/site/regions/namerica/item
```

(φ predicate is omitted); branch 2 goes to state 7.2 with the output:

```
/site/regions/namerica/item[quantity>0]
```

(predicate from rule `R5'` is attached). Branch 1 is rejected at state 7.1 while branch 2 continues through state 7.2 and is finally accepted at state 11. At the end, this query is returned as the output:

```
/site/regions/namerica/item[description]/name/text()
```

Handling "//" in queries

The fragment"`//x`" asks for element "`x`" with any path(s) precede it. The `//x` or `//*` step in Q matches the key `x` (or any key) in the current NFA state, or any of its descendant state. Therefore, either "`//x`" or "`//*`" in Q triggers the state transition from the current state to all of its subsequent states, then tries to match "`x`" or "`*`" with keys in their state transition table. In this case, the query Q is split into branches that continue at each of the subsequent states of the current state (where the "`//`" input is detected). Such a query needs to be rewritten. In general, we rewrite "//" with the path from the current state to the destination state, where the branch continues to be executed. Then, each branch of the QFilter

Start	Destination	Re-written Query	Start	Destination	Re-written Query
12	12		12	15	`/person/address`
	13	`/person`		16	`/person/emailaddress`
	14	`/person/name`		17/18	`/person/address`

Table 5.1. "//" transition look-up table.

execution restarts by matching the input element name ("`x`" or "`*`") with keys in the state transition table.

Example 5.6. Let us use the aforementioned QFilter to process the query:

```
/site/people//name
```

The first two steps "`/site/people`" trigger the state transition from $0 \rightarrow 1 \rightarrow 12$. Then, when it encounters the "//", Q breaks into the following six branches, each has the input element name "`name`":

1. `/site/people` restarts at state 12;
2. `/site/people/person` restarts at state 13;
3. `/site/people/person/name` restarts at state 14;
4. `/site/people/person/address` restarts at state 15;
5. `/site/people/person/emailaddress` restarts at state 16;
6. `/site/people/`
   `person/address/` restarts at state 17/18.

Obviously, only the state 13 (branch 2) and 17/18 (branch 6) can accept the input token `name`. Thus the final output is:

```
/site/people/person/name | /site/people/person/address//name
```

To speed up the traversal, we can build a look-up table for each state. It is an index to all the sub-states, together with the replacing string. As an example, the look-up table of state 12 is shown in Table 5.1.

5.3 Analysis

5.3.1 Computational Complexity

Computational cost of QFilter includes time for bothQFilter construction and execution. For QFilter construction, the complexity is $O(N)$, where N is the number of steps of XPath expression in ACR. For QFilter execution, (1) When there is no wildcard in Q, filtering Q costs $O(M)$, where M is the number of steps of Q. The worst case occurs when Q is accepted or rewritten; (2) When the wildcard "*" exists in Q, filtering costs $O(|NFA|)$. The worst case occurs when Q is "/*/*.../*", since it requires the traversing of the entire NFA; (3) For Q with "//" step, the cost becomes $O(M * n_1 * n_2 * ... * n_k)$, where k is the number of wildcards "//" in Q and n_i is the size of the child QFilter at the state which first meets the i^{th} "//" path. Note that this is an acceptable cost since the worst case query of "//*//*...//*", is rather rare in real-world XML queries. Overall, QFilter is computationally practical since the worst case of filtering rarely occurs. Furthermore, we experimentally validate this claim in the experimentation.

5.3.2 Security

Next, we prove that using QFilter is safe so that no non-accessible data are to be returned to unauthorized authors.

Theorem 5.1. *The* QFilter *execution algorithm of Algorithm 8 always generates secure answers when Q and object parts of ACR are limited to XPath expressions, and ACR contains only positive rules or negative rules.*

Proof. Suppose an NFA M is created by the QFilter, and $M(Q)$ refers to the evaluation of a query Q against M. Then, Theorem 5.1 can be represented as: $Q' = M(Q) = Q \overset{D}{\cap} ACR$, which can be decomposed into two sub-theorems, $Q' \subseteq Q \cap ACR$ and $Q' \supseteq Q \cap ACR$. For security concerns, people are more interested in the first sub-theorem, $Q' \subseteq Q \cap ACR$, which ensures the output query Q' is secure. We provide its sketch proof here. The second sub-theorem can be proved similarly. To prove $Q' \subseteq Q \cap ACR$, we need to show that (1) $Q' \subseteq Q$ and (2) $Q' \subseteq ACR$.

1. $Q' \subseteq Q$: For the accepted queries, the output of QFilter is the original query itself: $M(Q) = Q$. For the rejected queries, the output of M is an empty string or error message, where we also have $M(Q) = \varphi \subseteq Q$. For the rewritten queries, since the rewriting algorithm only changes wildcards into more specified path strings, it is obvious that $M(Q) \subseteq Q$.

2. $Q' \subseteq ACR$: From the NFA theory, we can see that each accept state accepts XPath expressions according to the particular rule. In this way, each accepted branch of input query Q is accordance with the specified rule ($Q_i \subseteq R_j$). Therefore, Q', as the union of these branches, is also accordance with the union of all the rules: $Q' \subseteq ACR$.

□

5.4 Experimental Validation

To use QFilter for XML access control, we first need to construct QFilter based on access control rules. Then, input queries are processed by QFilter to generate safe queries; and safe queries are finally sent to underlying XML engine to retrieve XML data. According to this process, we test QFilter in the following aspects: (1) QFilter initialization, (2) QFilter execution, and (3) the evaluation of filtered query.

The major concerns of performance are computational speed and storage cost. For storage issue, since QFilter is basically an extended NFA, its total size in memory is $(size\ of\ state) \times (number\ of\ states)$, which costs $O(N)$ with N denoting the number of states. The size of a state highly depends on the implementation, but is relatively small due to the simplicity of a state (the main data structure is the state transition hash table). The number of states is also limited since the number of rules is limited and path sharing exists. Therefore, it is trivial to store QFilter created from thousands of rules in main memory. In this section, therefore, we focus on the evaluation of the speed of QFilter from the aforementioned three aspects. In (1), we test the impact of complexity and the number of rules to QFilter construction speed. In (2) and (3), the complexity and number of rules along with the impact of complexity of Q are studied.

Comparison is another important issue. Due to the scarcity of comparable

approaches to our QFilter, to our best knowledge, the static analysis method of [52] is the only pre-processing based method similar to ours. It conducts pre-processing "security check" only, thus we only compare its initialization and execution with QFilter. We still want to emphasize that QFilter generates safe queries thus does not need access control functions from underlying XML engines, while static analysis approach still require security support from XML engine. For (3), we compare the end-to-end query processing time between primitive, QFilter, and post-processing approaches. We also provide the processing time for original (unsafe) queries for reference.

5.4.1 Set-Up

We used the well-known XMark schema [64] and its XML document generator to generate test XML documents. Since the size of test data was not a major factor to determine the performance of various methods, here we present only the case of test set with 5 MB. As an underlying XML database, we used Galax 0.3.1 [67] that can evaluate XQuery (and thus XPath) efficiently. Pre-processing approach, QFilter, was implemented in Java (JDK 1.4.2) and communicated with Galax through its Java-API. For post-processing approach, we used the YFilter [21] from UC Berkeley as an implementation of AFilter.

The types of queries and number of access control rules are important in our experimentation, and thus carefully selected and measured. For XPath expressions in Q and ACR, both user-defined (denoted as UD) as well as synthetic (denoted as SN) tests were used. That is, we have four test cases by combining two factors in two dimensions; UD-Q/UD-ACR, UD-Q/SN-ACR, SN-Q/UD-ACR, and SN-Q/SN-ACR. Note the combination of user-defined queries over synthetic rules does not really make sense. So we only test other combinations. All synthetic XPath expressions were generated by YFilter package. The Costumer Advertisement Manager (CAM) role is created as a user-defined role extended from the example in Section 5, and shown in Table 5.2. CAM is in charge of delivering advertisements to costumers, thus is permitted to access users information except for their credit card, profile, and item's basic information. This policy can be captured by the rules shown in Table 5.2 (all rules with RC type are already converted to equivalent ones

No.	Rule
1	(CAM, "/site/regions/*/item[@quantity>0]/location", +, LC)
2	(CAM, "/site/regions/*/item[@quantity>0]/quantity", +, LC)
3	(CAM, "/site/regions/*/item[@quantity>0]/name", +, LC)
4	(CAM, "/site/regions/*/item[@quantity>0]/description", +, LC)
5	(CAM, "/site/categories", +, LC)
6	(CAM, "/site/categories//*", +, LC)
7	(CAM, "/site/people/person/*", +, LC)
8	(CAM, "/site/people/person/creditcard", −, LC)
9	(CAM, "/site/people/person/profile", −, LC)

Table 5.2. User-defined ACR: CAM case.

QS	*	//	P	QS	*	//	P
QS1	0	0	0	QS2	0	0	2
QS3	0	10%	0	QS4	10%	0	0
QS5	10%	10%	0	QS6	10%	10%	2
QS7	0	20%	0	QS8	20%	0	0
QS9	20%	20%	0	QS10	20%	20%	4

Table 5.3. Synthetically-generated 10 user query sets (QS1 – QS10) with different probabilities of "*" and "//" at each XPath step and number of predicates.

with LC type [52]).

In order to show the impact of predicates in ACR, we test both rules with and without predicates. Hereafter, we use "user-defined rules with predicate" to indicate above nine rules. On the other hand, we use "rules without predicates" indicates the remaining rules after "[@quantity>0]" fragment is removed from them. User-defined queries are mainly used to validate the correctness of QFilter. In addition, we also created queries with the synthetically generated XPath expressions as shown in Table 5.3 to evaluate the scalability. 500 queries are created for each query set of Table 5.3.

We could not test value-based predicates. Because the query (XPath) generator we use does not know the real values of attributes in our XML documents, it produces XPath expressions based on schema only. The randomly generated attribute values do not match with the values in document, e.g. XPath generator produces predicates "person[@id=0]", while we have "<person id='person0'>" in the XML document. They will cause errors in the evaluation process. On the other hand, value-based predicates are processed in the same way as path-based predicates in QFilter. Thus testing with path-based predicates gives a accurate reference of QFilter performance.

Figure 5.6. QFilter construction using one single use-defined rule.

5.4.2 Evaluating QFilter Construction

In real world applications, QFilter is likely to be constructed offline. Once the service starts, we do not need to modify or reconstruct QFilter unless ACR is changed. Thus, the speed of QFilter construction is of less importance to users. Nevertheless, experiments show that QFilter construction is fast enough, even to be done online.

We first construct QFilter with user-defined rules (nine rules for the role CAM, as defined above) and record the construction time. We construct QFilter using each of the rules with and without predicates and compare the speed. According to Figure 6, QFilter construction time for different rules mainly depends on the complexity of the XPath expression, i.e., number of QFilter states to be built. QFilter construction is faster for shorter and simpler rules, since less parsing time is spent and less states are created. We also see that predicates bring more overhead to QFilter construction, since an additional predicate processing state is created.

Note, in real world applications, QFilters are not created for each individual rule. Rather, one QFilter is created for all the + rules and another QFilter for all the - rules. For CAM role, one QFilter for all the "+" rules is constructed in 1155 μs, and one QFilter for all the "-" rules is constructed in 496 μs.

Next, we construct QFilter with synthetic rules and record the construction time. In each experiment, we generate 10, 50, 100, 200, 300, 400, 500, 600, 700,

	RS	*	//	P	RS	*	//	P
(a)	RS1.1	0	0	0	RS1.2	10%	0	0
(a)	RS1.3	20%	0	0	RS1.4	30%	0	0
(b)	RS2.1	0	0	0	RS2.2	0	10%	0
(b)	RS2.3	0	20%	0	RS2.4	0	30%	0
(c)	RS3.1	0	0	0	RS3.2	0	0	1
(c)	RS3.3	0	0	2	RS3.4	0	0	3

Table 5.4. Synthetically-generated ACR with different probabilities of "*" and "//" at each XPath step and number of predicates.: (a) impact of * path; (b) impact of // path; and (c) impact of predicates.

800, 900 and 1000 rules (distinct rules) respectively, each with the maximum length of 10 path elements. We use uniform distribution in selecting child elements. Different groups of synthetic rules are defined in Table 5.4.

Figure 5.7 shows that QFilter construction is fast and scalable. * or // paths do not slow down the construction. On the contrary, when we set higher * or // probability in rules, QFilter construction becomes faster. There are two reasons for this: (1) since XPath string parsing takes much of the QFilter construction time, the existence of * and // in the path makes the string shorter: as one path step, * is shorter than a string value path name; moreover the XPath generator we used tends to generate shorter XPath expressions (with less steps) upon existence of //; and (2) in QFilter implementation, * and // paths are processed separately (not in the state transition hash table), thus we do not search or insert the state transition hash table, which makes it faster.

For predicate, it seems that QFilter construction is faster with rules than with predicates. This is because predicate states are constructed faster than regular NFA states. For XPath strings of similar length, those with predicates are processed faster. But, many predicates (e.g., 6 predicates in rule set 4) may increase the total length of XPath strings, and thus slow down QFilter construction.

