On Fine-Grained Access Control for XML

by

Donghui Zhuo

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2003

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

Donghui Zhuo

I understand that my thesis may be made electronically available to the public.

Donghui Zhuo

Abstract

Fine-grained access control for XML is about controlling access to XML documents at the granularity of individual elements or attributes. This thesis addresses two problems related to XML access controls. The first is efficient, secure evaluation of XPath expressions. We present a technique that secures path expressions by means of query modification, and we show that the query modification algorithm is correct under a language-independent semantics for secure query evaluation. The second problem is to provide a compact, yet useful, representation of the access matrix. Since determining a user's privilege directly from access control policies can be extremely inefficient, materializing the access matrix—the net effect of the access control policies—is a common approach to speed up the authorization decision making. The fine-grained nature of XML access controls, however, makes the space cost of matrix materialization a significant issue. We present a codebook-based technique that records access matrices compactly. Our experimental study shows that the codebook approach exhibits significant space savings over other storage schemes, such as the access control list and the compressed accessibility map. The solutions to the above two problems provide a foundation for the development of an efficient mechanism that enforces fine-grained access controls for XML databases in the cases of query access.

Acknowledgements

I would like to express my most sincere gratitude to my supervisor, Dr. Kenneth Salem, who is a truly nice gentleman with immense knowledge and endless patience. Without his insightful guidance and invaluable assistance, this thesis would not be possible. I feel extremely lucky to be under his supervision. I am very much grateful to my readers, Dr. Frank Tompa and Dr. Tamer Özsu, for their critical reviews and useful suggestions on this thesis. I am also grateful to Ting Yu of the University of Illinois, H.V. Jagadish of the University of Michigan, and Gary Promhouse of Open Text Corporation, for providing the experiment data. I am forever indebted to my parents, Tibang and Kuiqing, for their love and support. They are the persons who brought me up and encouraged me to pursue science. Special thanks go to my elder brother Lieguang, the best brother in the world, who has always been on my side since my childhood. Lastly and most importantly, I want to thank my wife, Patty Peng, for everything. This thesis is dedicated to you.

Trademarks

LiveLink is a trademark of Open Text Corporation.

Contents

1 Introduction **1**

1.1 Problems and Challenges 2

1.1.1 Secure Query Evaluation 2

1.1.2 Compact Representation of Access Matrix 3

1.2 Contributions 4

1.3 Organization 5

2 Preliminary Concepts **6**

2.1 XPath 6

2.2 Twig Query 12

2.3 Access Control Basics 13

2.3.1 Access Control Policies 14

2.3.2 Access Control Mechanisms 18

3 Secure Query Evaluation **23**

3.1 Models of XML Data and Access Control 24

3.2 Semantics of Secure Query Evaluation 27

3.2.1 Cho's semantics 28
3.2.2 View Based Semantics 29
3.3 Enforcement of Secure XPath Evaluation 32
3.3.1 Overview of the Query Modification Algorithm 33
3.3.2 Security Functions 35
3.3.3 Query Rewriting Function 41
3.4 Proof Overview 45

4 Compact Representation of Access Matrix 47
4.1 ACL, CL and AR 48
4.2 Compressed Accessibility Map 50
4.3 Codebook Based Scheme 52
4.3.1 Vector Based Scheme 53
4.3.2 Slab Based Scheme 56
4.4 Experiment 57
4.4.1 Hypothesis Verification 57
4.4.2 Frequency Distribution of Access Control Vectors 61
4.4.3 Performance Evaluation 64

5 Related Work 70
5.1 Models Proposed by Damiani and Bertino 71
5.2 XACL 73
5.3 Optimizing the Secure Evaluation of Twig Queries 74

6 Conclusions 76

6.1 Future Work 77

A XPath Query Modification Algorithm **78**

A.1 Definition of Query Rewriting Function 78

A.2 Definitions of Security Functions 80

B Correctness Proof of the XPath Query Modification Algorithm **88**

B.1 Objective and Assumptions 88

B.2 Notation 89

B.3 Proof Skeleton 89

B.4 Proof for Function Calls 93

B.5 Proof for Comparison Operators 102

B.6 Proof for Numeric Operators 104

B.7 Proof for Navigation Operators 105

B.8 Proof for Predicates 106

B.9 Proof for Logical Operations 106

Bibliography **108**

List of Tables

2.1 Access Matrix A 18

2.2 Access Cube A′ 20

3.1 Security Function List 36

3.2 The Mapping from Insecure Functions to Secure Functions 39

3.3 The Mapping from Insecure Operators to Secure Functions 41

4.1 Authorization Relation 50

4.2 Access Control Vector Analysis for Waterloo Data 59

4.3 Access Control Vector Analysis for LiveLink Data 60

List of Figures

2.1 Examples of Twig Queries 12

2.2 Directory Tree of the Toy File System 19

2.3 View Construction Mechanism vs Query Modification Mechanism . 21

3.1 An Example of an XML Document 26

3.2 Hierarchical Model of the XML Document 27

3.3 An (Invalid) Access Control Specification that Hides Mary's Information 27

3.4 A Valid Access Control Specification that Hides Mary's Information 32

3.5 Overview of the Query Modification Mechanism 34

3.6 A Valid Accessibility View 38

3.7 XPath Query Rewriting Function 42

4.1 Compression Accessibility Map (CAM) 52

4.2 Codebook Implementation Schemes 54

4.3 Access Control Vector Analysis for CAM Data 62

4.4 Frequency Distribution of Access Control Vectors in LiveLink Data 63

4.5 Total Space Cost Comparison between ACV and SLAB 67

4.6 In-Memory Space Cost Comparison between ACV and SLAB 68
4.7 Space Cost Comparison between ACL, CAM and Codebook 69

To Patty

Chapter 1

Introduction

The eXtensible Markup Language (XML) is a markup language promoted and standardized by the World Wide Web Consortium [31]. Largely because of its simplicity and powerful ability to describe information structure, in the past few years, XML has quickly grown to be the *de facto* standard of data representation and data exchange over the web. The amount of data encoded in XML format is increasing rapidly.

Given the sensitive nature of information, different XML documents or different portions of an XML document may require different levels of protection. Consider an XML document that contains the background and compensation information of all employees of one company. It is very likely that the background information is less secret and hence can be browsed by everyone in the company. The compensation information, on the other hand, may be more secret and can only be accessed by the employee and his/her managers. To protect information privacy, one security question must be answered—how to control the access to XML data.

1.1 Problems and Challenges

The problem of providing access controls for XML data has attracted considerable attention from both the security community and the database community in recent years. Much of the work on XML access controls to date, however, has been performed in the context of XML document management where the documents to be protected tend to be small and the access requests are often requests for browsing an entire document [3, 13, 2, 10, 8, 9]. Despite the importance of query access to XML, relatively little work has been done to enforce access controls for XML databases in the case of query access. Developing an efficient mechanism for XML databases to control query-based access is therefore the central theme of this thesis. In particular, we have focused our attention to two problems: the secure evaluation of XML queries and the compact representation of access matrices.

These two problems relate to two fundamental components of an access control mechanism: secure query evaluation is about the development of an effective and efficient *enforcement* mechanism that controls query-based access to XML databases, and the compact representation of an access matrix is about the development of an effective and efficient *decision making* mechanism for fine-grained XML access controls.

1.1.1 Secure Query Evaluation

Secure query evaluation concerns the evaluation of XML queries in the presence of access controls. It essentially requires the enforcement mechanism to guarantee that user queries only access, and return, the data items (in XML databases) that

the user is allowed to access.

An intuitive approach to secure an XML query is to evaluate the query first and then filter out inaccessible data items from the query result according to the access control policies. This approach, although attractive, is not secure. It guarantees that a user's query won't return unauthorized data; but it does not guarantee that the query won't *touch* (or check conditions on) unauthorized data during its evaluation. An alternative approach is to create a user's accessibility view first, as if by removing all of the inaccessible data items from the original XML document, and then evaluate the query against the user's accessibility view. This approach is secure and has been widely used to enforce access controls in the context of XML document management [3, 13, 2, 10, 8, 9]. Unfortunately, when the XML documents to be protected are large, this approach is not efficient, as generating a user's accessibility view is likely to be expensive. Therefore the first problem we are going to explore is to develop an efficient enforcement mechanism that guarantees the secure evaluation of XML queries.

1.1.2 Compact Representation of Access Matrix

A decision making mechanism is responsible for determining whether a user is authorized to access a data item in a given mode on the basis of the user's properties and the access control policies. Given an authorization inquiry, a decision making mechanism may derive the authorization decision directly from access control policies [18, 23, 3, 13, 2, 10, 8, 9]. However, when the policies are sophisticated, this approach can be extremely inefficient.

As an access matrix[1] captures the net effect of access control policies, one possible approach for fast authorization decision making is to materialize the access matrix. For example, we can maintain, for each data item in the XML database, an access control list recording the users who are authorized to access that data item. This approach works well in coarse-grained access controls, *e.g.*, in file systems and relational databases, where access controls are usually enforced at file level or relation level. In the context of fine-grained access controls, however, it exhibits significant space overhead. Consider an XML database of 1,000 users in which every data item (*e.g.*, every element) on average can be accessed by 10% of the users. Under this approach, each data item in the database, on average, would maintain a list of 100 users. If each user identifier occupies 2 bytes, the space cost would be 200 bytes per data item, which is obviously too much, and this space issue will further deteriorate if we allow access controls to be specified at the level of attributes. Finding a compact representation for the access matrix is therefore the second problem we are going to explore.

1.2 Contributions

This thesis makes two contributions. First, we present a technique that enforces the secure evaluation of XPath expressions by means of query modification, and prove that this query modification algorithm is correct under a language-independent semantics for secure query evaluation. Second, we describe and evaluate a codebook based scheme for the compact representation of access matrices. Our experimental

[1]The explanation of access matrix can be found in Section 2.3.2.

study reveals that the codebook scheme exhibits substantial space saving over other schemes, such as the access control list and the compressed accessibility map. In most cases, the space cost of the codebook scheme is less than 10% of that of the CAM scheme.

1.3 Organization

The remainder of the thesis is organized as follows. In Chapter 2, we review some preliminary concepts, including XPath, twig queries, and the access control systems. In Chapter 3, we first introduce a language-independent semantics for secure query evaluation. Then, on the basis of that semantics, we present a query modification algorithm for the secure evaluation of XPath expressions. An overview of the correctness proof of the query modification algorithm is provided at the end of the chapter. In Chapter 4, we propose a codebook scheme for the compact representation of access matrices, and compare its space efficiency with other schemes. We review some related work in Chapter 5 and conclude in Chapter 6.

Chapter 2

Preliminary Concepts

2.1 XPath

XPath[1], as a language for addressing parts of an XML document, is the basis of many XML languages, such as XSLT and XQuery [29]. It gets the name from the use of path notations for navigating through the hierarchical structure of an XML document.

XPath models an XML document as a tree of nodes. It defines seven types of node: root node, element node, text node, attribute node, namespace node, processing instruction node, and comment node. The primary syntactic construct in XPath is called an expression. An XPath expression is always evaluated within a context. The context, which is usually specified in the outside evaluation environment (*e.g.*, in XSLT or XQuery), has five elements:

[1]At the time of this writing, XPath version 2.0 is still a work in progress. Here we introduce the basic concepts of XPath version 1.0.

1. A node, *i.e.*, the context node
2. A pair of non-zero positive integers, *i.e.*, the context position and the context size
3. A set of variable bindings that contain the mapping from variable names to variable values
4. A function library that contains a mapping from function names to function definitions
5. A set of namespace declarations in scope for the expression that contains a mapping from prefixes to namespace URIs

In this thesis, we will use a 6-tuple ⟨context-node, context-position, context-size, variable-bindings, function-library, namespace-decl⟩ to denote a context. The result of an XPath expression is always an object of one of the following four basic types:

1. A node set (an unordered collection of nodes without duplicates)
2. A boolean value (either true or false)
3. A number (a floating-point number)
4. A string (a sequence of UCS characters)

For example, the expression

```
/descendant::name
```

is evaluated to be a node-set which contains all `name` elements descended from the document root. However, the expression

```
/descendant::employee[descendant::name="John"]
  /descendant::salary/child::textnode()
```

is evaluated to a string value, showing the salary of the employee whose name is `John`.

The most important type of XPath expression is the location path; it selects a set of nodes relative to the context node. A location path can be either a relative location path or an absolute location path. An absolute location path, which starts with a forward slash ("/"), is evaluated with respect to the root node of the document, whereas a relative location path, without the leading slash, is evaluated with respect to the current context node. The following are some valid location path expressions.

1. `child::employee` selects the `employee` elements which are children of the context node

2. `descendant::name` selects the `name` elements which are descendants of the context node

3. `descendant::contact/child::name` selects the `name` elements which are children of the `contact` elements which, in turn, are descendants of the context node

4. `descendant::employee[last()]` selects the last `employee` element which is a descendant of the context node

5. `/descendant::employee[position()=1]/descendant::salary` selects the `salary` element of the first employee

6. `/descendant::employee[descendant::salary>70000]/descendant::name [starts-with(child::textnode(), "J")]` selects the `name` elements of the employees whose salary is greater than 70,000 and whose name starts with a letter `J`

A location path consists of one or more location steps separated by "/". For example, the expression

```
descendant::employee/descendant::name
```

has two location steps: the `descendant::employee` and the `descendant::name`. A location path is evaluated from left to right, one step at a time. The initial step selects a set of nodes relative to a context node; each node in the result set generated by the initial step, then, is used as a context node for the second step. The union of the sets of nodes identified by the second step is the result of the composition of the first two steps. If there exists a third step, each node in the union will then be used as a context node for the evaluation of the third step, and so on, until all of the location steps are processed.

The syntax of a location step is `axisname::nodetest[predicate]*`. For example, in the expression

```
descendant::employee[last()]
```

`descendant` is the name of the axis, `employee` is the node test and `[last()]` is a predicate. The axis name specifies the tree relationship between the nodes to be selected and the context node. The node test specifies the node type and the expanded-name of the nodes to be selected. The predicates are expressions

used to further refine the node set selected by the axis and the node test. In our example, the initial node set selected by the axis name and the node test (*i.e.*, the `descendant::employee`) is a set containing all of the `employee` element nodes which are children of the context node. This initial node set will be further refined by predicate `[last()]`; the final result is therefore the set that contains the *last* `employee` element node only.

Location paths, which are used to select nodes from an XML document, are just special cases of XPath expressions. A general XPath expression is more powerful; it allows users to do operations on the node sets selected by location paths. The operations include function calls (*e.g.*, the function `last()`), logical operations (*e.g.*, `and`, `or`), numeric operations (*e.g.*, `+`,`-`), and so on. For instance, an expression that returns the employee names which starts with either a letter "J" or a letter "M", like

```
/descendant::employee
  /descendant::name[starts-with(child::textnode(), "J")] |
/descendant::employee
  /descendant::name[starts-with(child::textnode(), "M")]
```

involves several location paths, two function calls and one union operation. XPath 1.0 defines a core function library which must be supported by all XPath implementations. This core function library includes four categories of functions: node set functions, string functions, boolean functions and number functions. Besides that, XPath also permits users to extend the core function library by defining new functions.