5.4.3 Evaluating QFilter Execution

After QFilter is created with ACR, we use it to filter the input query Q to yield safe query Q'. Using the CAM role, we first test how the properties of user query Q affect the filtering speed. That is, we prepare ten different query categories (as

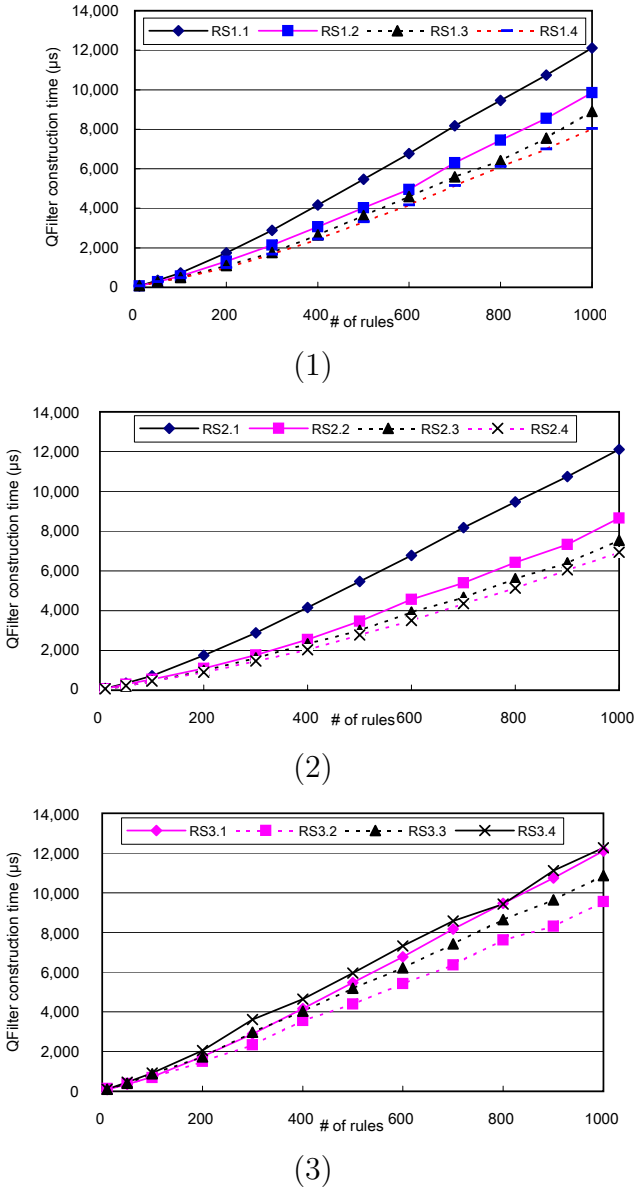


Figure 5.7. QFilter construction using synthetic rules. From left to right: impact of (1) * path; (2) // path; (3) predicates

shown in Section 5.4.1) and for each category, we generated 500 synthetic queries based on the XMark DTD. Using these random XPath expressions as input to QFilter, we measure the number of accepted, denied, or rewritten queries in each group. We also separate a category "*minus*" to indicate the queries that are rewritten by negative rules. Then we measure the average QFilter execution time for each group and for each output type (*accept*, *deny* and *rewrite*). The results

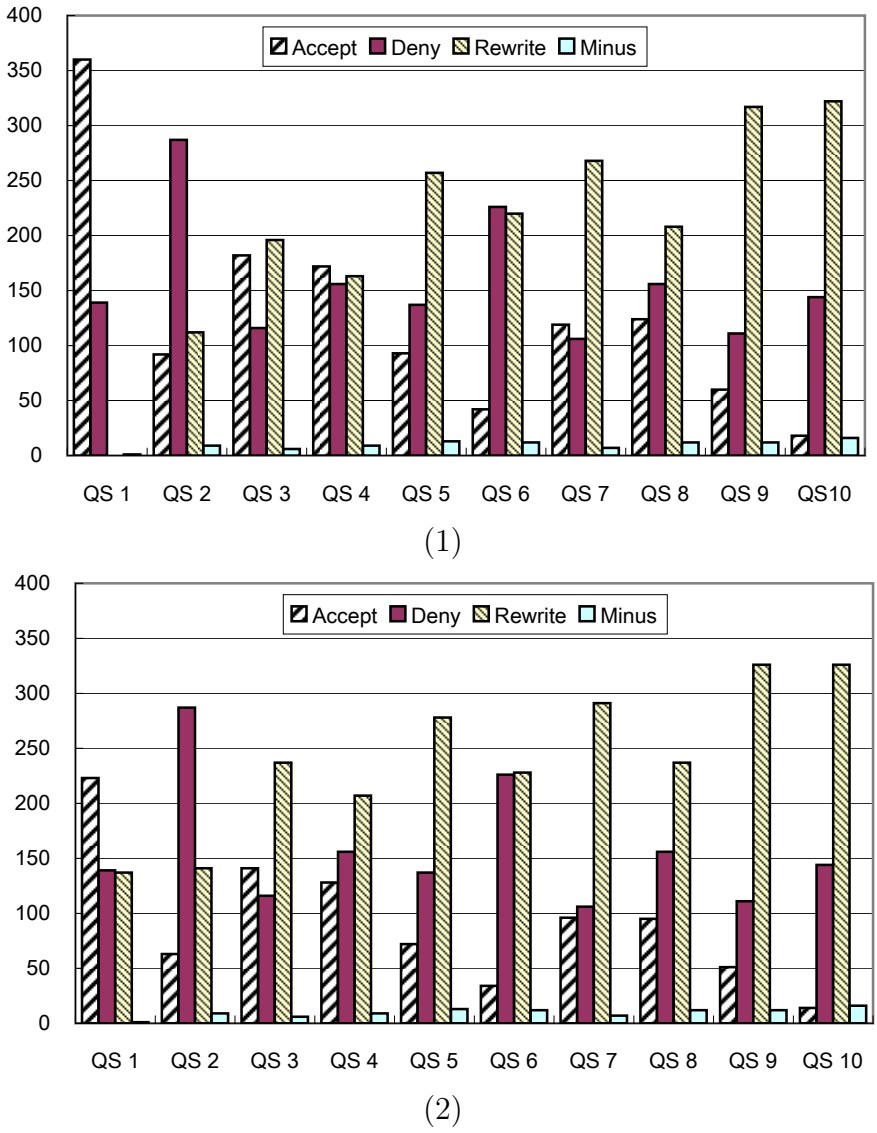
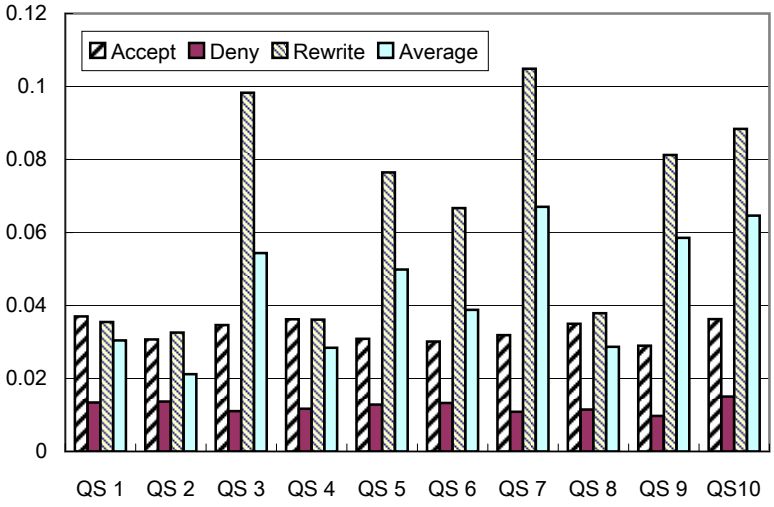


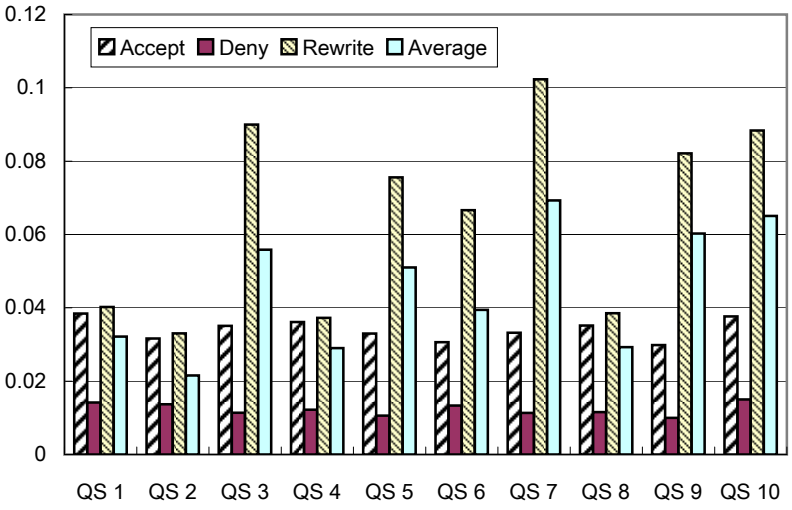
Figure 5.8. QFilter output: number of accepted, denied, rewritten, and minus rewritten queries. (1) rules without predicate; (2) rules with predicate(s)

are shown in Figures 5.8 and 5.9.

From Figure 5.8, we can summarize: (1) for rules without any predicate(left), queries in set 1 (no "*", no "//") are either rejected or accepted, since there are no wildcards to be rewritten; for rules with predicates, they may be rewritten: predicates can be inserted. (2) queries with higher probability of wildcards "*" and "//" are more likely to be rewritten; (3) fewer queries in set 6 and 10 are rewritten than sets 5 and 9: existence of predicates in queries causes less regular



(1)

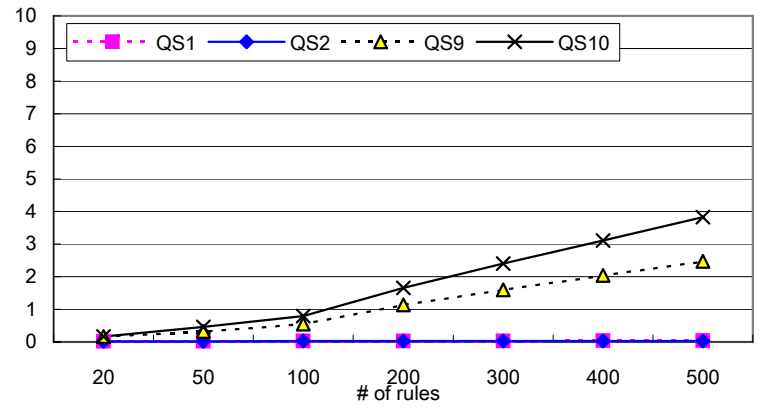


(2)

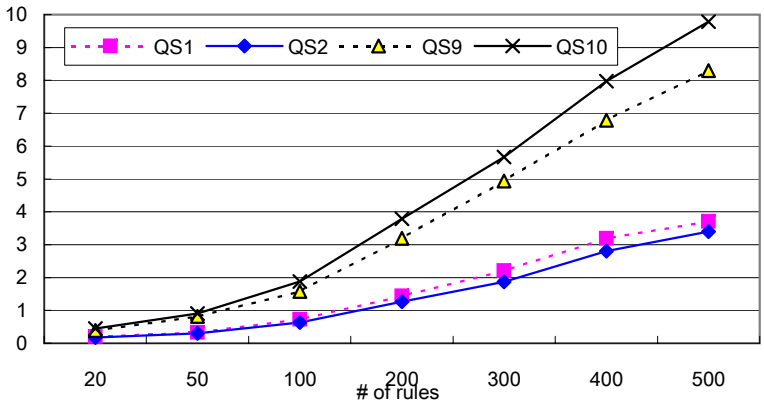
Figure 5.9. QFilter execution time (ms) for three types of outputs and their average. (1): rules without predicate; (2): rules with predicate(s)

path steps in each query, thus these queries generally have a lower probability of "*" and "//"; (4) Comparing two figures, we can see that emergence of predicates in rules do not affect denied queries, but some originally accepted queries are rewritten (predicates are inserted).

Here, let us explain more about (2). Queries in sets 3 to 10 are generated with 10% or 20% probability of having "*" or "//" at each step. However, the probability does not automatically indicate that generated queries should have one



(1)



(2)

Figure 5.10. QFilter execution time (ms) for synthetic rules and synthetic queries. (1) rule set 1, (2) rule set 2

or more "*" or "//" steps. When we manually look into the generated queries and the QFilter results, we found that some of these queries do not have any "*" or "//" steps, and most of them are either accepted or denied. From Figure 5.9, we can summarize the following: (1) QFilter is generally faster in accepting and denying queries, but *slower* to rewrite queries with wildcards, especially with "//" paths. This is because QFilter needs to traverse more states to process "*" and "//"; and (2) Predicates in rules does not bring much overhead to QFilter execution. Average processing time is quite similar, and query rewriting time is even reduced, since some of the originally accepted queries are just rewritten at predicate state, which is faster than "*" amd "//" rewritten.

Next, we test how QFilter execution performance degrades as the number of

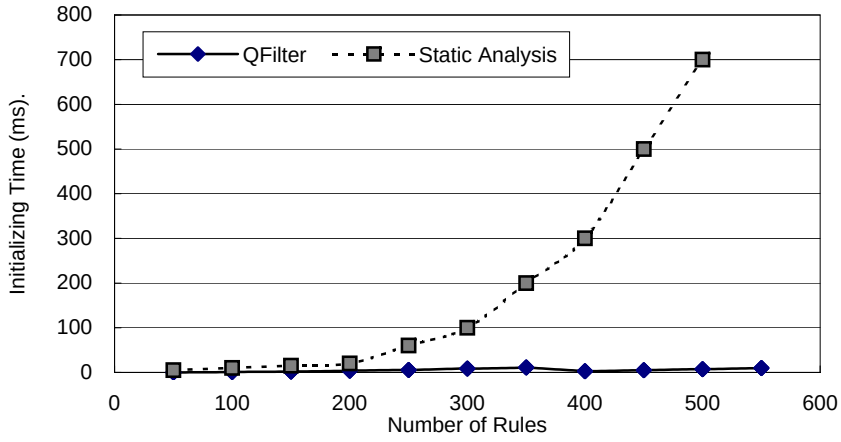
rules in ACR increases. We constructed a QFilter using 20 to 500 synthetic rules based on XMark DTD (SN-ACR) and tested with random queries (SN-Q). We create two sets of rules as follows: `Rule Set 1`contains rules with no * path, no // path, and no predicates, and `Rule Set 2` contains rules with 10% * probability, 10% // probability, and 2 path-based predicate. On the other hand, we pick query sets 1, 2, 9 and 10, then process them using QFilter with the above rules. Figure 5.10 shows the average QFilter execution time for each rule set By and large, as the number of rules in ACR increases, the QFilter execution time to filter out conflicting parts from Q increases too. This is understandable since there are more branches to test in QFilter. However, note that the longest time it took to rewrite Q, when QFilter has 500 synthetic rules, was still within only 10 millisecond.

5.4.4 Comparing QFilter vs. Static Analysis

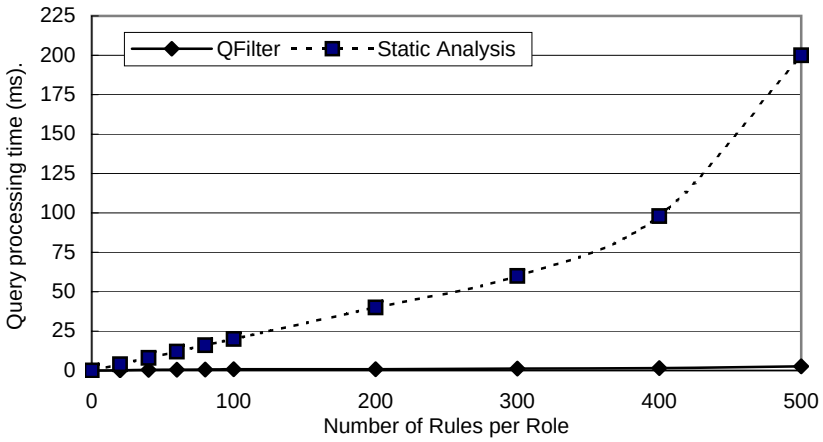
The static analysis method of [52] is the only pre-processing based method like ours. It can handle only two cases of queries vs. ACR; i.e., access fully granted ($Q \subseteq R$) or access fully denied ($Q \cap R = \varphi$), where Q is an input query and R is $ACR+$. However, our QFilter method is able to process all three cases; i.e., $Q \subseteq R$, $Q \cap R = \varphi$, and partial overlap ($Q \nsubseteq R \wedge Q \cap R \nsubseteq \varphi$). Therefore, QFilter method can run on any XML databases whether or not they have security support, which is not possible for [52]. Since the end-to-end processing time (i.e., from the moment a query is submitted to the time "safe answers" are returned) of [52] was not available to us, we only compared time to construct and to check security policies between QFilter and static analysis methods.

To directly compare QFilter against [52], we generate synthetic rules according to the XML specification DTD (xmlspec-v21.dtd). In this experiment, "+" and "-" rules are 50% each. Figure 5.11(left) shows the results. Comparing with [52], QFilter construction appears to be "flat", taking only 7 milliseconds for 550 rules. However, the initialization of Static Analysis [52] is more sensitive to the number of rules, so that the graph increases more sharply. Note the initialization time of static analysis mechanism was estimated from Figure 7 of [52] where it supports predicates using upper-bound and low-bound constrainers.

Next, QFilter's execution time is compared to that of [52], which essentially



(1)



(2)

Figure 5.11. Performance comparison of QFilter and Static Analysis [52]) approach. (1) initialization speed; (2) security check speed

spend substantial time to check the containment of two automata. The result is shown in Figure 5.11 (right). When 500 synthetic rules are used, QFilter is faster than the static analysis method of [52] up to 200 times. In additions, we are able to handle the partial overlap case: $Q \nsubseteq R \wedge Q \cap R \nsubseteq \varphi$, which [52] requires the underlying database engine to process.

5.4.5 End-to-end Query Processing

Finally, we compare the end-to-end processing time among four approaches of Figure 5.13: (1) No security check is made (thus final data is un-safe); (2) Primitive

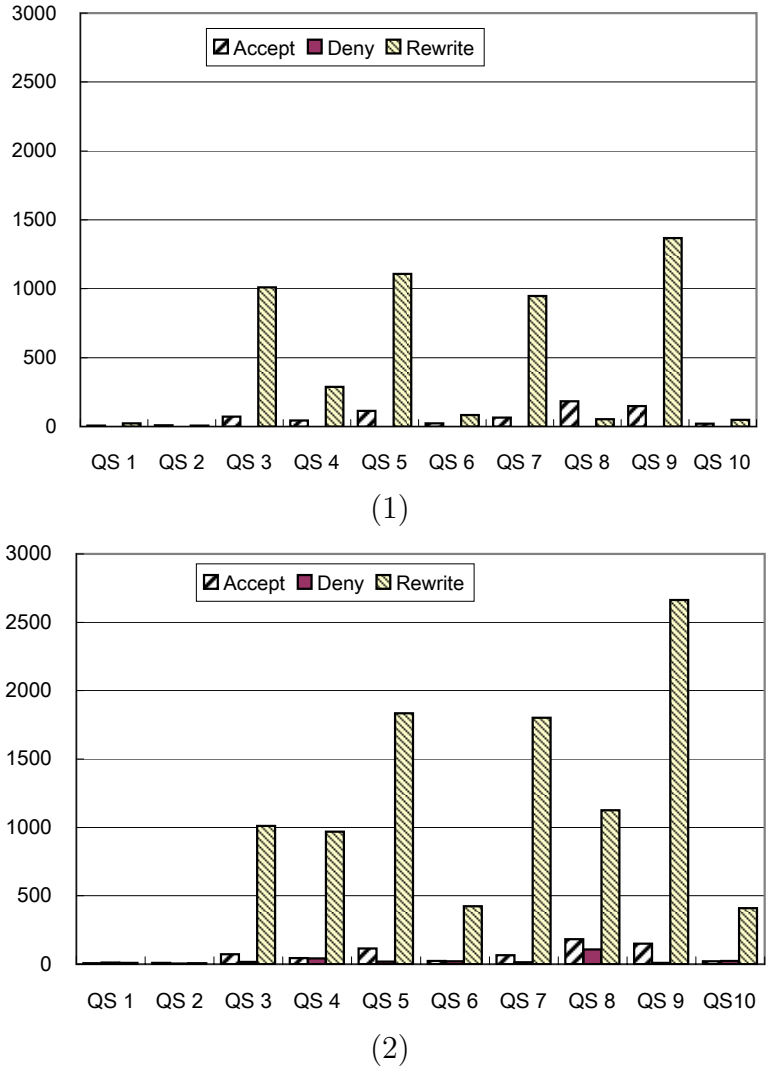


Figure 5.12. End-to-end query processing time for QFilter approach: (1) query processing with QFilter; (2) query processing without QFilter.

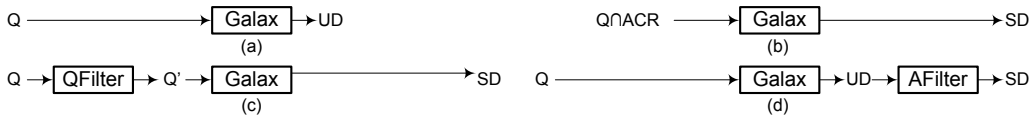


Figure 5.13. Four XML access control approaches for comparison.

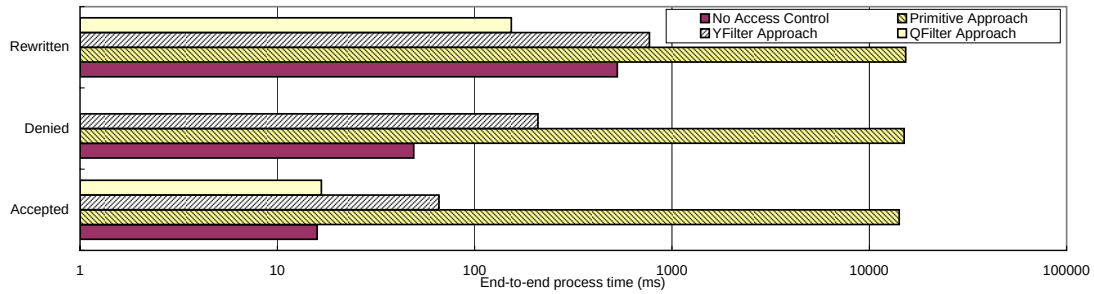


Figure 5.14. End-to-end query processing time comparison of all approaches, in logarithmic scale.

approach; (3) AFilter (post- processing); and (4) QFilter (pre-processing). In this experiment, due to the high computation of primitive approach, we used a smaller size (1.5MB) XML document, with the same format as described in section 5.2. End-to-end query processing time denotes the total time needed to process Q: from receipt of query until answer is returned. In Figure 5.13, (a) indicates the query processing without any security check, where the output document UD is un-safe, the end-to-end time is mainly evaluation time of Q; (b) indicates the primitive approach, which generates the safe result D, the end-to-end time is mainly the intersect query ($Q \cap ACR$) evaluation time; (c) shows the QFilter approach, where the end-to-end time includes the QFilter execution time and filtered query (Q') evaluation time; and (d) indicates the post-processing approach, where the end-to-end time includes the original query evaluation time and un-safe answer, UD, filtering time. Note that we do not count the I/O time of the query input and the answer output. Note that for (d), since XML engines return only queried node without their ancestor tags, we manually wrote an external script to recover ancestor tags when UD is generated. But to be fair, that extra time for running script was not counted in. However, it is worthwhile to point out that if one uses the recursive function of XQuery to implement this in XML databases, the cost would have been even higher. Thus, what we report here for post-processing approach is a significant "under-estimate".

Figure 5.14 summarizes the comparison of the four approaches. QFilter-based pre-processing approach is a clear winner regardless of the query categories, and thus a promising solution for XML access controls; it significantly outperforms the primitive approach and an (un-safe) query processing which does not enforce

XML access control. Interesting phenomenon is that QFilter even outperforms no security check case even for the queries re-written. This implies that when Q is filtered to Q' by QFilter, as a side-effect, Q' was optimized so that Q' is processed more efficiently than Q. That is, when Q' is processed by Galax, since its query constraints have been tightened by additional conditions added by QFilter, it is typically much faster than the original query, while ensuring returning only safe answers.

Since the post-processing approach requires a data filtering stage after Q is evaluated, thus it is surely slower than the original query processing and much slower than QFilter approach. In many cases, QFilter can quickly determine whether the query is fully "Accepted" or "Denied" where the query filtering time is negligible compared to potential save from unnecessary query evaluation time.

Chapter 6

RDBMS-supported XML Database Systems

6.1 RDBMS-supported XML Database Systems

Before we go to access control issues, let us recall RDBMS-backed XML database systems (XRDB) introduced in Chapter 2.

As illustrated in Figure 6.1, in an XRDB system: XML documents (D_X) are converted into relations (D_R) using some conversion algorithm ($C()$). D_R is then managed in RDBMS. Any RDBMS that handles SQL could be used here. User issues XML query Q_X (XPath or XQuery) using published XML schema; Q_X is

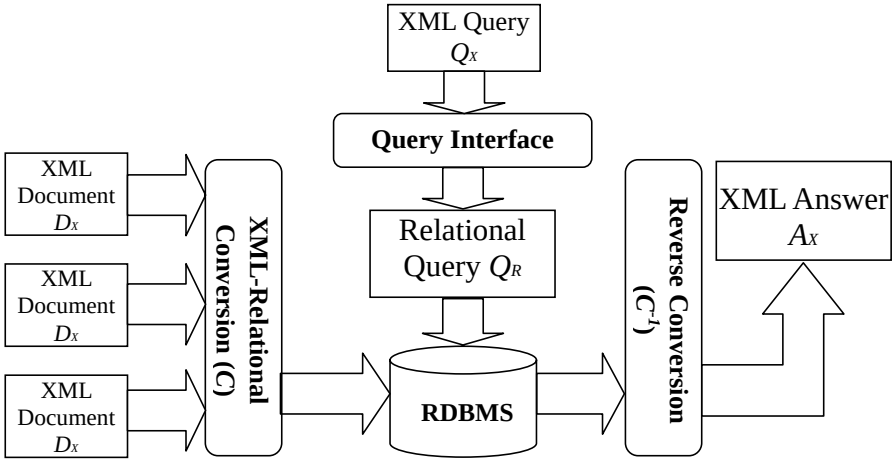


Figure 6.1. Overview of XRDB system architecture.

then converted into Q_R (in SQL) and evaluated against D_R. Relational answer A_R is finally converted back to XML answer A_X and returned to user.

XML to relational conversion algorithms are surveyed in Chapter 2. Here we give an example to show how Shared-inlining [66] and XRel [72] convert XML document into relations.

We use the XMark DTD, in which a "`person`" node consists of attribute "PersonID", element children "`name`", "`address`", "`creditcard`", "`emailaddress`", etc. Especially, "`address`" node has element children "`city`", "`country`" etc., as shown in Figure 6.2.

Shared-Inlining [66] is a schema-based XML to relational conversion approach, which shreds XML tree into two dimensional tables. In this example, `<people>` node is translated into a `people` table. Each row in the table represents a person, and each descendant node is converted into one column. Part of the relational schema is as follows (for the full relational schema converted from XMark XML schema, see [42]):

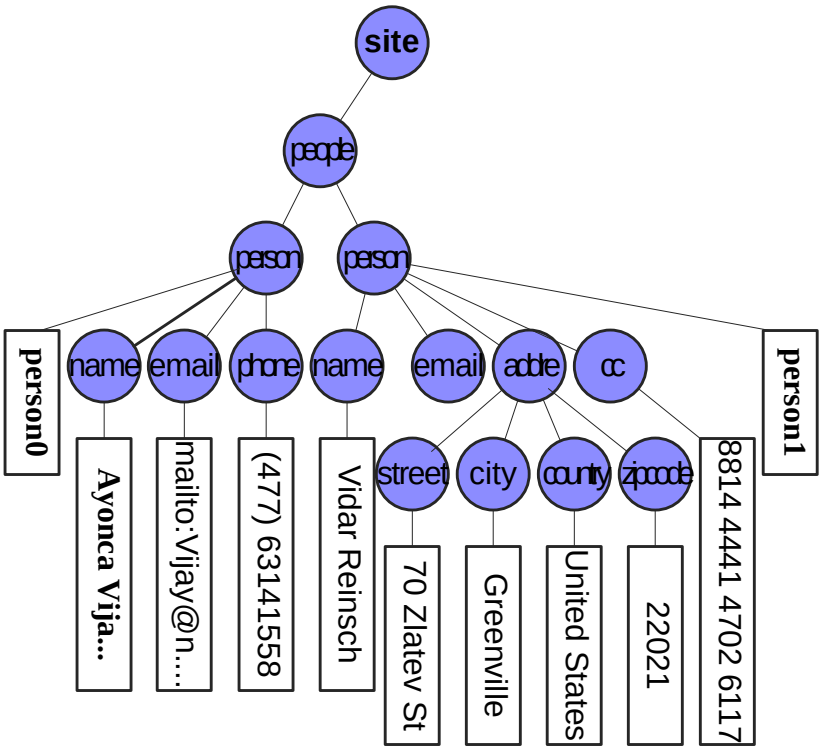


Figure 6.2. Part of the XMark tree.