In order to simplify the syntax of expressions, XPath 1.0 defines a set of syntactic abbreviations. One of the most important abbreviation is that the axis name "`child::`" can be omitted from a location step. For example, the expression

```
employee/name
```

is equivalent to the expression

```
child::employee/child::name
```

Other abbreviations include:

1. "@" is the abbreviation of "`attribute::`"
2. "." is the abbreviation of "`self::node()`"
3. ".." is the abbreviation of "`parent::node()`"
4. "//" is the abbreviation of "`/descendant-or-self::node()/`"

For example, the expression

```
//contact[../employee/@gender="male"]
```

is an abbreviation of the expression

```
/descendant-or-self::node()/child::contact
  [parent::node()/child::employee/attribute::gender="male"]
```

2.2 Twig Query

A twig query, or a tree pattern query, is a rooted node-labeled tree such that (1) its nodes are labeled by element tags or string values, (2) its edges are either single edges (representing parent-child relationship) or double edges (representing ancestor-descendant relationship), and (3) a subset of nodes is distinguished, usually with a star, indicating that they are in the query projection list [7].

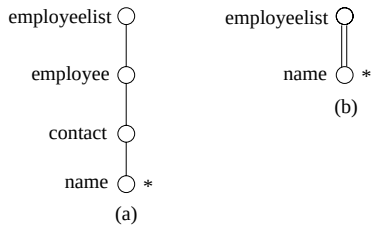

Figure 2.1: Examples of Twig Queries

Figure 2.1 shows two examples of twig queries. Twig query (a) can be expressed as a single XPath expression:

```
/child::employeelist/child::employee/child::contact/child::name
```

Twig query (b) can be expressed as

```
/child::employeelist/descendant::name
```

A twig query is answered by finding matchings. Suppose q is a twig query with nodes $(u_1, \ldots, u_n)$. A matching is a function that maps q's nodes to nodes in the database such that all node predicates are satisfied and, further, the structural relationships (*e.g.*, parent-child relationship or ancestor-descendant relationship) between nodes in q are satisfied. A matching, which binds each node u_i in the

query to a node x_i in the database, results in a binding tuple (x_1,...,x_n). The final answer to a twig query is produced by projecting out, from the binding tuples, the nodes that are not in the projection list.

Comparing with XPath expressions, twig queries are relatively simple, as they have only two axes, the `child` and the `descendant`, and they do not permit functions and operators.

2.3 Access Control Basics

The goal of database security is to protect the secrecy, integrity, and availability of data against unintentional or malicious threats.

Establishing a secure database is a complex task. Access control, which governs the *direct* access to databases, deals with only one of many crucial issues in database security. Besides access control, the other security issues that must be considered include authentication, auditing, and even the security control of the underlying operating system and network. While these issues are equally important for maintaining the security of a database, they are not in the scope of this thesis.

An access control system is logically composed of two parts: an access control policy that describes the security requirements and an access control mechanism that enforces the given policy. Ideally, the specification of access control policies should be distinct from the implementation of the access control mechanisms, in which case, an access control policy can be enforced by different mechanisms, and an access control mechanism can enforce multiple policies.

2.3.1 Access Control Policies

An access control policy is a high level guideline that specifies the security requirements of a database; it describes the access privileges of the users in the database, and states how the access privileges should be administered. In general, there are three types of access control policies: discretionary policies, mandatory policies and role based policies.

Discretionary Access Control Policy

Discretionary access control (DAC) policies govern access to objects[2] on the basis of user identities and authorizations [16]. A major property of discretionary policies is that the decisions of granting and revoking access privileges (on an object) are left to the discretion of individual users. In other words, in discretionary policies, a user with a certain access privilege (*e.g.*, read, write, or execute) is capable of passing that privilege on to other users.

A discretionary access control policy can be specified as a collection of authorizations, each of which, conceptually, is a $\langle subject, object, mode \rangle$ tuple, stating that *subject* is authorized to access *object* in access mode[3] *mode*. Since such an authorization always grants a privilege to a user, it is also called a positive authorization. A user's request to access an object is checked against the specified authorizations; if there exists an authorization stating that the user can access the object in the specific mode, the access is granted, otherwise it is denied.

[2]We use the term *object* to refer to an arbitrary unit of access control in a database.

[3]We use the term *access mode* to refer to a privilege, *e.g.*, read, write, or execute.

However, in a large database with thousands of users and millions of objects, specifying an access control policy in terms of positive authorizations in this manner may be tedious. To ease the task of policy specification, implicit authorizations and negative authorizations are introduced.

Implicit authorizations, in contrast to explicit authorizations, are not explicitly specified by the users and the security administrators; instead, they are derived from explicit authorizations, according to some pre-defined derivation rules. The derivation rules, usually defined by the security administrators, direct how to derive implicit authorizations from explicit authorizations. Suppose, for instance, there is an explicit authorization stating that group FACULTY has READ privilege on the REPORT document. This authorization may derive implicit authorizations for the members of the FACULTY group. As a result, Professor ROSS, who doesn't have explicit authorizations specified but is a member of group FACULTY, may be permitted to READ document REPORT.

Whereas positive authorizations always grant privileges, a negative authorization expresses the notion of denying a user's access to an object. The adoption of negative authorizations is largely due to the need to express exceptions. Consider an example in which we want to authorize all members of group FACULTY the privilege to READ document REPORT, except for Professor ROSS. If only positive authorizations are allowed, we have to specify one authorization for each member in group FACULTY, excluding ROSS. With negative and implicit authorizations, this can be expressed by only two authorizations: a positive authorization for group FACULTY and a negative authorization for Professor ROSS.

The coexistence of positive and negative authorizations, however, may introduce conflicts. In our previous example, ROSS, who has an explicit negative authorization on document REPORT, also has an implicit positive authorization as being a member of the FACULTY. In this case some conflict resolution rules must be specified. In short, the effect of implicit authorizations and negative authorizations is twofold. On one hand, it largely simplifies the task of policy specification. On the other hand, it greatly complicates the authorization decision making.

Although discretionary access control policies are flexible and widely used in commercial environments, they have one inherent security flaw. They are unable to secure the data flow in a system. This flaw makes the system vulnerable to Trojan horse attacks. Systems that require data flow controls usually use mandatory access control policies.

Mandatory Access Control Policy

Mandatory access control (MAC) policies govern access to objects based on the sensitivity of the information contained in the objects and the trustworthiness of users. In MAC, every object is assigned a security level, called classification, which reflects the sensitivity of the information contained in the object; every user is assigned a security level, called clearance, which reflects the trustworthiness of the user. Security levels are usually elements of a partially ordered set. For example, a common set of security levels for military system is {TOPSECRET, SECRET, CONFIDENTIAL, UNCLASSIFIED}, where TOPSECRET is the most secure level and UNCLASSIFIED is the least secure level.

Most mandatory policies impose the following two restrictions: (1) A subject is

authorized to read an object only if the clearance of the subject is no less than the classification of the object, and (2) a subject is authorized to write an object only if its clearance is no higher than the object's classification. Enforcement of these two principles ensures that the information can only flow within the same security level or upward to higher security levels, and, thus, can never leak to lower levels. Mandatory access control policies and discretionary access control policies can be used together to strengthen access controls.

Role Based Access Control Policy

Role based access control (RBAC) policies are supplements to the discretionary policies and the mandatory policies. In RBAC, users are assigned roles based on their competencies and responsibilities in the organization. The process of defining roles is usually based on a thorough analysis of how an organization operates. The operations the user is permitted to perform and the objects the user is authorized to access are determined by the active roles the user assumes. Most systems allow a single user to adopt different roles on different occasions. Also the same role can be played by several users. Although the concept of role in RBAC resembles the group in discretionary access controls, they are two different concepts. In discretionary access controls, the allocation of access privileges to groups is usually determined at the discretion of individual users, whereas in RBAC, the allocation of access privileges to roles, as well as determination of membership in a role, are determined by the security administrator.

2.3.2 Access Control Mechanisms

An access control mechanism enforces a given access control policy by ensuring that all direct access to the database is in accordance with that policy. An access control mechanism has two basic functions: decision making and decision enforcement.

Decision Making

Decision making is a process of deriving access control decisions on the basis of the user's identity and the access control policies. Regardless how an access control policy is specified, the net effect of a policy can be captured by a matrix, namely the access matrix. The rows of the matrix represent users, and the columns represent objects. A matrix entry, say $A[s, o]$, contains the access modes in which user s is authorized to access object o. Table 2.1 shows an access matrix for a toy file system whose structure is shown in Figure 2.2. From the access matrix, we can tell that user S1 has READ and WRITE privileges on object O1, and user S3 has only READ privileges on object O2, O4, and O5.

	O1	O2	O3	O4	O5	O6
S1	R W					
S2		R W		R W	R W	
S3		R		R	R	
S4	R		R W			W

Table 2.1: Access Matrix A

A common variation of the access matrix is the access cube whose three dimensions are user, object, and access mode, respectively. The entries of an access cube are boolean values, either 1 or 0. An access cube entry, say $A[s, o, m]$, is 1 if

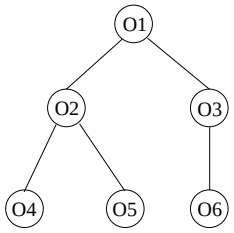

Figure 2.2: Directory Tree of the Toy File System

and only if user s is authorized to access object o in access mode m. The access matrix and the access cube are two different representations of the access control information or, equivalently, the net effect of the access control policies.

Suppose A' is the access cube corresponding to the access matrix A in Table 2.1. Table 2.2 shows a planar expression of the access cube A'. Each table in Table 2.2 represents a slice of A': Table (a) represents the slice of access mode READ, and table (b) represents the slice of access mode WRITE. Access cube A' tells us that user S1 is authorized to READ object O1, as the entry A'[S1, O1, READ] is 1; and user S1 is not authorized to WRITE object O2, as the entry A'[S1, O2, WRITE] is 0.

A spectrum of techniques can be used to implement decision making mechanisms. At one end of the spectrum, the implementation does not materialize the access matrix. Authorization decisions are always computed directly from access control policies upon request. This approach is flexible in that it allows the security administrators to change the access control policies at run time and the changes become effective immediately after commitment. However, when the access control policy is complex, this approach may be inefficient. At the other end of the spectrum, the implementation always materializes the access matrix. Consequently,

	o1	o2	o3	o4	o5	o6
s1	1	0	0	0	0	0
s2	0	1	0	1	1	0
s3	0	1	0	1	1	0
s4	1	0	1	0	0	0

(a) The READ Slice

	o1	o2	o3	o4	o5	o6
s1	1	0	0	0	0	0
s2	0	1	0	1	1	0
s3	0	0	0	0	0	0
s4	0	0	1	0	0	1

(b) The WRITE Slice

Table 2.2: Access Cube A′

upon receiving an authorization inquiry, the decision making mechanism can locate the answer in the materialized access matrix without computing it from scratch. The trade-off between these two approaches is in the decision computation time versus the decision search time. If we can somehow materialize the access matrix in a data structure that takes little space and permits fast lookups, the second approach may become a promising solution to the speed problem of the first approach. This is the problem that we are going to explore in Chapter 4.

Decision Enforcement

Decision enforcement is a process that guarantees that a user is always granted (or denied) access to an object if the authorization decision is affirmative (or negative). In the context of XML access controls, two techniques are commonly used to control

query based access: the view construction approach and the query modification approach.

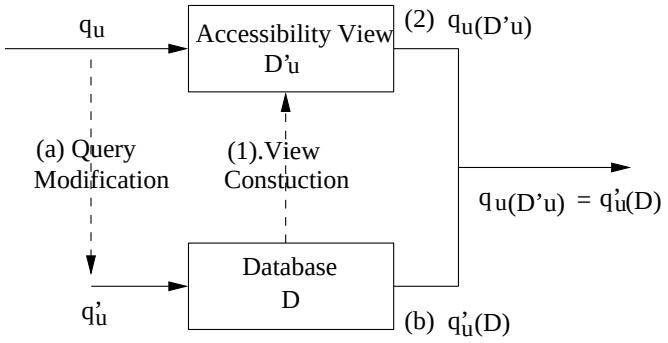

Figure 2.3: View Construction Mechanism vs Query Modification Mechanism

The basic concepts of these two approaches are illustrated in Figure 2.3, in which D represents the base database and q_u represents a query posted by user u. Upon receiving a query q_u, a view construction mechanism performs the following two steps[4]: (1) create user u's accessibility view D'_u. Conceptually, D'_u is created by filtering out all of the objects (from D) which u is not authorized to access; (2) evaluate query q_u against D'_u. The view construction approach is widely used to secure small XML documents [10, 8, 13]. However, when D is large, this approach may become inefficient, as generating a user's accessibility view is likely to be expensive.

An alternative approach is query modification. As shown in Figure 2.3, upon receiving query q_u, the query modification mechanism will[5] (a) rewrite query q_u to query q'_u, and (b) evaluate the modified query q'_u against the base database D.

[4]Refers to the steps marked as (1) and (2) in Figure 2.3.

[5]Refers to the steps marked as (a) and (b) in Figure 2.3.

The query modification algorithm should guarantee that the result of evaluating q'_u against D is equal to the result of evaluating q against D', *i.e.*, $q'_u(D)$ should be equal to $q_u(D'_u)$.

The query modification approach exhibits a few advantages over the view construction approach. First, it avoids the expensive computation of the accessibility view. Second, it does not require changes to the query evaluator. Third, since the modified queries are queries against the base database D, this approach allows the query evaluator to utilize the existing indexes maintained on D. The downside of the query modification approach is that the query modification algorithm is language dependent, *i.e.*, different query languages require different modification algorithms.

Chapter 3

Secure Query Evaluation

Secure query evaluation refers to the evaluation of queries with respect to the issuer's privileges, or, in other words, the evaluation of queries in the presence of access controls.

With the increasing popularity of XML, managing XML documents using conventional file management techniques is not sufficient; users require more flexible query approaches and more reliable storage management. Driven by this demand, research on XML data management has attracted considerable attention from the database community in recent years. Many XML databases have been developed, and many XML query languages have been proposed. A security problem brought on by the emergence of XML databases is how to protect the XML data in case of query access, or, how to evaluate XML queries securely so that sensitive information won't leak out to unauthorized users through query evaluation.

In this chapter we consider the secure evaluation of XPath expressions. We have chosen XPath expressions as our focus because they are the means by which

major XML query languages, such as XSLT and XQuery, address parts of XML documents [32, 30]. Specifically, we consider the path expressions defined in XPath specification version 1.0, because, at the time of this writing, XPath 2.0 is still a work in progress and not completely defined yet [29].

This chapter is organized as follows. In Section 3.1, we introduce the data model and the security model that will be used in this chapter. In Section 3.2, we define the semantics of secure query evaluation. Based on that semantics, we present a technique that secures path expressions by means of query modification, in Section 3.3. Finally, in Section 3.4, we give a brief overview of the correctness proof of the query modification algorithm.

3.1 Models of XML Data and Access Control

An XML database contains a collection of XML documents. Every XML document has a logical hierarchical structure. A collection of XML documents, when they are organized hierarchically, also has a hierarchical structure, just like files in Unix can be organized hierarchically in a directory tree. Based on this observation, we model the data in an XML database as an ordered tree, namely an XML database tree. The nodes represent various types of objects in the database, *e.g.*, elements or attributes, and the arcs represent various types of relationships, *e.g.*, element-subelement relationships or element-attribute relationships. Our model is a simplification of the data model adopted by XPath 1.0. We do not distinguish data type. No matter what type the object is, it is uniformly represented by a node in the database tree. Moreover, we do not care whether an XML database

tree represents one single XML document or a collection of XML documents. As far as what we are concerned, it represents the data in an XML database at the finest granularity. Hereinafter, the term *document* and *database* will be used interchangeably to refer to XML data.

Figure 3.1 shows an example of an XML document whose corresponding hierarchical model is shown in Figure 3.2. The number in each node is the node identifier. This document records the contact and payroll information of the employees in one company. Every `employee` element contains one `contact` element and one `payroll` element. The `contact` element records the employee's name and postcode, and the `payroll` element records the employee's salary and bonus. In addition, every `employee` element has one attribute indicating the gender of the employee. This is the document we are going to use in the following sections to explain the query modification algorithm.

Regarding access controls, as we have introduced in Chapter 2, the net effect of an access control policy can be captured by an access cube, say A. Assuming that U denotes the set of users, M denotes the set of access modes and O denotes the set of objects in database D, the access cube A is a $|U| \times |O| \times |M|$ 3-dimensional cube that uniquely determines which user can access which object in a given access mode.

Restricted to a specific user u and access mode m, the access cube A reduces to a 0-1 vector in which every bit corresponds to an object in D. This vector can be viewed as a labeling that assigns every node in the database tree a boolean tag, either 1 or 0, indicating whether this node is accessible with respect to user u and

Figure 3.1 An Example of an XML Document

```
<employeelist>
  <employee gender="male">
    <contact>
      <name> John </name>
      <postcode> N4W2H8 </postcode>
    </contact>
    <payroll>
      <salary> 75000 </salary>
      <bonus> 20000 </bonus>
    </payroll>
  </employee>
  <employee gender="female">
    <contact>
      <name> Mary </name>
      <postcode> M3R5H3 </postcode>
    </contact>
    <payroll>
      <salary> 85000 </salary>
      <bonus> 20000 </bonus>
    </payroll>
  </employee>
</employeelist>
```

access mode m. If we say that an access cube captures the effect of an access control policy, a labeled database tree, then, captures the effect of an access control policy for a specific user and access mode.

We assume that the *read* mode, among many other access modes, is used to control query access. Figure 3.3, for instance, shows a labeled database tree for a user (perhaps the user John) who is not permitted to *read* Mary's information, except for her name.

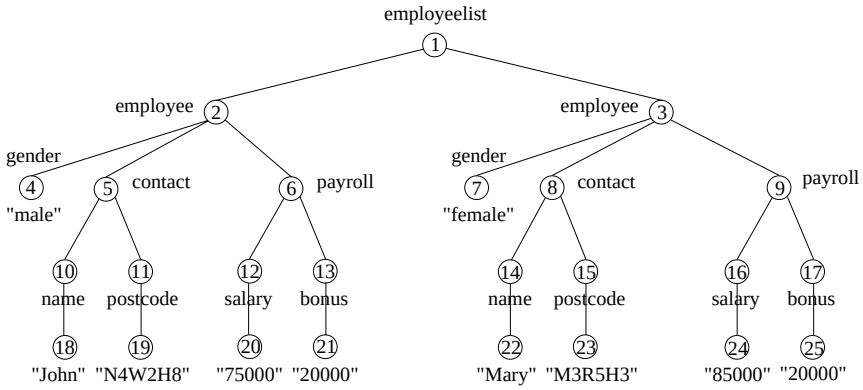

Figure 3.2: Hierarchical Model of the XML Document

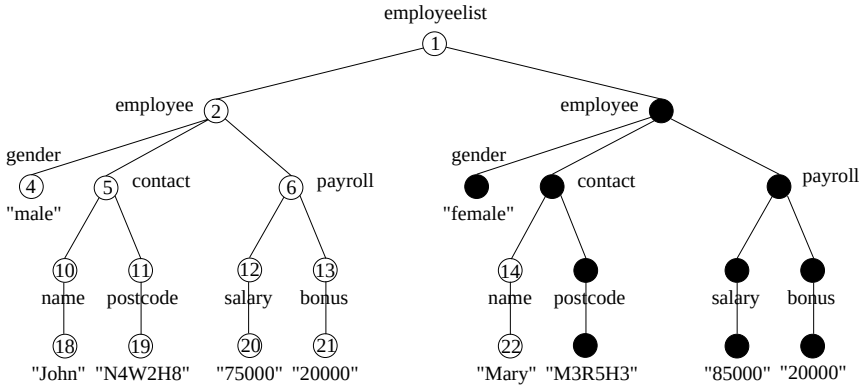

Figure 3.3: An (Invalid) Access Control Specification that Hides Mary's Information

3.2 Semantics of Secure Query Evaluation

Before we are able to consider how to evaluate queries securely, we must first understand what the correct result of a query should be, when it is evaluated in the presence of access controls.

Intuitively, we may say that a query is securely evaluated if the query uses and returns only the data that the user is authorized to access. However, this statement is too vague to be a definition—it does not define the verb "use" clearly. Consider

the following example. Suppose the labeled database tree D_{ur} in Figure 3.3 reflects user u's *read* rights in database D, and user u wishes to evaluate expression

```
/descendant::name/child::textnode()
```

against database D to retrieve the string values of the names of all employees. What should the safe answer to the query be? It is agreeable that the string `John` should be in the result set, as the node itself is accessible and the evaluation (presumably) does not use any inaccessible node. But what about the string `Mary`? Should it be in the result set? In other words, do we *use* the inaccessible nodes 3 and 8, when we navigate from node 1 to node 14? This question is unanswerable without further clarification.

3.2.1 Cho's semantics

SungRan Cho and his colleagues introduced a semantics for the secure evaluation of twig queries [7]. According to Cho's semantics, given a database D with access controls and a twig query q_u with nodes $(u_1, \ldots, u_n)$ posed by user u, the safe answer to query q_u is determined by the following procedure:

1. Find out the set of binding tuples, say T. A binding tuple $t \in T$ is of the form $(x_1, \ldots, x_n)$ where x_i is a node in D.

2. Generate, from set T, the set of safe binding tuples, say T_s. A binding tuple $t \in T$ is safe if, and only if, every component x_i is accessible with respect to user u.

3. Generate the safe answer to query q_u by projecting set T_s onto the set of nodes appearing in the projection list.

Suppose, for example, user u wishes to evaluate the twig query (a), shown in Figure 2.1, against the labeled XML tree in Figure 3.3. This query will find two binding tuples in database, $\langle 1, 2, 5, 10\rangle$ and $\langle 1, 3, 8, 14\rangle$, in which the second one is unsafe, as it contains the inaccessible nodes 3 and 8. Therefore, the safe answer to query (a) is the first `name` element (node 10), created by projecting out, from the safe binding tuples, the nodes not appearing in the projection list. Similarly, twig query (b) will find two binding tuples: $\langle 1, 10\rangle$ and $\langle 1, 14\rangle$. Since both of tuples are safe, the safe answer to query (b) is a node set which contains both `name` elements.

This semantics, although clear, has one weakness: it is language dependent. Since Cho's semantics is defined in terms of bindings, a technique specific to the evaluation of twig queries, it is difficult to apply it to other query languages whose evaluations don't rely on binding, like XPath or XQuery. An XML database, however, is supposed to be queried by various kinds of query languages, not just twig queries. Ideally, the semantics of secure query evaluation should be defined independently of the query language.

3.2.2 View Based Semantics

To settle the problem of Cho's semantics, we define the semantics of secure query evaluation in terms of a language independent concept: the accessibility view. An accessibility view, as defined in Definition 1, is always associated with a user and an access mode; it is a view (of the original database) that contains only the data

that the specific user is authorized to access in the given mode. Since this chapter concerns the security controls for query access, we are especially interested in a user's *read* accessibility view, *i.e.*, the accessibility view in *read* mode.

Definition 1 (Accessibility View) *Let $D = \{V, E\}$ be an XML database tree where V is the set of nodes and E is the set of edges, A be an access cube specified on D, u be a user and m be an access mode. The accessibility view of user u on database D with respect to access mode m is $D'_{um} = \{V', E'\}$, where $V' = \{o \mid o \in V \wedge A[u, o, m] = 1\}$ and $E' = \{(p, q) \mid (p, q) \in E \wedge p \in V' \wedge q \in V'\}$.*

Given a database D with access controls and a query q_u posted by user u, a natural idea is that the correct result of query q_u should be determined by the following procedure:

1. Create user u's *read* accessibility view D'_{ur}.

2. Evaluate query q_u against D'_{ur}. The result of evaluating query q_u against D'_{ur} is defined as the correct result of the secure evaluation of query q_u.

One problem with this natural idea is that a user's read accessibility view may not be a tree. For example, the read accessibility view implied by the labeled XML tree in Figure 3.3 is not a tree, but a forest. From our standing, this view is *invalid*, as we are not able to evaluate queries against it. To avoid the problem, we refine the natural idea and define the semantics of secure query evaluation on a user's *valid* read accessibility view, a relatively narrow concept, as shown in the following two definitions.

Definition 2 (Valid Accessibility View) *Let D be an XML database tree, D'_{ur} be user u's read accessibility view. The view D'_{ur} is valid if, and only if, D'_{ur} is a tree whose root is the same as the root of D.*

Definition 3 (Secure Query Evaluation) *Let D be an XML database tree, q_u be a query against D posed by user u, and D'_{ur} be user u's valid read accessibility view. The correct result of secure evaluation of query q_u is defined to be the result of evaluating query q_u against user u's valid read accessibility view, D'_{ur}.*

A user's read accessibility view is determined by the access control policy. In general, an access control policy may generate valid read accessibility views, or invalid read accessibility views, or both. Our semantics, however, requires that the access control policy that is to be enforced must derive valid read accessibility views only. In other words, if we classify access control policies into two categories—valid access control policies and invalid access control policies—based on Definition 4, only valid access control policies can be enforced by our semantics.

Definition 4 (Valid Access Control Policy) *Let U be a set of users, $D = \{V, E\}$ be an XML database tree where V is a set of nodes and E is a set of edges, P be an access control policy whose net effect is captured by an access cube A, and r be the read mode. Security policy P is valid if, and only if, it does not derive invalid read accessibility views, i.e.,* $\forall p \in U, \forall q \in V : A[p, q, r] = 0 \rightarrow \neg\exists t \in V$ s.t. $(A[p, t, r] = 1 \wedge\ t$ is a descendant of $q)$.

Defining the semantics of secure query evaluation on the basis of valid accessibility views sounds like a considerable restriction. However, it is actually not,

provided that access controls can be specified at the granularity of individual attributes. That is because we can always relax an invalid access control policy a little bid to make it valid. For example, the invalid access controls illustrated in Figure 3.3 can be relaxed by making the nodes 3 and 8 readable, as shown in Figure 3.4. These two access controls are not the same. For example, a user who wants to count the number of employees may find two employees in the view of Figure 3.3, but only one employee in the view of Figure 3.4. Nonetheless, the valid access controls in Figure 3.4 also successfully hide the contact and payroll information of Mary, except for the existence of the `employee` element and the `contact` element.

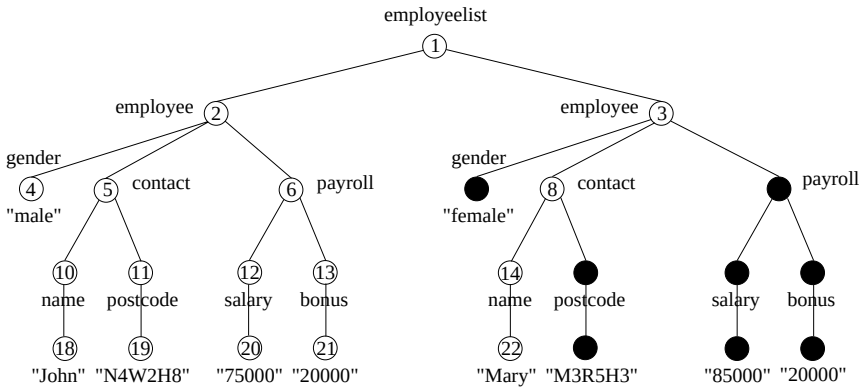

Figure 3.4: A Valid Access Control Specification that Hides Mary's Information

3.3 Enforcement of Secure XPath Evaluation

We have defined the semantics of secure query evaluation in the previous section. In this section we consider the enforcement of secure query evaluation, in particular the enforcement of the secure evaluation of XPath expressions. Assuming that D is

an XML database tree, P is a valid access control policy specified on D whose effect is captured by the access cube A, and q_u is an arbitrary XPath expression posted by user u, our objective is to develop an enforcement mechanism that guarantees the secure evaluation of q_u, with respect to A. Hereinafter, we refer to the user u as the *context* user.

As discussed in Chapter 2, there are two basic approaches for access control enforcement: the view construction approach and the query modification approach. Because, as was explained earlier, the view construction approach is likely to be expensive, in this section, we describe an approach that enforces the secure evaluation of path expressions by means of query modification. Comparing with the view construction approach, the query modification approach has the following three advantages:

1. It avoids the expensive materialization of accessibility views.

2. Since the modified queries are still XPath expressions, they can be passed to standard XPath evaluators for processing. No changes to an XPath evaluator is required.

3. Since the modified queries are queries against the original database D, the query evaluator may utilize the existing indexes maintained on D.

3.3.1 Overview of the Query Modification Algorithm

A correct query modification algorithm is the key to the success of the query modification approach. Appendix A shows a complete description of the XPath query

modification algorithm. The algorithm is composed of two parts: a query rewriting function and a collection of security functions. The query rewriting function is the heart of the modification algorithm—it defines how a query should be modified; the security functions, on the other hand, are supporting functions defined in the query modification algorithm to perform security-related activities. An implementation of the query modification algorithm should include both parts.