```
Person(PersonId, ParentId, Person, Person_address,
       Person_address_city, Person_address_country,
       Person_address_province, Person_address_street,
       Person_address_zipcode, Person_creditcard, ......)
```

Figure 6.3 shows part of the table. We can see that three `<person>` nodes in the original XMark XML document is converted into three rows. Each text element is converted into one cell.

XRel [72] is a non-schema XML to relational conversion approach. It decomposes XML documents into document, element, attribute, text, and path tables. The structure of the element and path table are: `element(docID, elementID, parentID, depth, pathID, st, ed, idx, reidx)` and `pth(pathID,pathexp)`, respectively. In this approach, each node is stored as one record in the `element` table, and each distinct path is stored as one record in the `pth` table. It uses `pth.pathexp` to keep the path expressions (similar to XPath), and `element.st` and `element.ed` to mark the start and end offset of each node in the document. As a simple example, we decompose an XMark document using XRel and show part of `element` and `path` tables in Figure 6.4. As we can see, element 252 is a node of path 164 ("`/site/people`", as in the `path` table); which starts from offset 33996 (byte) and ends at 36229 in the original XML document.

6.2 XML to Relational Conversion: the Model

We model the X2R conversion algorithm (surveyed in Chapter 2) as follows:

Remark 3. A relational to XML conversion method contains: (1) $C_D()$ to convert XML to relational data, (2) $C_Q()$ to convert XML query (XQuery or XPath) to

Person	Person _address _street	Person _address _city	Person _address _country	Person _address _zipcode	Person_name	Person_eamil	Person _credicard	…
Person0					Ayonca Vijay	mailto:Vijay@nyu.edu		…
Person1	70 Zlatev St	Greenville	United States	22021	Vidar Reinsch	mailto:Reinsch@usa.net	8814 4441 4702 6117	…
Person2	49 Nandavar St	Tallahassee	United States	10034	Rens Rifaut	mailto:Rifaut@duke.edu	4464 2718 4111 2553	…

Figure 6.3. Part of the `people` table of shared-inling approach.

SQL, and (3) C^{-1} to convert relational answer back to XML.

That is, $Q_R = C_Q(Q_X)$, $D_R = C_D(D_X)$, and $A_X = C^{-1}(A_R)$. From this, the process of "evaluating XML query on XRDB" can be modeled as the following Equation:

$$A_X = C^{-1}(A_R) = C^{-1}(Q_R\langle D_R\rangle) = C^{-1}(C_Q(Q_X)\langle C_D(D_X)\rangle) \tag{6.1}$$

Remark 4. An X2R conversion algorithm is *lossless* iff: (1) (lossless node conversion) $\forall$ XML node $x_i, C_D^{-1}(C_D(x_i)) = x_i$; (2) (lossless node set decomposition) $\forall$ XML node set $\{x_1, ..., x_n\}, C_D^{-1}(C_D(\{x_1, ...x_n\})) = C_D^{-1}(\{C_D(x_1), ...C_D(x_n)\}) = \{C_D^{-1}(C_D(x_1)), ...C_D^{-1}(C_D(x_n))\}$; and (3) (exclusive conversion) $C_D(x_1) = C_D(x_2)$ only when $x_1 = x_2$, and $C_D^{-1}(r_1) = C_D^{-1}(r_2)$ only when $r_1 = r_2$.

Remark 5. An X2R conversion algorithm is *correct* iff: $\forall$ query Q and $\forall$ document X, $Q\langle X\rangle = C^{-1}(Q_R\langle D_R\rangle) = C^{-1}(C_Q(Q_X)\langle C_D(X)\rangle)$.

Definition 6.1 (Soundness)**.** An X2R conversion algorithm A is **sound** iff it is *lossless* and *correct*.

In the remainder of the dissertation, we assume that the conversion algorithm being used is *sound*. Finally, we ignore the order of XML nodes when we com-

pathID	pathexp
0	#/site
1	#/site#/categories
.....	
164	#/site#/people
165	#/site#/people#/person
.....	
188	#/site#/people#/person#/creditcard

docID	elementID	parentID	depth	pathID	st	ed	...
0	0		1	0	0	563258	...
.....							
0	252	0	2	164	33996	36229	...
0	293	252	3	165	35592	35826	...
0	299	252	3	165	35832	36217	...
0	303	299	4	188	35989	36032	...
.....							

Figure 6.4. Part of the `path` table and `element` table of the XRel approach.

pare the correctness, since this feature is not supported in most X2R conversion algorithms.

In the research community, most X2R conversion algorithms only support a subset of XQuery/XPath. For instance, many of them support parent-child (/), ancestor-descendant (//), wildcard (*) and predicates. Later, we will show that our approach does not alter the query or data conversion algorithm. Therefore, the query conversion totally depend on the X2R conversion algorithm; i.e. for a particular X2R conversion method X, our algorithm supports everything that X supports. For ease of understanding, we do not use predicates in the examples, however, we test queries with predicates in our experiments.

6.3 XML Access Control in XRDB: the Problem

The goal of XML access control is in a sense to ensure that only *safe answer* (SA) is returned to users. As shown in [44] and [46], safe answer of a query Q includes all the XML nodes n such that: (1) n is part of $\langle Q \rangle$, (2) the access of n is granted by positive rules, and (3) the access of n is *not* denied by negative rules. That is, the precise semantics of "safe XML answer," SA_X, can be modeled as:

$$SA_X = \langle Q_X \rangle \overset{D}{\cap}_X (\langle ACR^+ \rangle \overset{D}{-}_X \langle ACR^- \rangle) \quad (6.2)$$

$$= \langle Q_X \rangle \overset{D}{\cap}_X [(\langle R^+_{X_1} \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^+_{X_n} \rangle) \overset{D}{-}_X (\langle R^-_{X_1} \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^-_{X_m} \rangle)] \quad (6.3)$$

Equation 6.1 models how XML query is evaluated in XRDB to return XML answer, A_X. Similarly, Equation 6.2 models how only safe XML answers, SA_X, are returned. Therefore, we have:

Definition 6.2 (Secure XRDB)**.** An XRDB is called secure iff $\forall$ access control rule set ACR_X and $\forall$ query Q_X, it always returns the safe answer A_X:

$$A_X \equiv SA_X \quad (6.4)$$

$$\iff \quad C^{-1}(\{C_Q(Q_X)\langle C_D(D_X)\rangle\}') \quad (6.5)$$

$$\equiv \langle Q_X \rangle \overset{D}{\cap}_X [(\langle R^+_{X1} \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^+_{Xn} \rangle) \overset{D}{-}_X (\langle R^-_{X1} \rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^-_{Xm} \rangle)] \quad (6.6)$$

Note that $\{C_Q(Q_X)\langle C_D(D_X)\rangle\}'$ indicates that access control mechanism inter-

venes in relational query processing.

Our goal in the second part of this dissertation is to enforce XML access controls on RDBMS so that Equation 6.4 holds in *XRDB* setting. In this way, we need to develop *relational* access control rules and *relational* deep set operators that are "*equivalent*" to their corresponding *XML* access control rules and *XML* deep set operators. In next chapter, we first give formal definition of "*equivalent*", and then present our solutions based on the defined equivalency.

Chapter 7

XML Access Control in XRDB: A Theory

All entities of the 4-tuple XML access control model, except the *object* entity, can be directly adopted to relational access control model. Since the object entity is specified in XPath, we may apply an X2R algorithm $C(R_X.object)$ to get $R_R.object$. As a result, we can convert XML access control rules to "equivalent" relational access control rules:

$$\mathbf{R_R} = \{\mathbf{R_X.subject}, \mathbf{C(R_X.object)}, \mathbf{R_X.action}, \mathbf{R_X.sign}\}$$

However, the converted relational access control rules cannot be directly enforced in XRDB – naive enforcement of R_R may not generate correct answer, or even leads to security leakage, as demonstrated in the following example:

Example 7.1. Consider two rules of Figure 7.1(a) with XRDB(XRel) – that is, XML data are stored in RDBMS using XRel [72] conversion algorithm. The "element" table is partly shown in Figure 7.1(b). Rule 1 indicates that a user is allowed to access `<person>` nodes, i.e., nodes 293 and 299 (second and third record in Figure 2 (b)), and rule 2 indicates that a user cannot access `<credicard>` nodes, i.e., node 303. Naive enforcement of the rules will grant access to the record of element 293 and 299, and revoke the access to the element 303.

Now, a query "`//people`" is desired to yield an answer containing two `<person>` nodes, since they are the descendants (that are accessible) of the requested node.

1. {user, /site/people/person, read, +}
2. {user, /site/people/person/credicard, read, -}

(a)

DOCID	ELEMENTID	PATHID	ST	ED	
0	252	164	33996	36229	<people>
0	293	165	35592	35826	<person>
0	299	165	35832	36217	<person>
0	303	188	35989	36032	<creditcard>

(b)

```
SELECT  e0.DOCID, e0.ELEMENTID, e0.PATHID, e0.ST, e0.ED
FROM document d, element e0, pth p0
WHERE p0.pathexp LIKE '#%/people'
AND e0.pathid = p0.pathid AND d.docid = e0.docid
```

(c)

```
SELECT  e0.DOCID, e0.ELEMENTID, e0.PATHID, e0.ST, e0.ED
FROM document d, element e0, pth p0
WHERE p0.pathexp LIKE '#%/people#/person'
AND e0.pathid = p0.pathid AND d.docid = e0.docid
```

(d)

Figure 7.1. Examples of naive enforcement of "equivalent" relational rules leading to incorrect answer or security leakage.

However, the converted SQL query (Figure 7.1(c)) yields no answer since access to the record of element 252 is prohibited by default. Moreover, for a query "`//person`", the converted SQL (Figure 7.1(d)) returns both `<person>` nodes to the user (with the unauthorized `<creditcard>` node). This is so because both records of element 293 and 299 are accessible, while revoking access to element 303 does not affect its ancestor.

Practically, since XML access control rules often use XPath expressions to specify object nodes, values of $R_{Relational}.object$ are usually XPath expressions. In this way, the $Convert()$ function is the query translation function of corresponding X2R conversion and it returns SQL queries. $R_{relational}.object$ includes the cells selected by these SQL queries.

7.1 Object and Operation Equivalency

To solve the problem illustrated in Example 7.1, we propose our framework of supporting access control in XRDB systems. First, we define object and operation equivalency between XML and relational.

Definition 7.1 (Object Equivalency)**.** When both $R = C(X)$ and $X = C^{-1}(R)$ hold for XML node set X and relation R, we consider X and R equivalent w.r.t. C/C^{-1}, and denote as $X \equiv R$.

Note that, when we talk about equivalency of X and R, we have to predefine the context, i.e., select the X2R conversion algorithm C/C^{-1}. For a XML node set X, $C(X)$ may be different under different X2R conversion algorithms.

Definition 7.2 (Operation Equivalency)**.** Suppose $X_1 \equiv R_1$ and $X_2 \equiv R_2$ w.r.t. C/C^{-1}. Then, an XML operation OP_X is equivalent to a relational operation OP_R (denoted as $OP_X \equiv OP_R$) w.r.t. C and C^{-1} if:

$$C(X_1\ OP_X\ X_2) = C(X_1)\ OP_R\ C(X_2) = R_1\ OP_R\ R_2$$

It is worth to note that XML operator takes two node sets as operands while its equivalent relational counterpart may not take two generic relations as operands. Rather, each operand is the equivalent objects of the corresponding XML node set, which may be tables, columns, records, or cells. Many relational operations require operands to be domain compatible (e.g., intersect, union etc.). We loosen this requirement for OP_R.