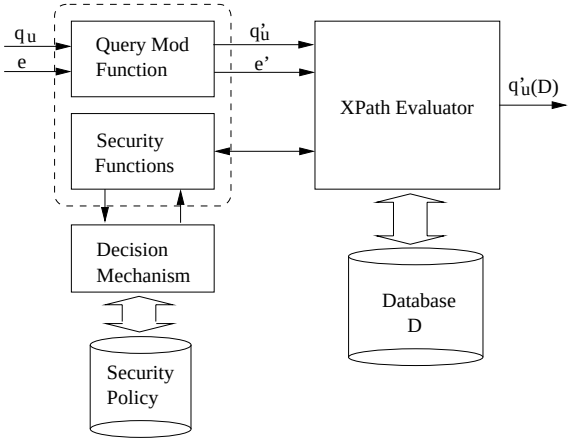

Figure 3.5: Overview of the Query Modification Mechanism

Figure 3.5 shows an overview of the query modification mechanism and its interactions with other components. The two blocks surrounded by the dashed lines are together referred to as the query modification mechanism. One is the implementation of the query rewriting function; the other is the implementation of the security functions. Given an XPath expression q_u and its evaluating context e, the query rewriting function transforms them into q'_u and e' respectively, and passes them to a standard XPath evaluator.[1] The XPath evaluator evaluates the

[1]Some readers may wonder why the query rewriting function modify the evaluating context e? This question will be addressed later when we discuss the context modification.

rewritten expression q'_u, under the context e', against the original database D, and returns the result. It is the responsibility of the query modification algorithm to guarantee the correctness of the result, *i.e.*, to guarantee that $q'_u(D)$, the result of evaluating q'_u against D, be equal to the result of evaluating q_u against user u's read accessibility view, $q_u(D'_{ur})$.

For example, given an XPath expression q_u that returns the string values of the names of all employees, like

```
/descendant::employee
  /descendant::name
  /child::textnode()
```

the query rewriting function will rewrite it, for user u, into q'_u as

```
/descendant::employee[sec-inview()]
  /descendant::name[sec-inview()]
  /child::textnode()[sec-inview()]
```

where the function `sec-inview()` is a security function for testing a user's access rights. We will introduce the security functions and the query rewriting function in detail in the following two subsections.

3.3.2 Security Functions

Security functions are supporting functions defined in the query modification algorithm to perform security-related activities. Table 3.1 shows a complete list of security functions. Their definitions are provided in the Section A.2 of Appendix

Num	Security Functions
1	`boolean sec-inview()`
2	`string sec-string(object?)`
3	`string sec-string-value(node-set)`
4	`node-set sec-id(object)`
5	`number sec-number(object?)`
6	`number sec-sum(node-set)`
7	`number sec-string-length(string?)`
8	`string sec-normalize-space(string?)`
9	`boolean sec-eq(object, object)`
10	`boolean sec-ne(object, object)`
11	`boolean sec-le(object, object)`
12	`boolean sec-lt(object, object)`
13	`boolean sec-ge(object, object)`
14	`boolean sec-gt(object, object)`
15	`number sec-addition(object, object)`
16	`number sec-subtraction(object, object)`
17	`number sec-multiply(object, object)`
18	`number sec-div(object, object)`
19	`number sec-mod(object, object)`

Table 3.1: Security Function List

A. In general, it is easy to tell security functions from XPath standard functions,

as the names of security functions are always prefixed by "sec-".

We need security functions for two reasons. First, security functions provide a means by which the XPath evaluator can communicate with the underlying decision making mechanism. For example, `sec-inview()`, is a function that the XPath evaluator can use to check the access privileges for a user. Specifically, `sec-inview()` returns TRUE if, and only if, the *context* user is authorized to read the *context* node under the access control policy. In other words, it returns TRUE if, and only if, $A[u, c_n, r] = 1$, where u is the context user, c_n is the context node and r is the read mode.[2]

Second, security functions are used to replace the insecure functions and operators. XPath 1.0 defines a set of standard functions and operators, *e.g.*, the function `string()`, the function `id()`, the operator `=`, the operator `+`, and so on. Some of them are insecure. This can be demonstrated by the following two examples.

Consider the standard `string()` function first. Suppose the tree in Figure 3.2 represents the database D, the labeled tree D_{ur} in Figure 3.4 represents user u's read privileges on D, and the tree in Figure 3.6 represents user u's accessibility view D'_{ur}, derived from D_{ur}. Suppose user u wishes to evaluate a path expression

```
string(/child::employeelist)
```

against D to retrieve the string value of the `employeelist` element. According to the XPath specification, the standard `string()` function, when receiving a node set as an input argument, will return the concatenation of the string-values of all

[2]The context user and context node are always available in the evaluation context. We will explain this when we introduce the context modification.

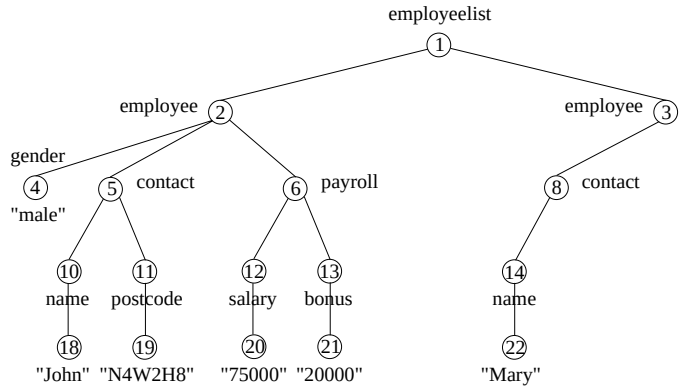

Figure 3.6: A Valid Accessibility View

text node descendants of the first node in the given node set. Therefore, the result of evaluating the above expression against D is the string

‘‘`JohnN4W2H87500020000MaryM3R5H38500020000`’’

According to our semantics of secure query evaluation, this result is insecure, as it is not equal to the result of evaluating the same query against the accessibility view D'_{ur}, *i.e.*, the string “`JohnN4W2H87500020000Mary`”.

To prevent information from leaking through the insecure `string()` function, we developed a secure version, the `sec-string()`, for the insecure `string()` function. The semantics[3] of `sec-string()` is very much like that of the standard `string()`, except: when receiving a node set as the input argument, `sec-string()` will return the concatenation of string-values of all *accessible* text node descendants of the first node in the given node set. Consequently, a secure rewriting of the previous expression is

```
sec-string(/child::employeelist)
```

[3]The formal definition of `sec-string()` can be found in Appendix A.2.

Besides the `string()` function, there are several other XPath standard functions that are also insecure. Table 3.2 shows a list of all insecure XPath standard functions, which must be replaced by the query modification algorithm, and their corresponding secure functions.

Insecure Functions	Corresponding Secure Functions
`node-set id(object)`	`node-set sec-id(object)`
`string string(object?)`	`string sec-string(object?)`
`number string-length(string?)`	`number sec-string-length(string?)`
`string normalize-space(string?)`	`string sec-normalize-space(string?)`
`number number(object?)`	`number sec-number(object?)`
`number sum(node-set)`	`number sec-sum(node-set)`

Table 3.2: The Mapping from Insecure Functions to Secure Functions

Now consider an example of an insecure XPath operator. Suppose user u wishes to evaluate an expression q_u

```
/child::employeelist="JohnN4W2H87500020000MaryM3R5H38500020000"
```

against D to check if the string-value of the element `employeelist` equals that given string. According to the XPath specification, if one operand of the operator "=" is a node-set (*e.g.*, `/child::employeelist`) and the other is a string, the result of the comparison is TRUE if, and only if, there is a node in the node-set such that the string-value of that node is equal to the argument string; the string-value of a

node is computed as if by a call to the standard `string()` function. The previous example has shown that the string value of element `employeelist` in D is

‘‘`JohnN4W2H87500020000MaryM3R5H38500020000`’’

Therefore, the result of evaluating q_u against D is TRUE. However, according to our semantics, this result is insecure, as it is not equal to the result of evaluating q_u against D'_{ur}. The essential reason behind the insecurity of the operator “=” is that the semantics of the standard “=” operator is defined on the standard `string()` function, and the standard `string()` function, as we have shown, is insecure.

To prevent information from leaking through insecure XPath operators, we developed a secure function, the `sec-eq()`, for the insecure operator “=”. The semantics of `sec-eq()` is very much like that of the standard comparison operator “=”. But, unlike the standard comparison operator which is defined on the standard `string()` function, `sec-eq()` is defined on the secure `sec-string()` function. As a result, a secure rewriting of q_u is

```
sec-eq(/child::employeelist,
      "JohnN4W2H87500020000MaryM3R5H38500020000")
```

Table 3.3 shows a complete list of insecure XPath operators, which must be replaced by the query modification algorithm, and their corresponding secure functions.

Insecure Operators	Corresponding Secure Functions
=	boolean sec-eq(object, object)
!=	boolean sec-ne(object, object)
<=	boolean sec-le(object, object)
<	boolean sec-lt(object, object)
>=	boolean sec-ge(object, object)
>	boolean sec-gt(object, object)
+	number sec-addition(object, object)
-	number sec-subtraction(object, object)
*	number sec-multiply(object, object)
div	number sec-div(object, object)
mod	number sec-mod(object, object)

Table 3.3: The Mapping from Insecure Operators to Secure Functions

3.3.3 Query Rewriting Function

The pseudocode in Figure 3.7 summarizes the XPath query rewriting function[4]. The query rewriting function has two parts: the query rewriting part and the context rewriting part. It can be conceived as a function that takes two arguments—an XPath expression q_u and its initial evaluation context e—and returns the rewritten expression q'_u and the rewritten context e'. The query rewriting function assumes that expression q_u is in the verbose syntax.[5]

Query Modification Part

Given an XPath expression q_u posted by user u, the query rewriting function will rewrite it in the following three steps:

[4]The formal definition of the XPath query rewriting function is described in Section A.1.

[5]In order to keep the query rewriting function simple, we assume that the incoming XPath expression is in the verbose syntax. However, it is easy to adapt the query rewriting function to handle expressions in abbreviated syntax.

Figure 3.7 XPath Query Rewriting Function

Query-Rewriting(q_u, e)
Begin
// 1. Query Rewriting
$q'_u \leftarrow q_u$
// a. Insert security check predicates
insert predicate `[sec-inview()]` after each occurrence
of clause `Axis::NodeTest` in q'_u
// b. Replace insecure functions
replace every occurrence of insecure function in q'_u
with its equivalent secure function.
// c. Replace insecure operators
replace every occurrence of insecure operator in q'_u
with its equivalent secure function.
// 2. Context Rewriting
e'.context-node $\leftarrow e$.context-node
e'.context-position $\leftarrow e$.context-position
e'.context-size $\leftarrow e$.context-size
e'.namespace $\leftarrow e$.namespace
e'.variable-binding $\leftarrow e$.variable-binding $\cup$ {(u, context-user)}
e'.function-library $\leftarrow e$.function-library $\cup$ { `sec-inview`, `sec-id`, `sec-string`, `sec-sum`, `sec-string-length`, `sec-normalize-space`, `sec-numeric-add`, `sec-numeric-subtract`, `sec-numeric-multiply`, `sec-numeric-divide`, `sec-numeric-integer-divide`, `sec-numeric-mod`, `sec-numeric-unary-plus`, `sec-numeric-unary-minus`, `sec-eq`, `sec-neq`, `sec-lt`, `sec-number`, `sec-gt`, `sec-le`, `sec-ge`, `sec-string-value` }
return q'_u, e'
End

1. Locate every occurrence of clause `Axis::NodeTest` in q_u, and insert a predicate `[sec-inview()]` immediately after each occurrence.

2. Replace every occurrence of insecure XPath functions in query q_u with its corresponding secure function, according to Table 3.2.

3. Replace every occurrence of insecure XPath operators in the query q_u with its corresponding secure function, according to Table 3.3.

For example, given a query q_u that returns the postcodes of all employees, like

```
/descendant::employee/descendant::postcode
```

the query rewriting function will rewrite it to q'_u as

```
/descendant::employee[sec-inview()]
  /descendant::postcode[sec-inview()]
```

Similarly, a more complicated expression that returns the name of male employees whose total income is greater than 100,000, like

```
/descendant::employee
  [attribute::gender="male"]
  [descendant::salary + descendant::bonus >= 100000]
  /descendant::name
```

would be rewritten[6] as

```
/descendant::employee[sec-inview()]
  [sec-eq(attribute::gender[sec-inview()], "male")]
  [sec-ge(sec-addition(descendant::salary[sec-inview()],
          descendant::bonus[sec-inview()]), 10000)]
  /client::name[sec-inview()]
```

[6]Since we require that the access control policy to be enforced must be valid, some of the inserted predicates are actually unnecessary. We leave the optimization of rewritten queries for future study.

Context Rewriting Part

Recall that a path expression is always evaluated within a context, and a context can be modeled as a 6-tuple like ⟨context-node, context-position, context-size, variable-bindings, function-library, namespace-decl⟩, in which the function-library consists of a mapping from function names to function definitions and the variable-bindings consists of a mapping from variable names to variable values.

In the previous sections, we have shown that the query rewriting function rewrites a path expression by inserting the predicate `[sec-inview()]` and by replacing insecure functions and operators with their equivalent security functions. Since the security functions are defined by the query modification algorithm, they are not included in the function library of the initial context. Consequently, a standard XPath evaluator has no idea how to process these security functions. In addition, to process a security function, *e.g.*, the `sec-inview()`, the XPath operator has to know who is the context user. This information, again, is not available in the initial context. In order to enable an XPath evaluator to evaluate the rewritten expressions, we have to add the missing information into the initial context.

The pseudocode in Figure 3.7 shows the definition of the rewritten context e'. The context e' is a superset of e. Besides the elements in e, e' also includes a variable u which is bound to the context user, and a name-definition mapping for the security functions.

3.4 The Overview of the Correctness Proof of the Query Modification Algorithm

In this section we give a overview of the correctness proof of the query modification algorithm. The complete proof can be found in Appendix B.

Theorem 1 *Let D be an XML database, D'_{ur} be a valid read accessibility view (on D) for user u, and q_u be an arbitrary XPath expression posted by u under context e. The query modification algorithm will rewrite q_u and e into q'_u and e' separately, such that the result of evaluating q_u against D'_{ur} under context e is equal to the result of evaluating q'_u against D under e'.*

The objective of this proof is to show that the above theorem is true. The whole proof idea is founded on one concept—the expression level. The XPath grammar shows that there is a level structure in path expressions. For example, a complex XPath expression can be conceived as an expression that applies operations[7] to less complex expressions; a less complex expression, in turn, can be conceived as an expression that applies operations to even simpler expressions, and so on. The process continues until it reaches the *atomic* expressions—the simplest expressions that cannot be further decomposed. Based on this observation, we define the concept of expression level as follows.

Definition 5 (Expression Level) *Every XPath expression has its expression level. Atomic expressions are expressions of level* 1*. An expression is of level k*

[7]We use the term "operation" in a generic sense to refer to not only the standard logical, numeric and comparison operations, but also the function calls and path navigations.

if it has at least one immediate sub-expression of level $k-1$ and no immediate sub-expression of level k or greater.

The level structure of an expression can be visualized as a tree, in which the internal nodes represent operators and the leaves represent atomic expressions. The number of steps of the longest path from the root to leaves represents the level of that expression. A k-level path expression, for instance, can be visualized as a tree of height k, in which the root represents the operation to be applied—*e.g.*, a function call or a logical `and` operation—and the subtrees represent the sub-expressions on which the operation is to be applied. If we use L(XPath) to denote the set of all XPath expressions and L(XPath_k) to denote the set of k-level XPath expressions, obviously we have

$$\bigcup_{k=1}^{\infty} \text{L}(\text{XPath}_k) = \text{L}(\text{XPath})$$

We prove that the theorem holds for all XPath expressions by induction on the expression level k.

Chapter 4

Compact Representation of Access Matrix

In the previous section, we have proposed a query modification algorithm for the secure evaluation of XPath expressions. Given an XPath expression, the query modification algorithm will rewrite it into a secure expression containing the `sec-inview()` function. Each evaluation of the `sec-inview()` involves an authorization decision, for a specific user and object, on the basis of the access control policies. Therefore, the performance of the decision making mechanism, the one that performs the evaluation of `sec-inview()`, largely impacts the speed of secure query evaluation.

As was mentioned in Chapter 2, in discretionary access controls where the access control policies are specified in terms of authorizations, the interactions between authorizations may become extremely complex. Therefore determining a user's privilege directly from access control policies may be slow. A possible solution to fast decision making is to materialize the access matrix. In this case, a decision

making mechanism can answer authorization inquiries by looking up the answers in the materialized access matrix, without computing them from policies. However, in the context of XML databases in which access controls can be specified at the granularity of individual elements or attributes, the space cost of the matrix materialization becomes a serious issue. Storing the matrix as a two-dimensional array is obviously not efficient. How to record the access matrix compactly is therefore the problem we are going to explore in this chapter.

This chapter is organized as follows. In Section 4.1, we review a few conventional implementations of the access matrix. In Section 4.2, we introduce the Compressed Accessibility Map (CAM), a solution recently proposed by Jagadish and his colleagues to the same problem. We describe our codebook method in Section 4.3, and evaluate its space efficiency in Section 4.4.

4.1 Access Control Lists, Capability Lists and Authorization Relations

Access control lists and capability lists, commonly used in operating systems, are two popular implementations of access matrices. Both approaches are based on one observation: as an access matrix is usually large and sparse, *i.e.*, most matrix entries are empty, storing only non-empty matrix entries will achieve good compression.

Under the access control list approach, an access matrix is stored by columns. That is, every object in the matrix is associated with a list of ⟨user, access-mode⟩ pairs, called an access control list, indicating the users and the corresponding access

modes granted on that object. Consider, for example, the access matrix in Table 2.1. The access control list of object O1 consists of ⟨S1, READ⟩, ⟨S1, WRITE⟩, and ⟨S4, READ⟩. Likewise, the access control list of object O2 consists of ⟨S2, READ⟩, ⟨S2, WRITE⟩, and ⟨S3, READ⟩. This implementation makes object-centric operations easy to perform. Operations such as determining all of the users who have access to a specific object or revoking all access to a given object can be easily performed by examining or deleting the access control list of the object in question. However, it complicates subject-centric operations. Revoking all of the access privileges of a given user, for instance, requires a review of all access control lists.

Under the capability list approach, an access matrix is stored by rows. Every subject in the matrix is associated with a list ⟨object, access-mode⟩, called a capability list, indicating the objects and the access modes for which the subject is authorized. For example, the capability list of user S1 in Table 2.1 has two pairs: ⟨O1, READ⟩ and ⟨O1, WRITE⟩. The capability list of user S2 has six pairs: ⟨O2, READ⟩, ⟨O2, WRITE⟩, ⟨O4, READ⟩, ⟨O4, WRITE⟩, ⟨O5, READ⟩, and ⟨O5, WRITE⟩. In contrast to access control lists, capability lists make subject-centric operations easy and object-centric operations hard. For example, it is easy to determine, for a given user, all of the objects the user is authorized to access, but difficult to determine all of the users who have privilege to access a specific object.

An authorization relation, or authorization table, is another popular representation of the access matrix [28]. The table shown in Table 4.1 is an authorization relation for the access matrix in Table 2.1. Each row in the table represents one authorization. If the table is sorted by users, as shown in this example, we get the

effect of capability lists. If the table is sorted by object, we get the effect of access control lists. This representation is commonly used in relational databases.

User	Access Mode	Object
s1	read	o1
s1	write	o1
s2	read	o2
s2	write	o2
s2	read	o4
s2	write	o4
s2	read	o5
s2	write	o5
s3	read	o2
s3	read	o4
s3	read	o5
s4	read	o1
s4	read	o3
s4	write	o3
s4	write	o6

Table 4.1: Authorization Relation

The aforementioned three techniques are proven successful in operating systems and relational databases. However, as we mentioned, they are not good enough for fine-grained XML access control.

4.2 Compressed Accessibility Map

The Compressed Accessibility Map (CAM) is a solution proposed by Jagadish and his colleagues to the problem of the compact representation of an access matrix [20]. The compression of CAM is achieved by exploiting the structural locality of accessibility in hierarchical data.

Recall that a labeled XML tree records a user's privileges in a specific access mode. As has just been described, one way to record a user's privileges in a specific access mode is to maintain, for that user, a capability list containing all of the objects the user is authorized to access in the given mode. This capability list, however, is redundant. Jagadish and his colleagues observed that the accessibility in hierarchical data, *e.g.*, in XML documents, exhibits strong structural locality. That is, in a user's labeled XML tree, the accessible nodes (or inaccessible nodes) tend to cluster together. Therefore, instead of explicitly keeping a list of all accessible objects, they record a labeled XML tree more compactly as a CAM tree. The CAM tree only keeps some *crucial* nodes in a labeled XML tree, as well as some additional information. From the crucial nodes and the additional information, the system can efficiently infer a user's privilege on any objects. A CAM tree is a compact representation of a labeled XML tree. To record the complete access control information, a system should maintain one CAM tree for each user and access mode.

Jagadish and his colleagues developed algorithms to construct an optimal (minimum size) CAM tree for a given user and access mode, and devised an algorithm for efficient lookup. The general idea of the CAM tree construction and lookup is illustrated in the following example. Figure 4.1(a) is a labeled XML tree, where square nodes are accessible and round nodes are not. Its corresponding CAM tree is shown in Figure 4.1(b). Each node in a CAM tree has a label. The semantics of the labels is defined as follows: if node x carries a label s+ (or s-), then node x itself is accessible (or inaccessible); if node x carries a label d+ (or d-) and node x

is node y's closest labeled ancestor, then node y is accessible (or inaccessible). For example, node B can be inferred to be accessible because of its own s+ label. Node U, however, should be inferred to be inaccessible, as its nearest labeled ancestor (the node J) has a d- label.

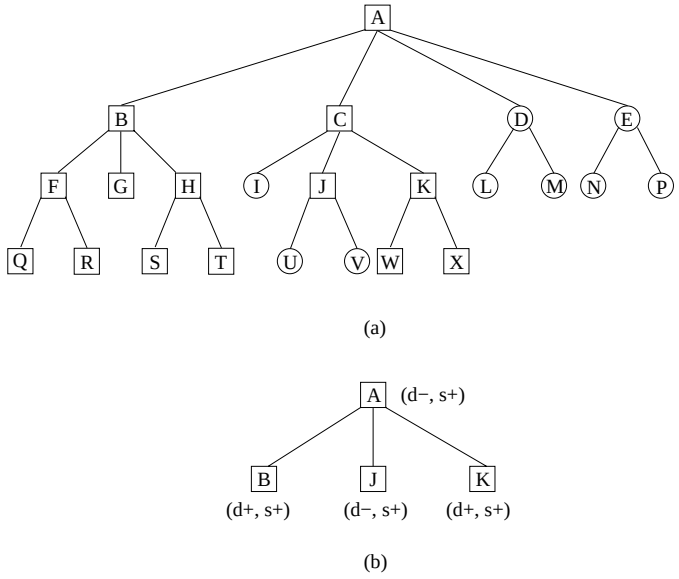

Figure 4.1: Compression Accessibility Map (CAM)

4.3 Codebook Based Scheme

Both the capability lists and the CAM record access control information on a per-subject basis. Their difference lies in the representation of capabilities. In the former approach, a user's capability is represented by a collection of lists, whereas in the latter approach it is represented by a collection of CAM trees. A CAM tree is usually more efficient, as it eliminates the redundancy of a capability list which is caused by the so called structural locality of accessibility. This observation raises a

question: can similar improvements be made to access control lists? After all, most operating systems use access control lists, rather than capability lists.[1] The answer to this question leads to the development of a new compact representation of the access matrix which we are going to present in this and the following sections.

4.3.1 Vector Based Scheme

Suppose, initially, a system has just one access mode. The access cube is then a $|O| \times |U|$ 2-dimensional matrix, where O is the set of objects and U is the set of users. In this matrix, every object is associated with a $|U|$-bit 0-1 vector. Such a vector is called an access control vector, for it records the *accessibility pattern* of that object. Theoretically, each object may have a distinct access control vector; a system with $|O|$ objects and $|U|$ users may have $min(|O|, 2^{|U|})$ distinct access control vectors, which is potentially enormous. However, considering the hierarchical structure of XML data and the propagative behaviors of access control policies, it is reasonable to conjecture that objects closely positioned in an XML hierarchy may share an identical accessibility pattern, and hence the actual number of distinct access control vectors may be much less than the theoretical value. This hypothesis, in fact, is the basis of our method. It will be experimentally validated in the next section. For the time being, we continue the description of our method, assuming that the hypothesis is true.

According to the hypothesis, the actual number of distinct access control vectors in an XML database should be small. Thus, we store the access matrix in two

[1]This is presumably because object-centric operations happen more frequently in operation systems and access control lists can handle object-centric operations gracefully.

parts: an in-memory part and an on-disk part. The in-memory part is an array that contains one copy of each distinct access control vector. We call this array the *access control codebook*. The on-disk part maintains an index for each object. This index, which points to an entry in the access control codebook, identifies the proper access control vector for that object. We call these indexes *access control codes*. The on-disk part can be implemented as a table, in which case the access control information is kept completely separate from the database objects, or it can be implemented by co-locating the indexes with the objects. In order to track the usage of access control vectors, we maintain a reference counter for each vector in the codebook, showing the number of objects currently sharing this vector. Figure 4.2 illustrates the above two implementation choices for the read slice of the access cube A' shown in Table 2.2.

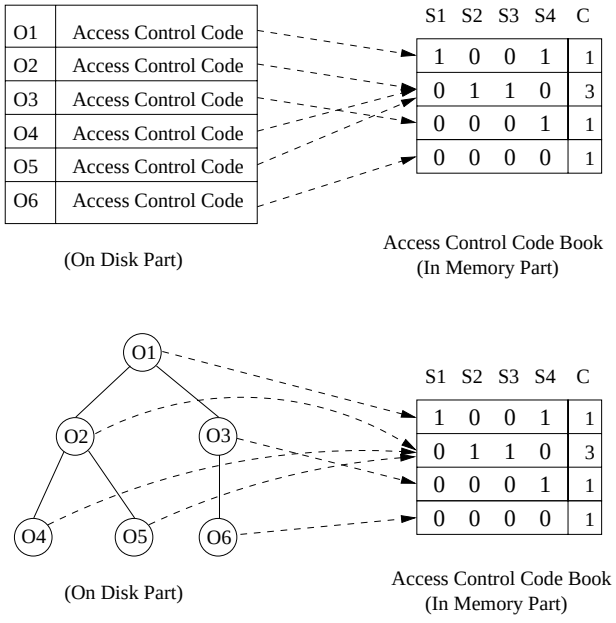

Figure 4.2: Codebook Implementation Schemes

Under this scheme, authorization decision lookup is efficient. Since the access control codebook is supposed to be in memory, the primary cost of a lookup operation is the time of locating the object to be accessed. Once the object is found, we can easily follow the access control code to get the access control vector, and determine the user's accessibility accordingly. If the object in question has been prefetched into memory, which is common in practice, a lookup operation can be performed without accessing secondary storage. Object updates are trivial. Objects can be added into or removed from a database directly, probably with a few minor changes to the codebook. Authorization updates are simple, too. Suppose we want to change an object's access control vector from $\vec{v_1}$ to $\vec{v_2}$. We first search the codebook for vector $\vec{v_2}$. If $\vec{v_2}$ exists, we simply assign the code of $\vec{v_2}$ to that object; otherwise, we insert $\vec{v_2}$ into the codebook first, and then assign its code to that object. When the reference counter of an access control vector hits zero, the vector becomes inactive. The system can either eliminate inactive vectors eagerly or clean them up periodically. Subject updates are relatively complex. Adding a new user, or deleting an existing user, may result in a dimensional change of the codebook. Changing a user's privileges on a set of objects, in the worst case, may result in a code change on every object in that set.

The extra complexity related to subject updates is predictable, because our method, as a variation of access control lists, inherently favors object-centric operations over subject-centric operations. We believe that efficient support for object-centric operations is critical for systems in which object-centric operations appear more frequently than subject-centric operations. In addition, implementing the

codebook as a 2-dimensional array, as demonstrated in this thesis, is just a basic approach. We believe that in some circumstances a more efficient codebook implementation scheme can be developed, and the complexity of subject-centric operations can be reduced. The efficient management of the codebook, however, is not a focus of this thesis. We leave it for future study.

4.3.2 Slab Based Scheme

If a system has multiple access modes, say $|M|$, we have two choices. First, we can view the $|U| \times |O| \times |M|$ 3-dimensional access cube as a stack of $|O|$ independent slabs; each slab is a $|U| \times |M|$ 2-dimensional matrix, holding the complete access control information for a specific object. Following this perspective, we continue to record one code for every object; but the entries of codebook are "slabs" of size $|U| \times |M|$. Alternatively, we can view the 3-dimensional cube as $|O| \times |M|$ 1-dimensional vectors; each vector has $|U|$ bits, recording the complete access control information for a specific object and access mode. In this case, the entries of codebook are still vectors of size $|U|$; but for each object, we have to maintain $|M|$ codes, one for each access mode.

Suppose R denotes the size of one access control code, C denotes the size of the reference counter, V_l denotes the number of distinct access control slabs, and V_c denotes the number of distinct access control vectors. The space cost of the slab approach and the vector approach can be calculated by formulas 4.1 and 4.2 respectively.

$$SIZE_{slab} = \underbrace{|O| \times R}_{\text{on disk}} + \underbrace{V_l \times (|M| \times |U| + C)}_{\text{in memory}} \tag{4.1}$$

$$SIZE_{vector} = \underbrace{|O| \times |M| \times R}_{\text{on disk}} + \underbrace{V_c \times (|U| + C)}_{\text{in memory}} \tag{4.2}$$

Both approaches have advantages and disadvantages. Since $|O|$ is usually much greater than $|U|$ and $|M|$, the overall space cost is likely to be dominated by the cost of the on-disk part. Comparing with the vector approach, the slab approach has less overall space cost, as it has a smaller on-disk part; but it requires more memory.

4.4 Experiment

The experiment has three objectives: (1) to verify the hypothesis, (2) to understand the frequency distribution of access control vectors, and (3) to evaluate the space efficiency of the codebook storage scheme.

4.4.1 Hypothesis Verification

The compression of our method is achieved on the basis of one hypothesis. That is, in most systems, the number of distinct access control vectors (or slabs) is small. In this subsection, we verify this hypothesis with experiments.