With the concept of operation equivalency, we can migrate all the exciting features of XML into XRDB by converting the atomic operations into equivalent relational counterparts. Our problem of secure XRDB is then articulated as follows:

Lemma 7.1. *In XRDB(C), if we can find relational operators,* $\overset{D}{\cup}_R$, $\overset{D}{\cap}_R$, *and* $\overset{D}{-}_R$, *which are equivalent to XML deep set operators,* $\overset{D}{\cup}_X$, $\overset{D}{\cap}_X$, *and* $\overset{D}{-}_X$, *w.r.t. the X2R conversion algorithm C, we are able to enforce XML access control in XRDB(C) such that Equation (6.4) always holds.*

Proof. First, according to the definition of object and equivalency, we are looking for $A_X = SA_X \equiv SA_R$, which means: $SA_R = C(SA_X)$. Since $\overset{D}{\cup}_R \equiv \overset{D}{\cup}_X$, $\overset{D}{\cap}_R \equiv \overset{D}{\cap}_X$ and $\overset{D}{-}_R \equiv \overset{D}{-}_X$ w.r.t. $C()$ and $C^{-1}()$, according to the definition of equivalent operation, we have:

$$\begin{aligned}
SA_R &= \mathbf{C}(\langle Q_X\rangle \overset{D}{\cap}_X [(\langle R^+_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^+_{Xn}\rangle) \overset{D}{-}_X (\langle R^-_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^-_{Xm}\rangle)]) \\
&= \mathbf{C}(\langle Q_X\rangle) \overset{D}{\cap}_R \mathbf{C}([(\langle R^+_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^+_{Xn}\rangle) \overset{D}{-}_X (\langle R^-_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^-_{Xm}\rangle)]) \\
&= \mathbf{C}(\langle Q_X\rangle) \overset{D}{\cap}_R [\mathbf{C}(\langle R^+_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^+_{Xn}\rangle) \overset{D}{-}_R \mathbf{C}(\langle R^-_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^-_{Xm}\rangle)] \\
&= \mathbf{C}(\langle Q_X\rangle) \overset{D}{\cap}_R [\mathbf{C}(\langle R^+_{X1}\rangle) \overset{D}{\cup}_R ..\mathbf{C}(\langle R^+_{Xn}\rangle) \overset{D}{-}_R \mathbf{C}(\langle R^-_{X1}\rangle) \overset{D}{\cup}_R ..\mathbf{C}(\langle R^-_{Xm}\rangle)]
\end{aligned}$$

Since we have $Q_X \equiv Q_R$, $R_{Xi} \equiv R_{Ri}$, therefore:

$$\begin{aligned} & \mathbf{C}(\langle Q_X\rangle) \overset{D}{\cap}_R [\mathbf{C}(\langle R^+_{X1}\rangle) \ \overset{D}{\cup}_R ... \overset{D}{\cup}_R \ \mathbf{C}(\langle R^+_{Xn}\rangle) \ \overset{D}{-}_R \ \mathbf{C}(\langle R^-_{X1}\rangle) \overset{D}{\cup}_R ... \overset{D}{\cup}_R \mathbf{C}(\langle R^-_{Xm}\rangle)] \\ = \quad & \langle Q_R\rangle \overset{D}{\cap}_R [\langle R^+_{R1}\rangle \ \overset{D}{\cup}_R ... \overset{D}{\cup}_R \ \langle R^+_{Rn}\rangle \ \overset{D}{-}_R \ \langle R^-_{R1}\rangle \overset{D}{\cup}_R ... \overset{D}{\cup}_R \langle R^-_{Rm}\rangle] \end{aligned}$$

As a conclusion, we are able to compose a SA_R within XRDB(C) such that $SA_R = C(SA_X)$. Since all steps above are reversible, we also have $SA_X = C^{-1}(SA_R)$. □

According to Lemma 7.1, in order to support access control in XRDB, we need to find equivalent operations such that $\overset{D}{\cup}_R \equiv \overset{D}{\cup}_X$, $\overset{D}{\cap}_R \equiv \overset{D}{\cap}_X$ and $\overset{D}{-}_R \equiv \overset{D}{-}_X$. Object and operation equivalency is based on specific X2R conversion method, therefore, the existence and representation of relational deep set operators also heavily depends on the particular conversion method C. Hereafter, we analyze the role of each deep set operator in Equation 6.2 and the existence of its equivalent relational counterpart under different X2R conversion algorithms.

7.2 On Equivalent Conversion of Deep Set Operators

Deep-union operator is used to integrate all the nodes that are defined accessible by individual positive rules (also, all the nodes that are defined inaccessible by individual negative rules), as shown in Equation 6.2. With the property $P \overset{D}{\cup} Q \subseteq P \cup Q$ [46], Remark 1 is rewritten into:

$$\langle P\rangle \overset{D}{\cup}_X \langle Q\rangle = \{n | (n \in \langle P\rangle \vee \ n \in \langle Q\rangle) \ \wedge (n \notin \langle P//*\rangle \wedge \ n \notin \langle Q//*\rangle)\} \quad (7.1)$$

Since n is an XML object and $\langle P\rangle$, $\langle Q\rangle$, $\langle P//*\rangle$, $\langle Q//*\rangle$ are all sets of XML objects, when C/C^{-1} is sound according to Definition 3, we have:

$$\begin{aligned} C(\langle P\rangle \overset{D}{\cup}_X \langle Q\rangle) \ &= \ \{C(n) | [C(n) \in C(\langle P\rangle) \vee \ C(n) \in C(\langle Q\rangle)] \\ &\qquad \wedge [C(n) \notin C(\langle P//*\rangle) \wedge \ C(n) \notin C(\langle Q//*\rangle)]\} \\ &= \ \{r | [r \in C(\langle P\rangle) \vee \ r \in C(\langle Q\rangle)] \ \wedge [r \notin C(\langle P//*\rangle) \\ &\qquad \wedge \ r \notin C(\langle Q//*\rangle)]\} \end{aligned}$$

Here, since we are to find $\overset{D}{\cup}_R$ such that $C(\langle P\rangle) \overset{D}{\cup}_R C(\langle Q\rangle) = C(\langle P\rangle \overset{D}{\cup}_X \langle Q\rangle)$:

$$\begin{aligned} & C(\langle P\rangle) \overset{D}{\cup}_R C(\langle Q\rangle) \\ = \ & \{r | [r \in C(\langle P\rangle) \vee \ r \in C(\langle Q\rangle)] \ \wedge [r \notin C(\langle P//*\rangle) \wedge \ r \notin C(\langle Q//*\rangle)]\} \end{aligned}$$

The condition of $[r \in C(\langle P\rangle) \vee \ r \in C(\langle Q\rangle)]$ is essentially the regular union. It is composed by set containment and Boolean operations. In XRDB, set containment check is supported when the soundness requirement in Definition 6.1 is fulfilled, and Boolean operation is generally supported in RDBMS. $[r \notin C(\langle P//*\rangle) \wedge \ r \notin C(\langle Q//*\rangle)]$ tends to support deep semantics. It requires XRDB to be able to identify $r \in C(\langle P//*\rangle)$ for any given relational object r and set $C(\langle P\rangle)$. This could be achieved in two ways: (1) directly calculate the containment relationship between r and all elements of $C(\langle P\rangle)$; or (2) enumerate all descendants of each element of $C(\langle P\rangle)$, and check if n is identical to any of them.

Lemma 7.2. *To implement deep-union operator in XRDB(C), the X2R conversion algorithm C should: (1) fulfil the soundness requirement stated in Definition 6.1; and (2) for given node n and node set $\langle P\rangle$, it should be able to check the containment condition of: $C(n) \in C(\langle P//*\rangle)$, e.g., it should recognize if $C(n)$ is a descendant of any node $C(p_i)$;*

At present, all X2R conversion algorithms (we are aware of) fulfill the above conditions. Here we show an example using XRDB/XRel.

Example 7.2. We manage XMark documents in XRDB(XRel). Evaluating query "`//person`" yields a set of "`<person>`" nodes that are represented as the tuples shown in Figure 7.2(a) in XRDB. Likely, "`//name`" yields a set of "`<name>`" nodes (including both person and item names) as shown in Figure 7.2(b). In XRel, each XML node is marked with a "*start*" and an "*end*" offset. Node containment is checked through a comparison of these offsets: for two node p_1 and p_2, if $(p_1.start < p_2.start)$ and $(p_1.end > p_2.end)$, then p_2 is an descendant of p_1. In our example, we can tell that node 294 and 300 are descendants of node 293 and 299 respectively. Therefore, nodes 294 and 300 are not include in: "`//person` $\overset{D}{\cup}_X$ `//name`", which is shown in Figure 7.2(c).

DOCID	ELEMENTID	PATHID	ST	ED
0	293	165	35592	35826
0	299	165	35832	36217

(a)

DOCID	ELEMENTID	PATHID	ST	ED
0	6	7	178	212
0	61	49	4577	4602
0	32	29	2261	2279
0	294	167	35620	35640
0	300	167	35860	35885

(b)

DOCID	ELEMENTID	PATHID	ST	ED
0	6	7	178	212
0	61	49	4577	4602
0	32	29	2261	2279
0	293	165	35592	35826
0	299	165	35832	36217

(c)

(a) /site/people/person
(b) //name including /site/people/person/name
/site/regions/europe/item/name
/site/regions/namerica/item/name
(c) /site/people/person deep-union //name

Figure 7.2. Deep-union in XRDB(XRel).

If a naive implementation of XRDB access control fails to support deep-union, instead, it implements regular union operator to arbitrarily collect "accessible nodes" and "forbidden nodes" into two sets; and this will not cause any security leak. This is because the positive set does not contain extra node, and the negative set does not miss any necessary node. However, this will cause duplicate nodes in the sets of accessible nodes, and then possibly in the answers to queries.

Deep-intersect operator is used to calculate the exact overlapping of user requested data and accessible data (i.e. $\langle Q \rangle$ and $\langle ACR \rangle$). Like deep-union, deep-intersect operator is defined as:

$$C(\langle P \rangle \overset{D}{\cap}_X \langle Q \rangle) = \{r | [r \in C(\langle P \rangle) \wedge r \in C(\langle Q \rangle)] \wedge [r \notin C(\langle P//* \rangle) \vee r \notin C(\langle Q//* \rangle)]\} \tag{7.2}$$

Therefore, any X2R conversion algorithm that supports deep-union is able to support deep-intersect. That is, Lemma 7.2 could be directly extended to deep-intersect. On the other hand, if an XRDB fails to implement deep-intersect operator, instead, it uses regular intersection, as a result: (1) if a query asks for a descendant of an access-granted node, the whole node should be returned, but may be missed (i.e., mistakenly "jailed" by XRDB); (2) if a query asks for a node, where only part of its subtree is granted access, the access-granted descendants should be returned, but might be missed (such as shown in Example 7.1).

Example 7.3. In Example 7.1, A query "`//people`" yields `<people>` nodes, i.e. element 252, as shown in Figure 7.1 (b) record 1. Meanwhile, *object* field of access control rule 1, "`/site/people/person`", yields `<person>` nodes, i.e. element 293 and 299, as shown in Figure 7.1 (b) record 2 and 3. In XRel, each XML node is marked with a "*start*" and an "*end*" offset. Node containment

is checked through a comparison of these offsets: for two node p_1 and p_2, if ($p_1.start < p_2.start$) and ($p_1.end > p_2.end$), then p_2 is an descendant of p_1. In our example, we can tell that node 293 and 299 are descendants of node 292. Therefore, "`//people` $\overset{D}{\cap}_X$ `//person`" will yield node 293 and node 299. Comparing with Example 7.1, "`//people` $\cap$ `//person`" yields $Null$.

The operands of XML deep-union/intersect operators may contain different nodes. In RDBMS, where domain compatibility is strictly enforced, their relational equivalent counterpart might be domain incompatible (e.g. a row "intersect" a cell). This happens when schema-based X2R conversion methods (e.g. [20, 66]) are employed, where different XML nodes could be converted to tables, rows, or cells. To tackle this problem, we can employ new RDBMS techniques such as Oracle VPD (Virtual Private Database) to enable us to fine-control relational tables to create relational views with any group of cells from a table.

Deep-except is used to remove inaccessible nodes from the answer. Recall that, in our XML access control model, all nodes are inaccessible by default. When a user is prohibited to access a node, there is no need to write a negative rule (R^-) to revoke its accessibility unless the node is covered by positive rules (ACR^+). In this way, negative rules are only used to specify exceptions to global permissions, i.e. "revoke" access proposed by ACR^+. Deep except operator is used to enforce negative rules. Regarding whether deep except could be implemented in XRDB with X2R conversion algorithm C, it depends upon the characteristics of the negative rules contained in the access control policy. In particular, we distinguish two types of negative rules, as shown below.

Definition 7.3 (Node elimination vs. Descendant elimination negative rules)**.** A negative rule in ACR restricts user from access a set of nodes $\{r_1^-, ...r_n^-\}$. If **none** of the nodes is a descendant of the context node of a positive rule, i.e.:

$$r_i^- \notin \langle R^+//* \rangle, \quad \forall r_i^- \in \{r_1^-, ...r_n^-\}; \forall \langle R^+ \rangle \in \langle ACR^+ \rangle$$

then it is called a **node elimination** (**NE**) negative rule. Else, if one of the nodes is a descendant of the context node of a positive rule, i.e.:

$$r_i^- \in \langle R^+//* \rangle, \quad \exists r_i^- \in \{r_1^-, ...r_n^-\}; \exists \langle R^+ \rangle \in \langle ACR^+ \rangle$$

it is called a **descendant elimination** (**DE**) negative rule.

Intuitively, a "*Node elimination*" (NE) negative rule removes context node from $\langle ACR^+ \rangle$, while a "*descendant elimination*" (DE) negative rule removes descendants from context node of $\langle ACR^+ \rangle$.

For XML nodes covered by node elimination negative rules $\langle ACR_1^- \rangle$, deep-except operator directly removes them from $\langle ACR^+ \rangle$, without breaking any XML nodes in the original document or creating any new nodes:

$$\langle ACR^+ \rangle \overset{D}{-}_X \langle ACR_1^- \rangle = \{n | n \in \langle ACR^+ \rangle \ \wedge \ n \notin \langle ACR_1^- \rangle\}$$

Essentially, this is the regular except semantics. In this way, in XRDB, we have,

$$\begin{aligned} & C(\langle ACR^+ \rangle) \overset{D}{-}_R C(\langle ACR_1^- \rangle) \\ = \ & C(\langle ACR^+ \rangle \overset{D}{-}_X \langle ACR_1^- \rangle) \\ = \ & \{r | r \in C(\langle ACR^+ \rangle) \ \wedge \ r \notin C(\langle ACR_1^- \rangle)\} \end{aligned}$$

To support deep except operator for node elimination negative rules only, the conditions described in Lemma 7.2 still apply. However, it takes more burden to process descendant elimination negative rules, where real "deep" semantics is required. That is,

$$\langle ACR^+ \rangle \overset{D}{-}_X \langle ACR_2^- \rangle = \{deepRemove(n, n \overset{D}{\cap}_X \langle ACR_2^- \rangle) | n \in \langle ACR^+ \rangle\}$$

where $deepRemove(p, \langle Q \rangle)$ takes a node and a set of its descendants as operands, removes the descendants from the subtree of the node and return the remaining. This function may not be directly converted to relational.

Lemma 7.3. *When a deep-except operator takes nodes specified by descendant elimination negative rules as the second operand, it is implemented through deepRemove() operation. To implement deep-except operator that supports descendant elimination negative rules in XRDB(C), the X2R conversion algorithm X should: (1) fully satisfy Lemma 7.2; and (2) for any node n_1 and its descendant n_2, $C(n_2)$ should be part of $C(n_1)$; and in the reverse conversion of $n_1 = C^{-1}(C(n_1))$, node n_2 in the subtree is entirely converted from $C(n_2)$.*

Example 7.4. For instance, in Example 7.1, Rule 2 is a descendant elimination negative rule since it revoke access towards descendants of Rule 1's context node (`<person>`).