Since we could not find examples of production XML databases with fine-grained access controls, we conducted experiments on three similar kinds of data sets. The first consists of access control information from a shared Unix file system at the University of Waterloo. The second consists of access control information from a production instance of Open Text LiveLink, a hierarchical system that provides

Web-based knowledge management services for corporations. The third consists of access control information from a collection of 50 Unix file systems. These 50 Unix file systems were randomly selected from a larger collection of 433 Unix file systems collected by Jagadish and his colleagues for the purpose of CAM evaluation. For each of these three data sets, we count the number of distinct access control vectors for each access mode, and the total number of distinct access control vectors across all of the access modes. For each of the 50 Unix file systems in the third data set, we also count the number of distinct access control slabs.

Waterloo File System

The first data set, collected at the University of Waterloo, describes access controls for a shared Unix file system with 186 users and 1,541,759 files[2]. It supports three access modes: read, write, and execute. The 186 users belong to 60 groups. The group membership information is specified in the file `/etc/group`, and a file's access control information is specified by a 9-character permission code, like `rwxrwxrwx`. We interpret the group membership information and permission codes with the standard Unix access control semantics to determine a user's access privilege.

The experiment was conducted in two steps. First, for each file in the file system, we calculate three access control vectors, one for each access mode, and partition them into three different result sets. An access control vector is a 186-bit 0-1 vector, one bit per user, in which bit i is set to 1 if the

[2]We use the term *file* in a broad sense to refer to any valid entity in a Unix file system, *e.g.*, a file, a directory, or a symbolic link, *etc.*

corresponding user is allowed to access that file in the appropriate access mode. Then, we count the number of distinct vectors for each of the three result sets, and the total number of distinct vectors for the union of the above three result sets, respectively. Table 4.2 summarizes the experimental results.

Mode	Objects	Users	Distinct Access Control Vectors
read	1,541,759	186	250
write	1,541,759	186	278
execute	1,541,759	186	252
combined	1,541,759	186	564

Table 4.2: Access Control Vector Analysis for Waterloo Data

A system like this, in the worst case, may have 1,541,759 distinct access control vectors, *i.e.*, each object may hold one distinct access control vector. However, Table 4.2 shows that the *actual* number of distinct access control vectors is remarkably small: 250 in read mode, 278 in write mode, and 252 in execute mode. In total, we have only 564 distinct access control vectors across three access modes, as shown in the "combined" mode.

LiveLink System

As the time of our data capture, the LiveLink system had 371,549 objects, 1,582 users, 7,057 groups and 10 access modes. The LiveLink system has a much more sophisticated subject hierarchy. Unlike the Unix system, in which a group contains only users, a group in the LiveLink system may contain other groups. Authorizations specified on a group always propagate down to its members. Table 4.3 shows the experimental results for the LiveLink system.

Again, the number of distinct access control vectors for each individual access mode is small, a little more than 4,000 in most cases. The number of distinct access control vectors across access modes, as shown in the "combined" mode, is also small—less than 10,000. It is easy to see that the number of distinct access control vectors in the "combined" mode is much less than the sum of the number of distinct vectors in each individual access mode. This fact indicates that there are considerable amount of duplicates among the sets of distinct access control vectors of each individual access mode.

Mode	Objects	Users	Distinct Access Control Vectors
checkout	371,549	1,582	4,235
creat_node	371,549	1,582	3,926
delete	371,549	1,582	5,591
delete_ver	371,549	1,582	4,273
edit_attr	371,549	1,582	4,144
edit	371,549	1,582	4,362
modify	371,549	1,582	4,684
rm_node	371,549	1,582	22
see_content	371,549	1,582	4,325
see	371,549	1,582	4,198
combined	371,549	1,582	9,863

Table 4.3: Access Control Vector Analysis for LiveLink Data

CAM Data

In order to evaluate the space efficiency of CAM, Jagadish and his colleagues collected access control information from 433 Unix file systems. We randomly selected 50 file systems from those 433 systems for further study. For each of the 50 file systems, we counted (1) the number of distinct access control

vectors for each individual access mode, (2) the number of distinct access control vectors across all access modes, and (3) the number of distinct access control slabs.

The experimental results are shown in Figure 4.3 on page 62. Three of the plots show the number of access control vectors versus the number of users, for read, write, and execute. Two show the number of distinct access control vectors and access control slabs across all access modes versus the number of users. The numbers of distinct access control vectors in those 50 file systems are consistently small. The maximum number of distinct access control vectors (across all access modes) in any file systems is just a little more than 200. About 90% of the systems have less than 100 distinct access control vectors. And, as we conjectured, the number of distinct slabs is greater than the number of distinct access control vectors.

Our experiments on three different data sets all verified the hypothesis. Although these three data sets are not XML data, they are well-defined hierarchical data. It is reasonable for us to conjecture that the same property also exists in XML data.

4.4.2 Frequency Distribution of Access Control Vectors

Our experiments also reveal that the frequency distribution of the access control vectors in some large hierarchical data loosely follows Zipf's law. Zipf's law, named after the Harvard linguistic professor George Kingsley Zipf, is the observation that frequency of occurrence of some event (P), as a function of the rank (i) when the

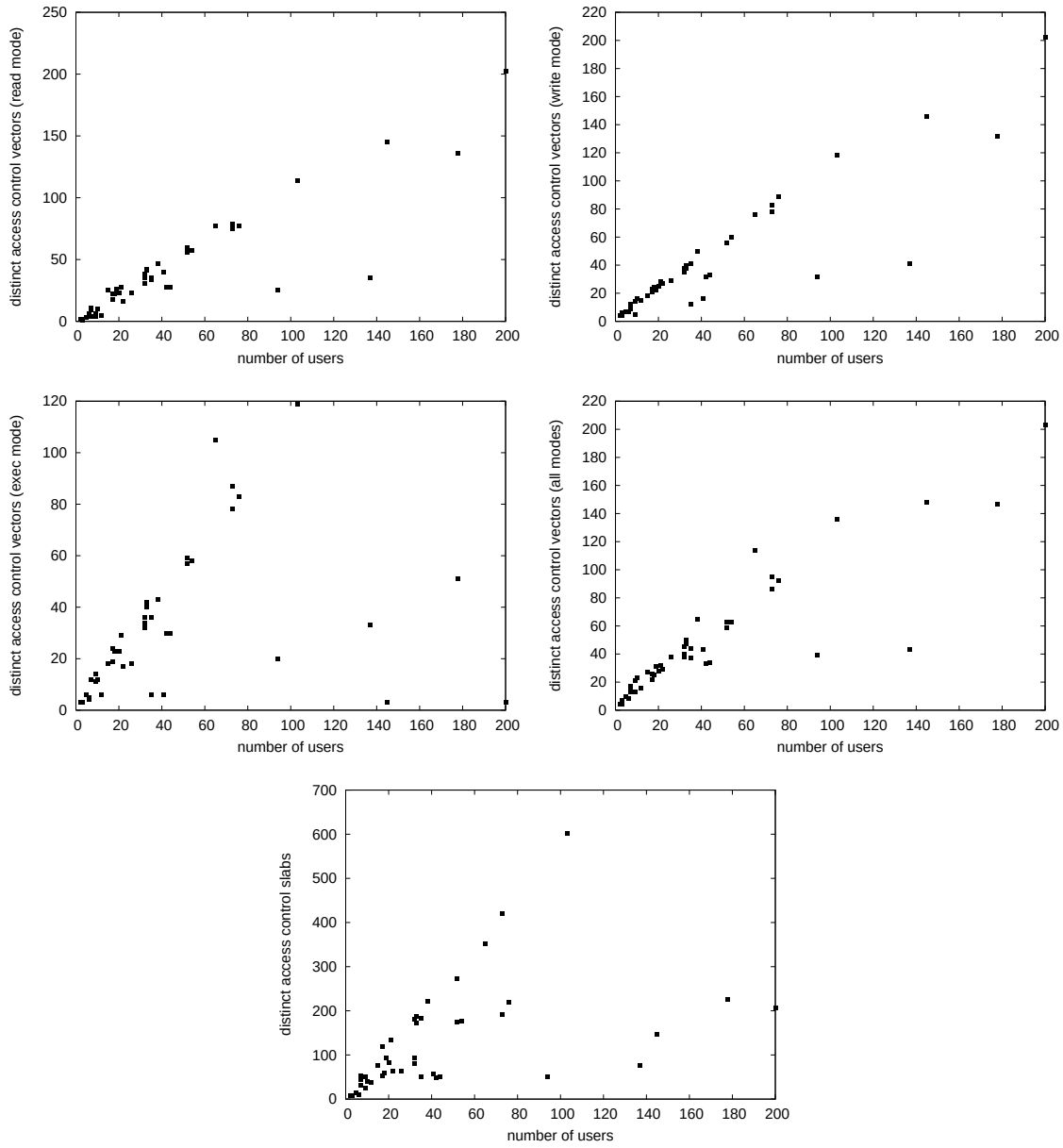

Figure 4.3: Access Control Vector Analysis for CAM Data

rank is determined by the above frequency of occurrence, is a power-law function $P_i \sim 1/i^{\alpha}$ where α is a constant parameter close to unity [33]. Plotted onto a

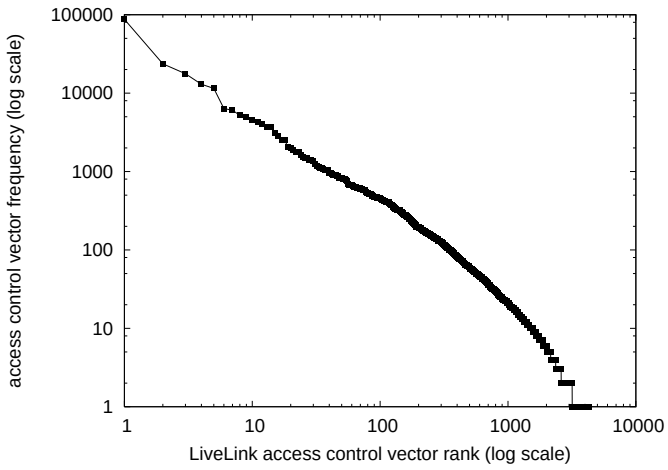

Figure 4.4: Frequency Distribution of Access Control Vectors in LiveLink Data

double-logarithm diagram, the curve of Zipf's distribution function is a descending straight line with a slope close to -1.

Figure 4.4 shows the frequency distribution of the "combined" access control vectors in the LiveLink system. We sort all of the distinct access control vectors in the "combined" mode in descending order by the number of their occurrences. An access control vector of rank 1 is the vector that appears most frequently, an access control vector of rank 2 is the one that appears second most frequently, and so on. Each spot in the diagram represents one distinct access control vector; its x-value represents the rank of the access control vector, and its y-value shows the number of occurrences. The curve shown in Figure 4.4, a descending straight line with a minor deviation at the low end, demonstrates that the frequency distribution of the access control vectors in the LiveLink system loosely follows Zipf's law.

We do not yet know whether it is a property existing in all hierarchical data; but, we know that it does exist in some large ones. Zipf's law says that access

control vectors do not appear at the same rate. There are always a few vectors that appear a lot of times and many vectors that appear only a few times. A possible application of this property is that, if the codebook is too big to fit in memory, we can cache a small portion of the codebook and still get a good hit ratio. We leave the verification of this property and its possible applications for future study.

4.4.3 Performance Evaluation

In this subsection, we evaluate the performance of the newly proposed codebook scheme from two perspectives. First, we compare the performance of the two different implementation approaches: the access control vector approach and the access control slab approach. Second, we compare, for each of the 50 file systems, the space costs of the codebook scheme (implemented in the access control slab approach) with that of two other techniques: the access control list and the CAM.

Vectors Versus Slabs

Recall that the total space cost of the codebook scheme includes two parts: the on-disk part occupied by the access control codes and the in-memory part occupied by the access control codebook. A codebook can be implemented either as an array of access control vectors or as an array of access control slabs. The space cost of the slab approach and the vector approach can be calculated by formulas 4.1 and 4.2 respectively.

In the following experiments, we assume that each access control code occupies 16 bits, which is sufficient to index 65,536 distinct access control vectors (or slabs).

For each of the 50 file systems in the CAM data set, we calculate its in-memory space cost and total space cost under the two different implementation approaches. Figure 4.5 shows the comparison between the total space costs of the two implementation approaches. The first diagram shows the results for the first 25 files and the second diagram shows the results for the remaining 25 files. Figure 4.6 shows the comparison between in-memory space cost (the size of the codebook) of the two implementation approaches. As predicated, the slab approach uses less total space than the vector approach. However, the slab approach requires more space in main memory.

Comparisons between ACL, CAM and Codebook

Under the access control list approach, every object maintains $|M|$ access control lists, one for each access mode. We assume that an access control list consists of a list of ⟨user-id, pointer⟩ pairs, where the user-id is a 16-bit user identifier and the pointer is a 32-bit address pointing to the next pair in the list. That is, each pair occupies 48 bits. The space cost of one access control list equals the number of pairs (in the list) times the size of the pair (48 bits); and, the total space cost of the access control list scheme is the sum of the sizes of all access control lists over all of the objects, which can be calculated by the following formula:

$$\text{Cost}_{\text{ACL}} = 48 \times \sum_{i \in O, j \in M} |\text{ACL}_{ij}|$$

where O is a set of objects, M is a set of access modes, and $|\text{ACL}_{ij}|$ is the number of pairs in the access control list associated with object i and access mode j.

Under the CAM approach, the system maintains $|M|$ CAM trees for each user, one for each access mode. In a system with $|U|$ users, the total number of CAM trees is $|M| \times |U|$. The total space cost of the CAM scheme is the sum of the sizes of all CAM trees, which can be calculated by the following formula:

$$\text{Cost}_{\text{CAM}} = \sum_{i \in U, j \in M} |\text{CAM}_{ij}|$$

where U is a set of users, M is a set of access modes, and $|\text{CAM}_{ij}|$ is the size of the CAM tree for user i and access mode j.

The space cost of a specific CAM tree, say $\text{CAM}_{i,j}$, can be calculated by multiplying the number of labeled nodes (in $\text{CAM}_{i,j}$) with the size of a labeled node. We assume that each labeled node occupies 131 bits. This assumption is in accordance with Jagadish's suggestion. Thus, the total of the CAM scheme can be expressed by the following formula:

$$\text{Cost}_{\text{CAM}} = 131 \times \sum_{i \in U, j \in M} N_{\text{CAM}_{ij}}$$

where $N_{\text{CAM}_{ij}}$ is the number of labeled nodes in CAM_{ij}. In this experiment, the number of labeled nodes in CAM trees, *i.e.*, the $N_{\text{CAM}_{ij}}$, was provided by Jagadish and his colleagues.

Figure 4.7 shows the experimental results. The comparison reveals that our codebook scheme is very space efficient. In most case, its space cost is less than 10% of that of the CAMs.