In XRDB(XRel) [72], descendants are converted to independent records that are stand alone from ancestors. As shown in Figure 7.1(b), `<creditcard>` node is converted to an individual record (i.e. $elementID = 303$), which is independent from it ancestor `<person>`. To reconstruct a `<person>` node, $C^{-1}_{XRel}()$ only takes the record with $elementID = 293$ (ancestor node) and returns a full person node. Although the descendant node `<creditcard>` is included in the answer, the record $elementID = 303$ is not touched by $C^{-1}_{XRel}()$. In this way, XRel violates condition (2) of Lemma 7.3, so that we cannot directly implement deep-except operator to support descendant elimination negative rules. When user requests for "`//person`", we are not able to revoke access towards `<creditcard>` child, unless we modify the relational data to the following record for $C^{-1}_{XRel}()$:

DOCID	ELEMENTID	pathID	st	ed
0	NULL	NULL	35832	35988
0	NULL	NULL	36033	36217

However, this is not directly supported in relational algebra or any existing RDBMS product.

In Shared-Inlining [66] approach, `<person>` nodes are translated into a table, with each row representing a person, and `<creditcard>` nodes are stored in one of the columns, "`person_credicard`". The relational schema is [42]:

```
Person(PersonId, ParentId, Person, Person_address,
       Person_address_city, Person_address_country,
       Person_address_province, Person_address_street,
       Person_address_zipcode, Person_creditcard, ......)
```

Here, the ancestor-descendant relationship is kept such that each row represents a "`person`" node, and each cell represents a child node. When $C^{-1}_{Inlining}()$ is called to reconstruct `<person>` nodes, the textual contents of `<credicard>` descendants are retrieved from "`person_credicard`" column. Therefore, to obtain `//person` $\overset{D}{-}_X$ `//creditcard`, we just mask "`person_creditcard`" column in the table; and the

reconstructed XML tree of "`person`" node will not have corresponding child, i.e., "`creditcard`" node is removed from the XML answer.

As a conclusion, there is a "semantic gap" between XML and relational data models. XML features a tree structure, where nodes are hierarchically nested, while, relational model only defines a two-dimensional structure. This fundamental difference makes us unable to directly maintain all structural information in X2R conversion. Some conversion approaches store each XML node independently, such as XRel showed above, where descendants are not utilized when converting an ancestor node back to XML. This is different from XML data model, in which ancestor node inherently includes descendants. For those approaches like XRel, descendant elimination negative rules could not be directly enforced through deep-except operator since we have difficulties sweeping off descendants from given node(s). Fortunately, for many other X2R conversion approaches (like shared-inlining), we are able to implement deep-except operator, and then directly enforce descendant elimination negative rules. Moreover, in those approaches where descendant elimination negative rules are not directly supported, we can still use post-processing filtering methods to remove access denied contents from the reconstructed XML answer (more details are provided in the next section).

Chapter 8

XML Access Control Enforcement in XRDB

In the previous sections we show how XML access control semantics could be converted into relational model to be used in XRDB. However, in real world applications, existing XML access control approaches do not exactly implement the basic semantics shown in Equation 3. A general framework is proposed in [44] to capture different XML access control approaches (as show in row 1 of Figure 8.1). Now, we extend this framework into XRDB.

As shown in Figures 2.4 and 8.1, similar to the framework in native XML

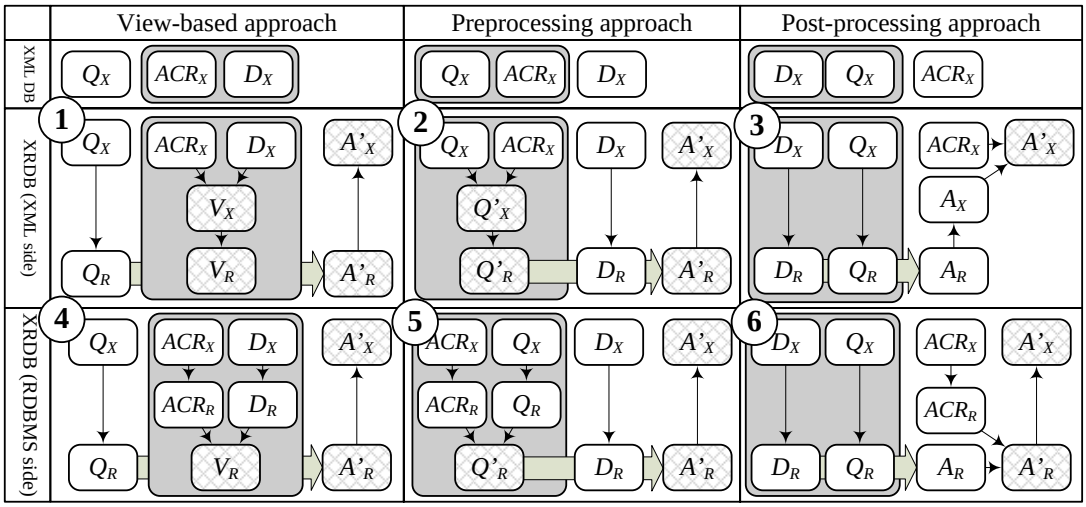


Figure 8.1. Access control enforcement approaches in XML DB and XRDB.

databases, there could be three categories of XML access control enforcement mechanisms in XRDB: (1) view-based approach (① ④ in Figures 2.4 and 8.1); (2) pre-processing approach (② ⑤ in Figures 2.4 and 8.1); and (3) post-processing approach (③ ⑥ in Figures 2.4 and 8.1). These approaches enforce access control policy on the document, query and answer, respectively. In this section, we articulate the algebra of these approaches using deep set operators. Then, we briefly describe how they could be converted to their relational counter parts in XRDB.

8.1 View-based Approach

When access control is first enforced on XML documents to create *views*, it is the traditional view-based approach. In this model, XML view V_X (or *safe document* SD) is constructed to capture:

$$V_X = [(\langle R^+_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^+_n\rangle) \overset{D}{-}_X (\langle R^-_1\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^-_m\rangle)]$$

And query is evaluated against the view

$$SA = Q\langle V_X\rangle = Q[(\langle R^+_{X1}\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^+_n\rangle) \overset{D}{-}_X (\langle R^-_1\rangle \overset{D}{\cup}_X ... \overset{D}{\cup}_X \langle R^-_m\rangle)] \quad (8.1)$$

To convert this approach into XRDB, a straightforward approach is to convert each XML view V_X into relational view $V_R = C(V_X)$, as shown in ① of Figure 8.1:

$$SA = C^{-1}(Q_R\langle V_R\rangle) = C^{-1}(Q_R\langle C(V_X)\rangle)$$

However, this approach suffers from several drawbacks: (1) since views for each role should be materialized, the storage requirement is substantial; and (2) each relational view V_R is independently stored, without any connection to D_R, thus synchronization is difficult to achieve, if not impossible.

Another solution is to employ view support from RDBMS to enforce access control on the relational side of XRDB, as show in ④ of Figure 8.1:

$$SA = C^{-1}(Q\langle V_R\rangle)$$

$$= C^{-1}(Q\langle(C(\langle R^+_{X1}\rangle) \overset{D}{\cup}_R ... \overset{D}{\cup}_R C(\langle R^+_n\rangle)) \overset{D}{-}_X (C(\langle R^-_1\rangle) \overset{D}{\cup}_R ... \overset{D}{\cup}_R C(\langle R^-_m\rangle))\rangle)$$

In implementation of view based approaches, there are three factors to be considered:

Construction of V_R: This issue includes two aspects: (1) the constructed V_R should capture the exact content of access control allowed data, i.e. $V_R \equiv V_X$, as we described in Section 4; and (2) this V_R should be legit to the underlying RDBMS. According to Lemma 7.3, some X2R conversion algorithms cannot directly support descendant elimination negative rules. Therefore, in the corresponding XRDB systems, we cannot directly employ relational view based approaches to enforce descendant elimination negative rules.

Evaluation of Q_R: Comparing Equation 8.1 with Equation 6.4, note that, in Equation 8.1 deep-intersect operator is replaced by the query evaluation process. Then we need to consider whether query evaluation process conducts the deep intersect semantics. In some XRDB such as XRDB(XRel), the original query translation and evaluation process only conducts intersect semantics, as shown in Example 7.1. Therefore, we cannot directly employ view based approach, or special treatment is required to implement the deep-intersect semantics.

Reconstruction of SA_X: We still need to mention that, no matter how we tailor D_R into V_R, we need to ensure that the relational answers from V_R is legit to $C^{-1}()$.

8.2 Pre-processing Approach

In preprocessing model, *safe query SQ* is constructed as:

$$SQ_X = Q_X \overset{D}{\cap}_X [(\ R^+_{X1}\ \overset{D}{\cup}_X ... \overset{D}{\cup}_X\ R^+_{Xn})\ \overset{D}{-}_X\ (R^-_{X1} \overset{D}{\cup}_X ... \overset{D}{\cup}_X R^-_{Xm})]$$

Safe answer is yielded by evaluating safe query against the original document: $SA_X = SQ_X\langle D_X\rangle$. To extend this approach to XRDB, we have two methods:

(1)XML Query Rewriting: as shown in ② in Figure 8.1, this approach is to convert the safe XML query into SQL, and follow the regular XRDB query

evaluation process:

$$SA_X = C^{-1}(SQ_R\langle D_R\rangle) = C^{-1}(C(SQ_X)\langle D_R\rangle)$$

In this approach, we can directly adopt the preprocessing of XML access control mechanisms, such as [45], to generate SA_X. We have to mention that, the generated SA_X should be legit to the X2R conversion algorithm, e.g. most X2R conversion algorithms can only process XPath queries, thus SA_X should only include XPath. However, this requirement may exceed the capability of XML access control mechanisms since XML deep set operators are implemented as user defined functions of XQuery, which is not supported in some X2R conversion algorithms. Therefore, when the safe XML query cannot be expressed as XPath, one cannot directly adopt XML query rewritten approach to enforce access control.

(2)Relational Query Rewriting: As shown in ⑤ in Figure 8.1, this approach follows regular XRDB query evaluation process to convert user XML query Q_X into SQL Q_R. Then, we conduct query rewriting on Q_R to generate safe query SQ_R:

$$SQ_R = Q_R \overset{D}{\cap}_R [(\ R^+_{R1}\ \overset{D}{\cup}_R ... \overset{D}{\cup}_R\ R^+_{Rn})\ \overset{D}{-}_X\ (R^-_{R1} \overset{D}{\cup}_X ... \overset{D}{\cup}_X R^-_{Rm})]$$

We present two approaches to implement this. First, develop an external query rewriting process, which sits as a middle-ware between X2R query conversion and relational query evaluation. Since we have clearly defined relational deep set operators, the implementation is straightforward, although the queries might be complicated. For instance,

Example 8.1. Let use revisit the previous examples: we manage XMark document in XRDB(XRel). Suppose we have access control rule `(user, //people, read, +)`, and user submits query `//name`. Figure 8.2(a) shows the relational query for $C(//people) \overset{D}{\cap}_R C(//name)$, which is implemented according to the definition in Equation 7.2 (we marked up all the sub-queries). Moreover, this query could be further optimized, as shown in Figure 8.2(b).

Second method is to use Oracle VPD. Oracle version 8.1.5 introduces a new security feature supporting non-view-based fine-grained access control, namely *Row*

```
SELECT   docID, pathid, elementID, st, ed FROM element
WHERE  (elementID IN                                          r∈C(//people::self-or-descendant())
          ( SELECT e0.elementID FROM document d, element e0, pth p0
            WHERE   p0.pathexp LIKE '#%/people%'  AND e0.pathid = p0.pathid AND d.docid = e0.docid )
        AND elementID IN                                      r∈C(//name::self-or-descendant())
          (SELECT e0.elementID FROM document d, element e0, pth p0
           WHERE p0.pathexp LIKE '#%/name%' AND e0.pathid = p0.pathid AND d.docid = e0.docid ))
        AND ( NOT elementID IN                                r∈C(//people//*)
          (SELECT e0.elementID FROM document d, element e0, pth p0
           WHERE p0.pathexp LIKE '#%/people#/%' AND e0.pathid = p0.pathid AND d.docid = e0.docid)
        OR NOT elementID IN                                   r∈C(//name//*)
          (SELECT e0.elementID FROM document d, element e0, pth p0
           WHERE p0.pathexp LIKE '#%/name#/%' AND e0.pathid = p0.pathid AND d.docid = e0.docid))
```

(a) SQL query for: //people $\overset{D}{\cap}$ //name

```
SELECT   e0.docID, e0.elementID, e0.st, e0.ed FROM demo.document d, demo.element e0, demo.pth p0
WHERE   (( p0.pathexp LIKE '#%/people%'  AND e0.pathid = p0.pathid AND d.docid = e0.docid)
AND      (p0.pathexp LIKE '#%/name%' AND e0.pathid = p0.pathid AND d.docid = e0.docid ))
AND      ((NOT p0.pathexp LIKE '#%/people#/%' AND e0.pathid = p0.pathid AND d.docid = e0.docid)
OR       (NOT p0.pathexp LIKE '#%/name#/%' AND e0.pathid = p0.pathid AND d.docid = e0.docid))
```

(b) optimized SQL query

Figure 8.2. Enforcing XML access control via external pre-processing

Level Security or *Virtual Private Database*. In SQL, user could only use GRANT and REVOKE statements to enforce access control at column or higher level. Fine-grained access control is only enforced via views, which suffer from expensive maintenance and excessive storage needs. It allows users to control accessibility towards row/cell level. In VPD, to restrict users' access to rows, a policy function is defined to generate additional predicates and attach them to the WHERE clause of the user query. Moreover, VPD allows user to "mask" individual cells to support cell level access control. Other access control mechanism through SQL rewriting or relational views can only work on relations. However, with VPD, we are able to tailor relational data into any shape we want.

To utilize VPD for access control in XRDB, we first construct relational predicates from the converted relational access control rules ACR_R, then define a VPD policy to enforce the predicates on converted SQL queries. Moreover, cell level access control capability of VPD is of special importance to XRDB systems that use schema-based X2R conversion algorithm, such as Inlining. In those XRDB systems, XML nodes are converted to different types of relational objects: tables, rows and cells. In this way, $\langle ACR \rangle$ may not be conventional relations, e.g. it could be arbitrary combinations of columns, rows and/or individual cells.

8.3 Post-processing Based Approach

In native XML DB, access control through post-processing described as:

$$SA_X = ACR\langle A_X\rangle = [(\ R^+_{X1}\ \overset{D}{\cup}_X ... \overset{D}{\cup}_X\ R^+_{Xn})\ \overset{D}{-}_X\ (R^-_{X1} \overset{D}{\cup}_X ... \overset{D}{\cup}_X R^-_{Xm})]\langle Q_X\langle D_X\rangle\rangle$$

In XRDB, this approach could be conducted through: (1) XML answer filtering (③ in Figure 8.1); or (2) relational answer filtering (⑥ in Figure 8.1). (1) is similar to the postprocessing approach described in [44], while (2) evaluates relational query Q_R to obtain unsafe relational answer, and process ACR_R against the answers:

$$SA_X = C^{-1}(SA_R) = C^{-1}(ACR_R\langle A_R\rangle) = C^{-1}(ACR_R\langle Q_R\langle D_R\rangle\rangle)$$

However, the post-processing filters often require the intermediate answers ($\langle A_R\rangle$ or $\langle A_X\rangle$) to retain additional information of the original paths for ACR to operate on. As an example of this approach, [9] check streaming XML data against both query and ACR at the same time. Since it works in the streaming data environment, full paths are retained. As an counter example, let us look at an XRDB in information pull model. Suppose a user asks for "`//name`", but she is only authorized to access person names, not item names. To enforce access control on $\langle A_R\rangle$ or $\langle A_X\rangle$, we need to be able to distinguish these two types of `<name>` nodes, i.e. recognize the original full path. Unfortunately, as designed in most X2R conversion algorithms, the intermediate answer A_R or A_X does not contain such information. Therefore, postprocessing approaches are not suitable for all applications.

8.4 Descendant Elimination Negative Rules

As we described in Section 3, enforcing descendant elimination negative rules needs the conversion of deep-except operator. Due to the semantic gap between XML and Relational data models, this may not be feasible in all XRDB systems. E.g., in XRel, to enforce descendant elimination negative rules, we need to block access to a descendant node in the `element` table. However, users are still able to retrieve the whole ancestor node (including the "blocked" descendant) since it is stored as an independent record. To avoid security leak, we need to manage these conversion

algorithms with extra treatment: an external post processing to enforce DE access control rules.

Let us revisit Example 7.4 again. Upon user query "`//person`", elements 293 and 299 shown in Figure 7.1 are included in the relational answer. After the reverse conversion, segments with offsets (35592, 35826) and (35832, 36217) from the XML documents are returned to the user. However, rule 2 restricts access to `//person/creditcard` nodes, thus this answer is not safe. To remove the restricted node from the answer, we first request $\langle Q_R \rangle \overset{D}{\cap}_R \langle ACR_R^- \rangle$ from RDBMS, to yield element 303. Then we remove this segment, i.e. (35989, 36032) from the XML answer.

8.5 Experimental Validation

To show that the proposed theory and implementations are practical yet efficient, we show our experimental results.

8.5.1 Setting

An XML document with 8517 nodes are generated by XMark [64], mimicking online auction scenario. Part of its schema structure is shown in Figure 2.4. We use XRDB(XRel) [72][1], with Oracle 10g as underlying RDBMS; i.e. we convert XML document into relations using XRel, and manage them in Oracle 10g.

We design five (5) roles, abbreviated as A (administrator), M (manager), RU (registered user), S (sales) and U (unregistered user), respectively. Roles have different levels of accessibility, e.g. U is able to access 5% of total nodes, RU is able to access 40%, and A could access all.

According to Lemma 7.3, XRDB(XRel) cannot directly handle descendant elimination negative rules, thus we only have positive and node elimination negative rules. As a reference, we also test situations where no access control is enforced – user could access everything.

As we described before, the types of queries that we support totally depend on the X2R conversion algorithm. XRel supports a subset of XPath, with parent-child

[1] We choose XRel because of its available implementations of both Query and Data convertor.

(/), ancestor-descendant (//) axes, wildcards (*) and predicates. We generate four groups of synthetic XPath queries, each has a different setting of wildcards and predicates.

8.5.2 Experimental Results

In the XRDB(XRel) environment described above, we convert all access control rules into relational, and enforce them through views and VPD. For a comparison, we also enforce same rule sets on the same XML document in native XML environment. We enforce XML access control rules using QFilter [45], and answer XML queries using Galax. In all the experiments, we use the *query processing time* as an evaluation metric.

Figure 8.3 shows the result of our experimentation. Comparing both view-based and VPD-based approaches with the reference (no security enforcement), our approaches do not add much overhead for fine-grained access control. Meanwhile, the size of accessible data tends to get smaller after security enforcement. Therefore, querying on smaller set of records is even faster than that on no-security case. XRDB query processing speed is significantly slower for Query Sets 3 and 4. This is because the XML queries have predicates, and they are converted to nested SQL queries under XRel.

With access control enforced, performance of XML querying in XRDB systems or native XML database systems (with QFilter security enforcement) is similar. Note that comparison with Galax is not perfectly fair since Galax is just an XQuery implementation, and does not have storage management or cache. Therefore, Galax take more time to load XML documents from disk to memory. It is just used as a reference.

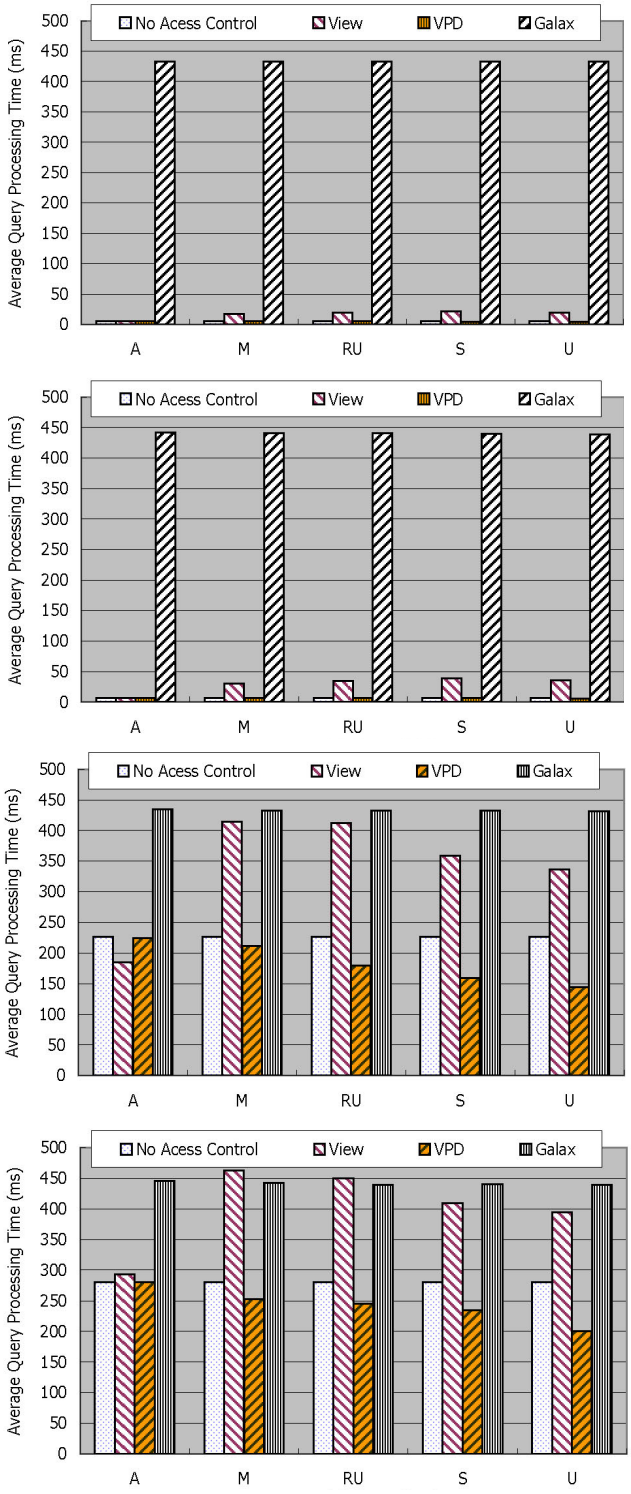


Figure 8.3. Query processing time for four sets of queries.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

In this thesis, we have addressed XML access control in native and RDBMS-supported XML database systems.

In native XML environment, we first introduce deep set operators and formally describe XML access control semantics with them. We propose a unified framework to capture all the XML access control enforcement approaches: building blocks, operations and their combinations. The approaches are qualitatively compared and each one is further described with deep set operators. Three approaches of novel solutions are presented to support XML access control without using views or security-support of underlying databases. In particular, a pre-processing based method, called QFilter, has been elaborated and shown to be particularly efficient and effective. QFilter, based on Non-deterministic Finite Automata (NFA), rewrites user's insecure queries to secure ones, not returning any data violating access control rules. We validate QFilter by showing it does not return any violating data via theoretical analysis, and by demonstrating its effectiveness through extensive experiments. As a result, QFilter demonstrates efficient and effective XML access control capabilities: (1) it does not require support from underlying database engine, which makes it feasible for any XML DBMS, native or RDBMS-based; (2) it consumes very small amount of memory (considering it is an NFA), especially comparing with traditional view-based approaches (an intensive study of QFilter memory consumption and optimization can be found at [41]); and (3)

its execution time is very short so that it is practical for real world applications.

In RDBMS-supported XML database systems (XRDB), we articulate the problem of XML access control as the problem of object and operation equivalency and conversion between XML and relational data models. We show that, equivalent counterparts of deep set operators in relational model are needed to fully implement XML access control in XRDB. We analyze the definition and semantics of each operator, and show how they can be converted to XRDB through two lemmas. Although detailed conversion implementation is connected with a specific X2R conversion algorithm used in XRDB, we propose an algebraic description of these operators. Moreover, we study possible implementations of XML access control in XRDB. We categorize them into three approaches, and formally describe the semantics of each approach using deep set operators. We also discuss the features and considerations of each approach. Finally, we show the validity of our approaches using experiment results.

9.2 Future Work

In this thesis, we have carefully explored the problem space of XML access control in XDB and XRDB. We have proposed theocratical and practical solutions, and discussed implementation approaches. However, there are still open questions in XDB and XRDB access control, especially the questions connected with particular implementation methods. E.g. how to enforce XML access control with minimal overhead and alternation upon underlying RDBMS? We leave these as our our future research topics. More specifically, the following topics could be further explored:

1. Current version of QFilter supports a subset of XPath: `/x`, `/*`, `//x`, `//*` and predicates. Although these are the most frequently used features of XPath, it is still desirable to cover the entire specification. This could be done by: (1) extending QFilter to support the entire set of XPath; or (2) rewriting unsupported queries into supported subset of XPath, using the current QFilter.

2. Currently, we support the 4 or 5-tuple XML access control model, which is

widely used in XML access control enforcement research. However, as more complicated hence powerful access control models are to be proposed (and standardized), it becomes more desirable to support these advanced XML access control models. Especially, further optimization of QFilter based on access control model features is highly anticipated.

3. We have summarized existing XML access control approaches into categories of built-in, view based, preprocessing, and post-processing. We have the inherent pros and cons of each approach. However, there could be a systematic study to evaluate performance of different approaches for different XML engines. This engine-dependent performance study could be used to suggest the best access control enforcement approach when the application scenario and underlying XML engine is given.

4. Moreover, there have not been any research on taking advantage of combining multiple XML access control approaches to build a hybrid mechanism. In many cases, a unified global security enforcement may be ineffective or inefficient. In a hybrid approach, security enforcement is tailored for individual rule or designated set of rules, based on the property of rules and data sources. This design will provide better flexibility and efficiency.

5. How to efficiently enforce *Descendant Elimination* negative rules in XRDB remains an open problem.

All the above are feasible extensions of this dissertation. Working on these problems may lead to more interesting issues. The journey of academic research is endless – thanks to the magic world of data management, security and privacy!

Bibliography

[1] Ayyagari, P., Mitra, P., Lee, D., Liu, P., and Lee, W.-C. Incremental adaptation of xpath access control views. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security* (New York, NY, USA, 2007), ACM, pp. 105–116.

[2] Barbosa, D., Freire, J., and Mendelzon, A. O. "Designing Information-preserving Mapping Schemes for XML". In *VLDB* (Trondheim, Norway, 2005), pp. 109–120.

[3] Berglund, A., Boag, S., Chamberlin, D., Fernndez, M. F., Kay, M., Robie, J., and Simeon, J. "XML Path Language (XPath) 2.0". W3C Working Draft, Nov. 2003. http://www.w3.org/TR/xpath20.

[4] Bertino, E., and Ferrari, E. "Secure and Selective Dissemination of XML Documents". *ACM Trans. on Information and System Security (TISSEC) 5*, 3 (Aug. 2002), 290–331.

[5] Bertino, E., Ferrari, E., and Provenza, L. P. Signature and access control policies for xml documents. In *ESORICS* (2003), E. Snekkenes and D. Gollmann, Eds., vol. 2808 of *Lecture Notes in Computer Science*, Springer, pp. 1–22.

[6] Beyer, K., Ozcan, F., Saiprasad, S., and der Linden, B. V. DB2/XML: designing for evolution. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2005), ACM Press, pp. 948–952.

[7] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., and Siméon, J. "XQuery 1.0: An XML Query Language". W3C Working Draft, Feb. 2005.

[8] Boag, S., Chamberlin, D., Fernndez, M. F., Florescu, D., Robie, J., and Simeon, J. "XQuery 1.0: An XML Query Language". W3C Working Draft, Nov. 2003. http://www.w3.org/TR/xquery.

[9] Bouganim, L., Ngoc, F. D., and Pucheral, P. "Client-Based Access Control Management for XML Documents". In *VLDB* (Toronto, Canada, 2004).