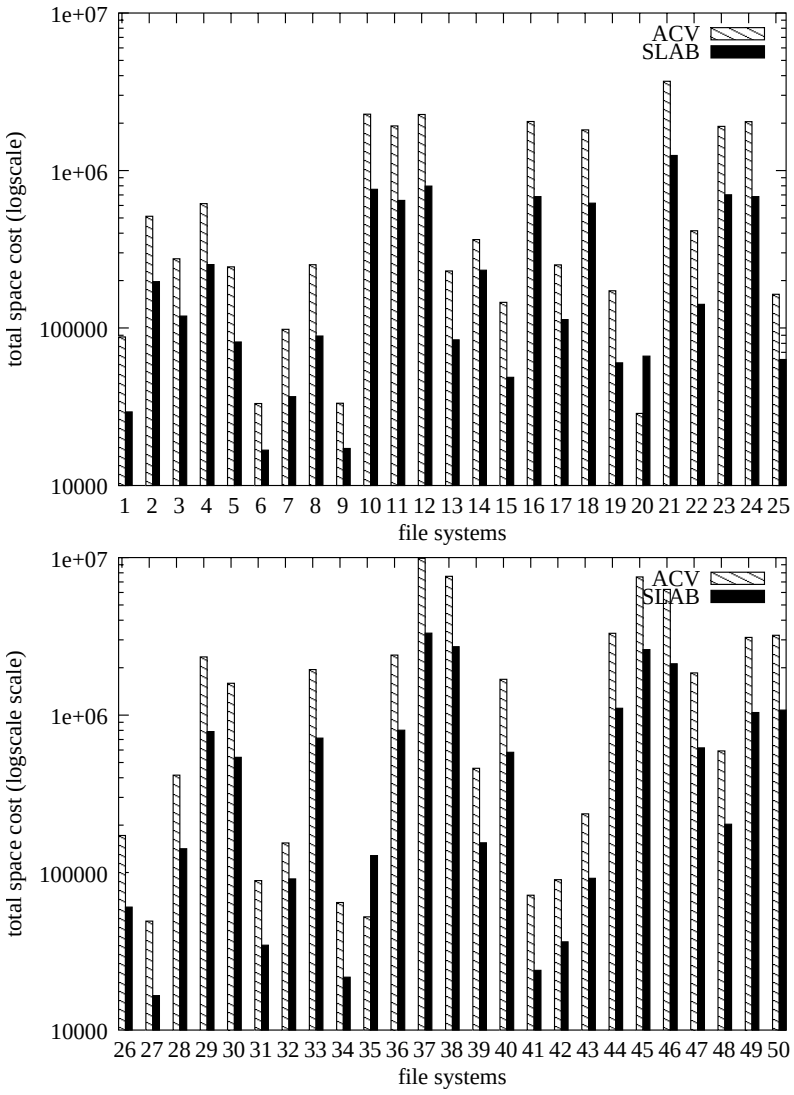

Figure 4.5: Total Space Cost Comparison between ACV and SLAB

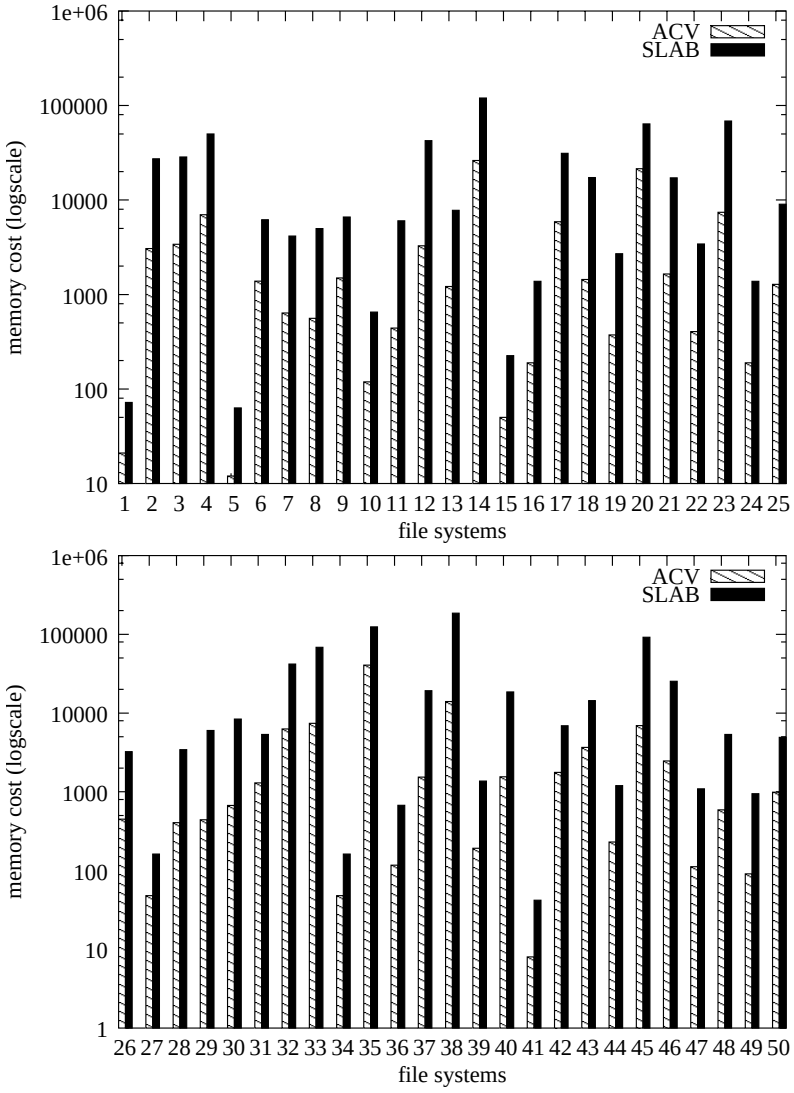

Figure 4.6: In-Memory Space Cost Comparison between ACV and SLAB

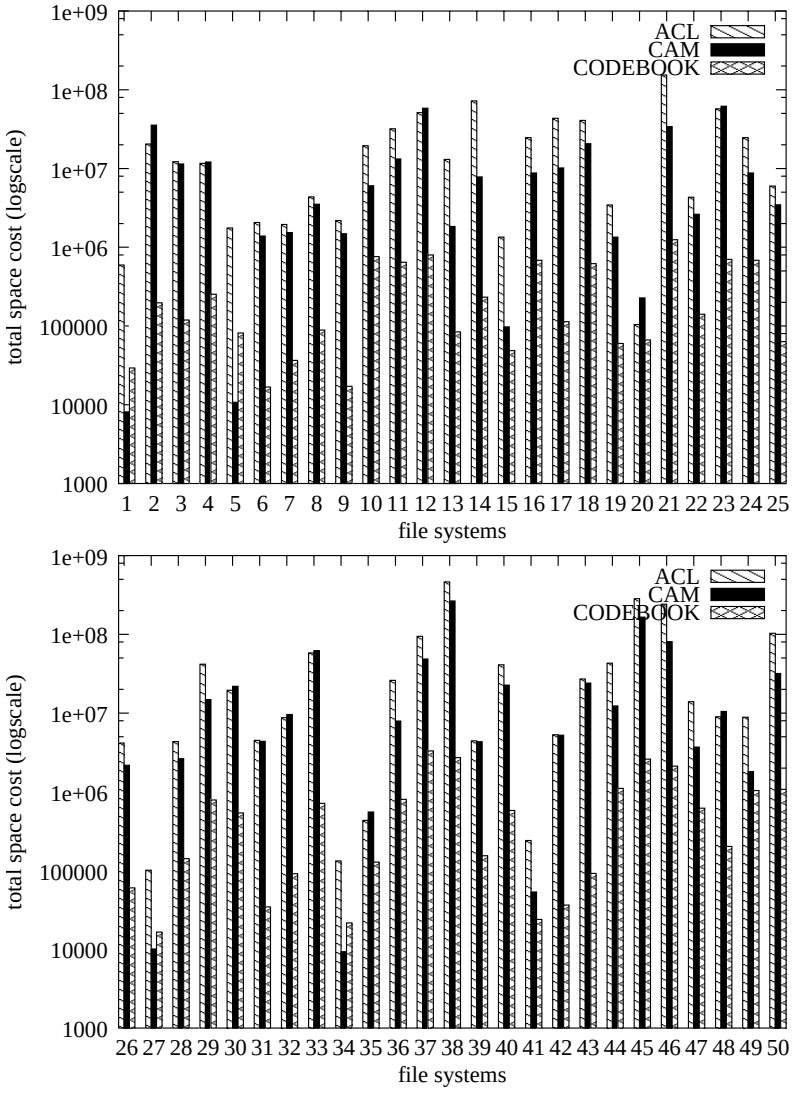

Figure 4.7: Space Cost Comparison between ACL, CAM and Codebook

Chapter 5

Related Work

Research on access controls can be traced back to the 1960s. At that time, the bulk of the research laid in the development of mathematical models for access controls. The goal was to find a simple high-level conceptual model to help researchers better understand the structure and the behavior of an access control system and provide researchers a tool for proving the correctness of an access control system. Of the large number of access control models that were proposed, the Lampson model [24], the Graham and Denning model [17], and the Harrision model [19] are milestones for discretionary access controls; the Bell and LaPadula model [1], the Dion model [11], and the Sea View model [14] are milestones for mandatory access controls.

With the emergence of object oriented databases, researchers began to consider how to apply traditional access control models to protect this new data format. As data items in object oriented databases usually have richer semantics and more complex correlations, traditional access control models were extended to meet the new requirements. Some representative efforts include the ORION model [26, 12]

which is an extension to the discretionary model, and the Meadows model [5] which is an extension to the mandatory model.[1]

Recent research on access controls focuses primarily on three areas. One is the design and the implementation of role based access control models[2] [15, 27]. Another is the development of the *unified* access control systems which is able to specify and enforce multiple access control policies [4, 21, 22]. The third active area is access controls for XML, which is also the focus of this work. In the remainder of this chapter, we explore four research efforts related to XML access controls: (1) the Damiani model, (2) the Bertino model, (3) the Kudo model, and (4) the optimization of the secure evaluation of twig queries. Since CAM has been introduced in Chapter 4, we will not discuss it here.

5.1 Models Proposed by Damiani and Bertino

Damiani *et al.* [10, 8, 9] and Bertino *et al.* [3, 13, 2] each independently proposed a fine-grained access control model for XML. The structures of the models are fairly similar: both of them are extensions to the traditional discretionary access control models; both of them express access control requirements in XML syntax; both of them define the semantics of access controls as a particular view on the XML document; and both of them create the view by means of tree labeling and pruning. Their difference lies in the details of designs, *e.g.*, how to model a subject, how to propagate authorizations, and so on. Here, we use Damiani's model as an example

[1] Castano, Fugini and Samarati provide a complete description of access control models [6].

[2] Some latest information about RBAC can be found at http://csrc.nist.gov/rbac/.

to introduce the major concepts of the models.

In Damiani's model, each XML/DTD document is associated with one authorization sheet. An authorization sheet is a well-formed XML document that contains the authorization rules related to the document to be protected. Each authorization rule is a 5-tuple ⟨subject, object, action, sign, type⟩, where:

Subject is the entity that requests access to the system.

Object is the resource to be protected. The object granularity on which access controls can be specified spans from the DTD level to the element level, or even down to the attribute level.

Action is the access mode. The model, at the time of its writing, supports the read operation only, but it can be extended to incorporate other access modes.

Sign could be a "+" or a "-", indicating whether the authorization rule is positive or negative. A positive authorization states the situations in which access should be authorized, whereas a negative authorization states the situations in which access should be forbidden.

Type specifies the propagation option of the authorization. The model defines eight propagation options, each of which derives implicit authorizations in a different manner. For example, a "recursive" authorization specified on an element will derive implicit authorizations for its subelements and attributes, whereas a "local" authorization will derive implicit authorizations for its attributes only.

The semantics of access control for a user is defined as a view of the document, namely the user's accessibility view, which contains only the data that the user is authorized to access. Damiani and his colleagues devised an algorithm to compute the view by means of tree labeling and pruning. In the tree labeling process, the algorithm traverses the document tree from the root, decides for each node whether the user is authorized to access that node, and marks the node accordingly. The result of the labeling process is a labeled document tree in which every node is marked either "accessible" or "inaccessible." The pruning process is then applied on the labeled document tree to generate the user's accessibility view by pruning off all of the "inaccessible" nodes.[3]

The emphasis in this work is on the design of the XML access control model, focusing on addressing some high level problems related to XML access controls, such as: how to specify access control requirements, how to derive authorization decisions, and how to enforce access controls, with little attention to the efficiency of the access control mechanism.

5.2 XACL

While almost all of the existing access control models assume that a system either authorizes or denies an access request, Kudo and his colleagues proposed a provisional access control model that can provide more sophisticated access controls [18, 23].

[3]In order to preserve the document structure, the pruning process will keep an inaccessible element, if it has accessible descendants.

A provisional access control model adds extended semantics to traditional access control models. Instead of simply authorizing or denying a user's request, a provisional access control model is able to make a more flexible decision, *e.g.*, telling a user that his request will be authorized if he (and/or the system) takes certain security actions prior to the authorization, say signing an agreement of terms and conditions. The provisional access control model is shown to be useful in many e-commerce applications, *e.g.*, online auctions or online contracting, which require conditional authorizations.

Kudo also proposed an XML access control language (XACL), on the basis of the provisional authorization model, which allows the security administrators to write flexible access control requirements in XML syntax. An implementation of an XACL processor is available as a part of XML Security Suite, which is downloadable at IBM's alphaWorks website[4].

5.3 Optimizing the Secure Evaluation of Twig Queries

Cho and colleagues proposed techniques for optimizing the secure evaluation of twig queries in a multi-level security model [7]. The semantics of secure query evaluation of twig queries used in this work was introduced in Chapter 3. In their model, security levels are specified as attributes at the granularity of XML elements, but not every element has a security attribute. For an element without a specified

[4]The XML Security Suite is available at http://www.alphaworks.ibm.com/tech/xmlsecuritysuite/

security attribute, its security level is inherited from its nearest ancestor with a specified security level. Users can only access the elements whose security levels are no higher than theirs.

The first contribution of this work is showing that a twig query can be securely evaluated, in a multi-level security model, by means of query rewriting. The essence of the query rewriting is to append additional security check predicates to the original twig query. The security check predicate determines whether a user is authorized to access a particular object, *i.e.*, whether the user's security level dominates the object's security level. For an element that inherits its security level from its ancestors, such a security check requires a recursive computation to identify the security level of the element's nearest ancestor. Since such a recursive computation is usually expensive, the second contribution of the work is developing a query optimization algorithm that can eliminate unnecessary access control predicates from the rewritten queries by exploiting the security constraints specified in the schemas.

Our work differs from Cho's work in several respects. First, we consider the secure evaluation of XPath expressions, a query language that has more operators than twig queries. Second, unlike Cho's work, which is based on a binding based semantics, our work is based on a view based semantics which is independent of query languages. Third, we also consider the problem of space efficient representation of the access matrix, which is a key problem for achieving fast authorization decision making.