[10] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., and Yergeau, F. "Extensible Markup Language (XML) 1.0 (Fourth Edition)". W3C Recommendation, Aug. 2006. http://www.w3.org/TR/2006/REC-xml-20060816/.

[11] Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F., and Cowan, J. XML 1.1 (Second Edition). W3C Recommendation, Aug. 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/.

[12] Bray, T., Paoli, J., and Sperberg-McQueen (Eds), C. M. "Extensible Markup Language (XML) 1.0 (2nd Ed.)". W3C Recommendation, Oct. 2000. http://www.w3.org/TR/2000/REC-xml-20001006.

[13] Buneman, P., Deutsch, A., and Tan, W.-C. "A Deterministic Model for Semistructured Data". In *Workshop on Query Processing for Semistructured Data and Non-Standard Data Formats* (1998).

[14] Carminati, B., Ferrari, E., and Bertino, E. Securing xml data in third-party distribution systems. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management* (New York, NY, USA, 2005), ACM Press, pp. 99–106.

[15] Cho, S., Amer-Yahia, S., Lakshmanan, L. V., and Srivastava, D. "Optimizing the Secure Evaluation of Twig Queries". In *VLDB* (Hong Kong, China, Aug. 2002).

[16] Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., and Samarati, P. "A Fine-Grained Access Control System for XML Documents". *ACM Trans. on Information and System Security (TISSEC) 5*, 2 (May 2002), 169–202.

[17] Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P. Securing xml documents. In *EDBT* (2000), C. Zaniolo, P. C. Lockemann, M. H. Scholl, and T. Grust, Eds., vol. 1777 of *Lecture Notes in Computer Science*, Springer, pp. 121–135.

[18] Damiani, E., di Vimercati, S. D. C., Paraboschi, S., and Samarati, P. Xml access control systems: A component-based approach. In *DBSec 00: Proceedings of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security* (Deventer, The Netherlands, The Netherlands, 2001), Kluwer, B.V., pp. 39–50.

[19] Damiani, E., Vimercati, S. D. C. D., Paraboschi, S., and Samarati, P. "Design and Implementation of an Access Control Processor for XML Documents". *Computer Networks 33*, 6 (2000), 59–75.

[20] Deutsch, A., Fernandez, M. F., and Suciu, D. "Storing Semistructured Data with STORED". In *ACM SIGMOD* (Philadephia, PA, Jun. 1998).

[21] Diao, Y., and Franklin, M. J. "High-Performance XML Filtering: An Overview of YFilter". *IEEE Data Eng. Bulletin* (Mar. 2003).

[22] Draper, D., Fankhauser, P., Fernandez, M., Malhotra, A., Ross, K., rys, M., Siméon, J., and Wadler (Eds), P. "XQuery 1.0 and XPath 2.0 Formal Semantics". W3C Working Draft, Apr. 2004. http://www.w3.org/TR/xquery-semantics/.

[23] Fan, W., Chan, C.-Y., and Garofalakis, M. Secure xml querying with security views. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2004), ACM Press, pp. 587–598.

[24] Fernandez, E., Gudes, E., and Song, H. "A Model of Evaluation and Administration of Security in Object-Oriented Databases". *IEEE Trans. on Knowledge and Data Engineering (TKDE) 6*, 2 (1994), 275–292.

[25] Finance, B., Medjdoub, S., and Pucheral, P. The case for access control on xml relationships. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management* (New York, NY, USA, 2005), ACM Press, pp. 107–114.

[26] Florescu, D., and Kossmann, D. "Storing and Querying XML Data Using an RDBMS". *IEEE Data Eng. Bulletin 22*, 3 (Sep. 1999), 27–34.

[27] Fundulaki, I., and Marx, M. Specifying access control policies for xml documents with xpath. In *SACMAT '04: Proceedings of the ninth ACM symposium on Access control models and technologies* (New York, NY, USA, 2004), ACM Press, pp. 61–69.

[28] Gabillon, A., and Bruno, E. Regulating access to xml documents. In *DBSec 01: Proceedings of the fifteenth annual working conference on Database*

and application security (Norwell, MA, USA, 2002), Kluwer Academic Publishers, pp. 299–314.

[29] Godik, S., and Moses (Eds), T. "eXtensible Access Control Markup Language (XACML) Version 1.0". OASIS Specification Set, Feb. 2003. http://www.oasis-open.org/committees/xacml/repository/.

[30] Griffiths, P. P., and Wade, B. W. "An Authorization Mechanism for a Relational Database System". *ACM Trans. on Database Systems (TODS) 1*, 3 (Sep. 1976), 242–255.

[31] Hacigumus, H., Iyer, B. R., Li, C., and Mehrotra, S. "Executing SQL over encrypted data in the database-service-provider model". In *ACM SIGMOD* (2002).

[32] Jagadish, H. V., Lakshmanan, L. V. S., Srivastava, D., and Thompson, K. "TAX: A Tree Algebra for XML". In *Int'l Workshop on Data Bases and Programming Languages (DBPL)* (Frascati, Rome, Sep. 2001).

[33] Jajodia, S., Samarati, P., Sapino, M. L., and Subrahmanian, V. S. "Flexible Support for Multiple Access Control Policies". *ACM Trans. on Database Systems (TODS) 26*, 2 (Jun. 2001), 214–260.

[34] Jajodia, S., Samarati, P., Subrahmanian, V. S., and Bertino, E. "A Unified Framework for Enforcing Multiple Access Control Policies". In *ACM SIGMOD* (May 1997), pp. 474–485.

[35] Jajodia, S., and Sandhu, R. "Toward a Multilevel Secure Relational Data Model". In *ACM SIGMOD* (May 1990).

[36] Jiang, M., and Fu, A. W.-C. Integration and efficient lookup of compressed xml accessibility maps. *IEEE Transactions on Knowledge and Data Engineering 17*, 7 (2005), 939–953.

[37] Kudo, M., and Hada, S. "XML Document Security Based on Provisional Authorization". In *ACM Conf. on Computer and Communications Security (CCS)* (2000).

[38] Kuper, G., Massacci, F., and Rassadko, N. Generalized xml security views. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies* (New York, NY, USA, 2005), ACM Press, pp. 77–84.

[39] Lee, D., and Chu, W. W. "Constraints-preserving Transformation from XML Document Type Definition to Relational Schema". In *Int'l Conf. on Conceptual Modeling (ER)* (Salt Lake City, UT, Oct. 2000), pp. 323–338.

[40] Lee, D., Lee, W.-C., and Liu, P. "Supporting XML Security Models using Relational Databases: A Vision". In *XML Database Symp. (XSym)* (Berlin, Germany, Sep. 2003).

[41] Li, F., Luo, B., Liu, P., Lee, D., Mitra, P., Lee, W.-C., and Chu, C.-H. In-broker access control: Towards efficient end-to-end performance of information brokerage systems. In *SUTC '06: Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing - Vol 1 (SUTC'06)* (Washington, DC, USA, 2006), IEEE Computer Society, pp. 252–259.

[42] Lu et al., H. What makes the differences: benchmarking xml database implementations. *ACM Trans. on Internet Technology (TOIT) 5*, 1 (2005), 154–194.

[43] Luo, B., Lee, D., , and Liu, P. Pragmatic XML Access Control using Off-the-shelf RDBMS. In *12th European Symposium On Research In Computer Security (ESORICS)* (Dresden, Germany, September 2007).

[44] Luo, B., Lee, D., Lee, W.-C., and Liu, P. "A Flexible Framework for Architecting XML Access Control Enforcement Mechanisms". In *VLDB Workshop on Secure Data Management in a Connected World (SDM)* (Toronto, Canada, Aug. 2004).

[45] Luo, B., Lee, D., Lee, W.-C., and Liu, P. "QFilter: Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting". In *ACM CIKM* (Washington D.C., USA, Nov. 2004).

[46] Luo, B., Lee, D., Lee, W.-C., and Liu, P. "Deep Set Operators for XQuery".

[47] Malhotra, A., Melton, J., and Walsh, N. "XQuery 1.0 and XPath 2.0 Functions and Operators". W3C Working Draft, Feb. 2005. http://www.w3.org/TR/2005/WD-xpath-functions-20050211/.

[48] Mohan, S., Klinginsmith, J., Sengupta, A., and Wu, Y. Acxess - access control for xml with enhanced security specifications. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)* (Washington, DC, USA, 2006), IEEE Computer Society, p. 171.

[49] Mohan, S., Sengupta, A., and Wu, Y. Access control for xml: a dynamic query rewriting approach. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management* (New York, NY, USA, 2005), ACM Press, pp. 251–252.

[50] Mohan, S., and Wu, Y. Ipac: an interactive approach to access control for semi-structured data. In *VLDB'2006: Proceedings of the 32nd international conference on Very large data bases* (2006), VLDB Endowment, pp. 1147–1150.

[51] Muench, S., and Scardina (Eds), M. "XSLT Requirements Version 2.0". W3C Working Draft, Feb. 2001. http://www.w3.org/TR/xslt20req.

[52] Murata, M., Tozawa, A., and Kudo, M. "XML Access Control using Static Analysis". In *ACM Conf. on Computer and Communications Security (CCS)* (Washington D.C., 2003).

[53] Murata, M., Tozawa, A., Kudo, M., and Hada, S. Xml access control using static analysis. *ACM Trans. Inf. Syst. Secur. 9*, 3 (2006), 292–324.

[54] Murthy, R., Liu, Z. H., Krishnaprasad, M., Chandrasekar, S., Tran, A.-T., Sedlar, E., Florescu, D., Kotsovolos, S., Agarwal, N., Arora, V., and Krishnamurthy, V. Towards an enterprise XML architecture. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2005), ACM Press, pp. 953–957.

[55] Nicola, M., and van der Linden, B. Native XML support in DB2 universal database. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases* (2005), VLDB Endowment, pp. 1164–1174.

[56] Paparizos, S., Al-Khalifa, S., Chapman, A., Jagadish, H. V., Lakshmanan, L. V. S., Nierman, A., Patel, J. M., Srivastava, D., Wiwatwattana, N., Wu, Y., and Yu, C. Timber: a native system for querying xml. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2003), ACM Press, pp. 672–672.

[57] Qi, N., and Kudo, M. Access-condition-table-driven access control for xml databases. In *ESORICS* (2004), P. Samarati, P. Y. A. Ryan, D. Gollmann, and R. Molva, Eds., vol. 3193 of *Lecture Notes in Computer Science*, Springer, pp. 17–32.

[58] Qi, N., and Kudo, M. Xml access control with policy matching tree. In *ESORICS 2005, 10th European Symposium on Research in Computer Security* (2005), pp. 3–23.

[59] Qi, N., Kudo, M., Myllymaki, J., and Pirahesh, H. A function-based access control model for xml databases. In *CIKM '05: Proceedings of the 14th ACM international conference on Information and knowledge management* (New York, NY, USA, 2005), ACM Press, pp. 115–122.

[60] Rabitti, F., Bertino, E., Kim, W., and Woelk, D. "A Model of Authorization for Next-Generation Database Systems". *ACM Trans. on Database Systems (TODS) 16*, 1 (1991), 89–131.

[61] Rys, M. XML and relational database management systems: inside Microsoft SQL Server 2005. In *SIGMOD '05: Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2005), ACM Press, pp. 958–962.

[62] Sandhu, R., and Chen, F. "The Multilevel Relational (MLR) Data Model". *ACM Trans. on Information and System Security (TISSEC) 1*, 1 (1998).

[63] Sandhu, R., Coyne, E., Feinstein, H., and Youman, C. "Role-Based Access Control Models". *IEEE Computer 29*, 2 (1996).

[64] Schmidt, A. R., Waas, F., Kersten, M. L., Florescu, D., Manolescu, I., Carey, M. J., and Busse, R. "The XML Benchmark Project". Tech. Rep. INS-R0103, CWI, April 2001.

[65] Schoning, H. Tamino - a dbms designed for xml. In *IEEE ICDE* (Washington, DC, USA, 2001), pp. 149–154.

[66] Shanmugasundaram, J., Tufte, K., He, G., Zhang, C., DeWitt, D., and Naughton, J. "Relational Databases for Querying XML Documents: Limitations and Opportunities". In *VLDB* (Edinburgh, Scotland, Sep. 1999).

[67] Simeon, J., and Fernandez, M. "Galax V 0.3.5", Jan. 2004. http://db.bell-labs.com/galax/.

[68] Stoica, A., and Farkas, C. Secure xml views. In *DBSec* (2002), E. Gudes and S. Shenoi, Eds., vol. 256 of *IFIP Conference Proceedings*, Kluwer, pp. 133–146.

[69] Tan, K.-L., Lee, M. L., and Wang, Y. "Access Control of XML Documents in Relational Database Systems". In *Int'l Conf. on Internet Computing (IC)* (Las Vegas, NV, Jun. 2001).

[70] Winslett, M., Smith, K., and Qian, X. "Formal Query Languages for Secure Relational Databases". *ACM Trans. on Database Systems (TODS) 19*, 4 (1994), 626–662.

[71] Xiao, Y., Luo, B., and Lee, D. "Security-Conscious XML Indexing". In *Int'l Conf. on Database Systems for Advanced Applications (DASFAA)* (Bangkok, Thailand, 2007).

[72] Yoshikawa, M., Amagasa, T., Shimura, T., and Uemura, S. "XRel: A Path-Based Approach to Storage and Retrieval of XML Documents using Relational Databases". *ACM Trans. on Internet Technology (TOIT) 1*, 2 (Nov. 2001), 110–141.

[73] Yu, T., Srivastava, D., Lakshmanan, L. V., and Jagadish, H. V. "Compressed Accessibility Map: Efficient Access Control for XML". In *VLDB* (Hong Kong, China, Aug. 2002).

Vita

Bo Luo

Bo Luo was born in Chengdu, China in 1978. He received his B.S. degree in Electronic and Information Engineering from the University of Science and Technology of China in 2001, and his M.Phil. degree in Information Engineering from the Chinese University of Hong Kong in 2003. His current research focuses on XML and relational database systems, especially security and privacy issues. He is also interested in security and privacy issues in the context of Internet and information retrieval.