Chapter 6

Conclusions

The first part of this thesis addressed the problem of secure evaluation of XPath expressions. We started with the introduction of a language-independent semantics for secure query evaluation. Based on this semantics, we proposed a query modification algorithm for the secure evaluation of path expressions. Given a path expression, the query modification algorithm rewrites it into a secure expression whose evaluation uses and returns only the data the query issuer is authorized to access. The correctness of this algorithm was proved under our semantics of secure query evaluation.

The second part of the thesis addressed the problem of the compact representation of access matrices, which is critical to making efficient authorization decisions. Our experimental study shows that the access matrices, although large, are very redundant. By exploiting the redundancy, we developed a codebook scheme for the compact storage of access matrices. The codebook scheme exhibits substantial space savings over other storage schemes, such as the access control lists and the

CAM. Savings are more than 90% in most cases.

These two techniques, combined together, provide a foundation for the efficient enforcement of fine-grained access control for query-based access to XML databases.

6.1 Future Work

Our work is only an initial step toward the establishment of secure XML databases. There are many issues that we plan to investigate. One immediate problem we want to explore is the dynamic maintenance of the codebook. The codebook storage scheme favors object-centric operations over subject-centric operations. Ideally, we should develop a structure for the codebook that can adapt to the changes in the subject hierarchy gracefully. Another problem we want to explore is how to combine the best features of CAM and our codebook scheme. Currently, we maintain one access control code for every object in the database. As an alternative, we can maintain a CAM tree for the access control codes. This will not only save space, but also improve the performance of bulk updates.

Appendix A

XPath Query Modification Algorithm

The XPath query modification algorithm is composed of two parts: a query rewriting function and a collection of security functions. The query rewriting function is the heart of the modification algorithm—it defines how a query should be modified; the security functions, on the other hand, are supporting functions defined in the query modification algorithm to perform security-related activities.

A.1 Definition of Query Rewriting Function

The query rewriting function is a function that takes two arguments—an XPath expression q_u posted by user u (the context user) and an initial evaluation context e—and returns the rewritten expression q'_u and the rewritten context e'.

Context Rewriting

Given a context e, the query rewriting function rewrites it to e' as follows.

e'.context-node $\leftarrow$ e.context-node

e'.context-position $\leftarrow$ e.context-position

e'.context-size $\leftarrow$ e.context-size

e'.namespace $\leftarrow$ e.namespace

e'.variable-binding $\leftarrow$ e.variable-binding $\cup$ {(u, context-user)}

e'.function-library $\leftarrow$ e.function-library $\cup$ { `sec-inview`, `sec-id`, `sec-string`, `sec-sum`, `sec-string-length`, `sec-normalize-space`, `sec-numeric-add`, `sec-numeric-subtract`, `sec-numeric-multiply`, `sec-numeric-divide`, `sec-numeric-integer-divide`, `sec-numeric-mod`, `sec-numeric-unary-plus`, `sec-numeric-unary-minus`, `sec-eq`, `sec-neq`, `sec-lt`, `sec-number`, `sec-gt`, `sec-le`, `sec-ge`, `sec-string-value` }

The rewritten context e' is a superset of the context e. Besides the elements in e, context e' also includes a variable u which is bound to the context user, and a name-definition mapping for the security functions.

Query Rewriting

Given a path expression q_u posed by user u, the query rewriting function rewrites it to q'_u. Since the query rewriting is highly related to the structure of path expressions, we define it in terms of a yacc specification [25].

The yacc specification in Figure A.1 is the main body of the definition of the query rewriting. It consists of a collection of grammar rules that are taken from the grammar of XPath specification version 1.0. For each of the grammar rules, we specify an action which will be performed at each time the

rule is recognized. For example, the third and fourth lines in the specification show that the following grammar rule

```
EqualityExpr ::= EqualityExpr '=' RelationalExpr.
```

is associated with the action

```
$$ = "sec-eq( $1, $3 )"
```

That means, if this rule is recognized, the string `sec-eq( $1, $3 )` will be returned, where `$1` and `$3` are pseudo-variables that should be replaced by the values of the nonterminal `EqualityExpr` and `RelationalExpr` on the right side of the grammar rule.

It is worth mentioning that the specification in Figure A.1 is not complete. Many XPath grammar rules are not included. For the sake of simplicity, we omit the grammar rules that are not subject to rewriting. To get a complete definition, one needs to add those missing rules back into this specification, and specify, for each of those missing rules, one action that assigns the concatenation of the values of the symbols on the right hand side of the rule to the symbol on the left hand side.

A.2 Definitions of Security Functions

Security functions are supporting functions defined in the query modification algorithm to perform security-related activities. Table 3.1 shows all of the security

functions defined by the XPath query modification algorithm. Table 3.2 shows the mapping from the insecure functions to their corresponding secure functions. Table 3.3 shows the mapping from the insecure operators to their corresponding secure functions. We give the definitions of the security functions in this section.

Function: ***boolean*** `sec-inview()`

Given the evaluation context e and the access cube A, `sec-inview()` returns true if, and only if, $A[e.u, e.c_n, r] = 1$, where r refers to the read mode, $e.u$ and $e.c_n$ refer to the context user and the context node respectively.

Function: ***string*** `sec-string(`*`object?`*`)`

The `sec-string()` function returns the secure string value of an object.

1. If the argument is omitted, it defaults to a node set with the context node as its only member.
2. If the `object` is an empty node set, an empty string is returned.
3. If the `object` is a non-empty node set that contains at least one accessible node, the *secure* string value of the first node (in document order) in the node set is returned, as if by a call to the `sec-string-value()` function. If the `object` is a non-empty node that does not contain an accessible node, an empty string is returned.
4. If the argument `object` is of other data types, the string value of `object` is returned as if by a call to the standard `string()` function.

Function: ***string*** `sec-string-value(`*`nodeset`*`)`

The `sec-string-value()` function returns the secure string value of the argu-

ment node set. The argument is always a node-set that contains one accessible node.

1. If the node in the argument *nodeset* is a root node, the concatenation of the string values of all accessible text node descendants (in document order) of the root node is returned.
2. If the node in the argument *nodeset* is an element node, the concatenation of the string values of all accessible text node descendants (in document order) of the element node is returned.
3. If the node in the argument *nodeset* is of other Node types, *e.g.*, an attribute node, a namespace node, a processing instruction node, a comment node, or a text node, its standard string value is returned, as if by a call to the standard `string()` function.

Function: ***node-set*** `sec-id(`*`object`*`)`

The `sec-id()` function selects elements by their unique IDs.

1. If the argument is a node set, the result is computed in the following two steps: (1) union the node sets generated by applying the standard `id()` function to the *secure* string value of each node in the argument node set; The secure string value of a node is computed as if by a call to the `sec-string()`. (2) return the accessible nodes in the union.
2. If the argument is an object of any other type, the result is computed in the following two step: (1) generate a node set by applying the standard `id()` function to the string value of that argument *object*; The string

value is computed as if by a call to the standard `string()` function. (2) return the accessible nodes in the node set generated by the previous step.

Function: ***number*** `sec-number(`*`object?`*`)`

The `sec-number()` function converts its argument to a number as follows.

1. If the argument is omitted, it defaults to a node-set with the context node as its only member.
2. If the argument is a node set, the result is a number returned by applying the standard `number()` function to the *secure* string value of the argument. The secure string value is produced as if by a call to the `sec-string()` function.
3. If the argument is an object of other data types, the result is a number returned by applying the standard `number()` function to that argument.

Function: ***number*** `sec-sum(`*`node-set`*`)`

The `sec-sum()` converts each node in the argument node set into a number by calling the `sec-number()` function and returns the sum.

Function: ***number*** `sec-string-length(`*`string?`*`)`

If the argument is omitted, it defaults to the secure string value of the context node. The secure value of the context node is produced as if by a call to the `sec-string()` function.

Otherwise, the `sec-string-length()` returns the number of characters in the string value of the argument by calling the standard `string-length()`

function.

Function: ***number*** `sec-normalize-space(`*`string?`*`)`

If the argument is omitted, it defaults to the secure string value of the context node. The secure value of the context node is produced as if by a call to the `sec-string()` function.

Otherwise, the `sec-normalize-space()` returns the normalized string of the argument by calling the standard `normalize-space()` function.

Secure Functions for Comparison Operators

XPath 1.0 defines six comparison operators which include `=`, `!=`, `<=`, `<`, `>=` and `>`. Their corresponding secure functions are: `sec-eq()`, `sec-ne()`, `sec-le()`, `sec-lt()`, `sec-ge()` and `sec-gt()`. The semantics of these comparison security functions is the same as that of those standard comparison operators, except that the standard comparison operators convert operants into numbers or strings by calling the standard functions, like `number()` and `string()`, but the comparison functions convert them by calling the secure functions, like `sec-number()` and `sec-string()`.

Secure Functions for Numeric Operators

XPath 1.0 defines five numeric operators which include `+`, `-`, `*`, `div`, and `mod`. Their corresponding secure functions are: `sec-addition()`, `sec-subtraction()`, `sec-multiply()`, `sec-div()` and `sec-mod()`.

The semantics of the numeric security functions is the same as that of those standard numeric operators, except that the standard numeric operators con-

vert operants to numbers by calling the standard `number()` function, but the numeric security functions convert their arguments to numbers by calling the secure `sec-number()` function.

Figure A.1 The Definition of The Query Rewriting

```
[4] EqualityExpr       ::= RelationalExpr
                             { $$ = $1 }
                         | EqualityExpr '=' RelationalExpr
                             { $$ = "sec-eq( $1, $3 )" }
                         | EqualityExpr '!=' RelationalExpr
                             { $$ = "sec-ne( $1, $3 )" }
[5] RelationalExpr     ::= AdditiveExpr
                             { $$ = $1 }
                         | RelationalExpr '<' AdditiveExpr
                             { $$ = "sec-lt( $1, $3 )" }
                         | RelationalExpr '>' AdditiveExpr
                             { $$ = "sec-gt( $1, $3 )" }
                         | RelationalExpr '<=' AdditiveExpr
                             { $$ = "sec-le( $1, $3 )" }
                         | RelationalExpr '>=' AdditiveExpr
                             { $$ = "sec-ge( $1, $3 )" }
[6] AdditiveExpr       ::= MultiplicativeExpr
                             { $$ = $1 }
                         | AdditiveExpr '+' MultiplicativeExpr
                             { $$ = "sec-addition( $1, $3 )" }
                         | AdditiveExpr '-' MultiplicativeExpr
                             { $$ = "sec-subtraction( $1, $3 )" }
[7] MultiplicativeExpr::= UnaryExpr
                             { $$ = $1 }
                         | MultiplicativeExpr MultiplyOperator UnaryExpr
                             { $$ = "sec-multiply( $1, $3 )" }
                         | MultiplicativeExpr 'div' UnaryExpr
                             { $$ = "sec-div( $1, $3 )" }
                         | MultiplicativeExpr 'mod' UnaryExpr
                             { $$ = "sec-mod( $1, $3 )" }
[16] Step               ::= AxisSpecifier NodeTest Predicate*
                             { $$ = "$1 $2 [sec-inview()] $3" }
```

```
[35] FunctionName      ::= Qname - NodeType
                             { switch(Qname)
                                 case id:
                                        replace id with sec-id
                                 case string:
                                        replace string with sec-string
                                 case string-length:
                                        replace string-length with
                                        sec-string-length
                                 case normalize-space:
                                        replace normalize-space with
                                        sec-normalize-space
                                 case number:
                                        replace number with sec-number
                                 case sum:
                                        replace sum with sec-sum
                                 case others:
                                        do nothing
                             }
```

Appendix B

Correctness Proof of the XPath Query Modification Algorithm

B.1 Objective and Assumptions

The objective of the proof is to show the correctness of Theorem 1. We prove it on the basis of the following assumptions:

1. The root node of an XML document is accessible with respect to user u.
2. The context node n_c in the initial context is accessible with respect to user u.
3. The original expression q_u does not contain functions that are not defined in the XPath core function library.
4. The input arguments of functions are always objects of the four basic types.
5. The original expression q_u is in the verbose syntax.

B.2 Notation

1. D denotes an XML database instance.
2. D'_{ur} denotes the user u's valid read accessibility view on D.
3. q_u denotes an XPath expression, posted by u, whose corresponding rewritten expression is q'_u.
4. e denotes the initial evaluating context whose corresponding rewritten context is e'.
5. $e.u$ denotes the context user in evaluation context e.
6. $q_u(D)$ denotes the result of evaluating q_u against D.
7. $q_u(D'_{ur})$ denotes the result of evaluating q_u against the view D'_{ur}.
8. L(XPath) denotes the set of all XPath expressions.
9. L(XPath_k) denotes the set of k-level XPath expressions.

B.3 Proof Skeleton

We prove that the theorem holds on L(XPath) by induction on the expression level k.

Base Case:

We wish to show that the theorem holds for L(XPath_1).

The grammar of XPath illustrates that an expression in L(XPath$_1$), *i.e.*, an atomic expression, must be in one of the following five forms: a float number, a quoted string literal, a variable reference, a location step without predicates, or a function call without arguments. We prove that the theorem holds for these five cases in three steps.

First step: suppose q_u is an atomic expression in one of the first three forms. According to the query modification algorithm, we have $q'_u = q_u$. If q_u is in one of the first three forms, the result of q_u is independent of the database. Thus, we have $q_u(D'_{ur}) = q'_u(D)$.

Second step: suppose q_u is a location step without predicates. Then, q_u is either a relative expression in the form of `Axis::NodeTest` or an absolute expression in the form of `/Axis::NodeTest`. The difference between a relative expression and an absolute expression is that a relative expression is evaluated with respect to the current context node, whereas, an absolute expression is evaluated with respect to the document root. According to the assumption, the context node (*i.e.*, the $e.n_c$) and the document root (addressed by "/") are both accessible with respect to the context user u. Thus the proof for these two cases are actually the same. We give a proof for the relative expression as an example.

According to the query modification algorithm, if q_u is `Axis::NodeTest`, the rewritten expression q'_u would be `Axis::NodeTest[sec-inview()]`.

Assume that n is an arbitrary node in `Axis::NodeTest`(D'_{ur}). It must satisfy the following conditions:

1. n is an accessible node in D, as $n \in D'_{ur}$.
2. The relationship between n and the context node $e.n_c$ satisfies the `Axis`.
3. The node type and the expanded-name of n satisfies the `NodeTest`.

Consequently, n must also be a node in `Axis::NodeTest[sec-inview()]`(D).

Similarly, assuming n is an arbitrary node in `Axis::NodeTest[sec-inview()]`(D). It must satisfy the following conditions:

1. The relationship between n and the context node $e'.n_c$ satisfies the `Axis`.
2. The node type and the expanded-name of n satisfies the `NodeTest`.
3. n is an accessible node in D. Since the security policy to be enforced is a valid policy, we have $n \in D'_{ur}$.

Consequently, n must also be a node in `Axis::NodeTest`(D'_{ur}).

Therefore, we have $q'_u(D) = q_u(D'_{ur})$.

Third step: suppose q_u is a function call without arguments. The XPath specification shows that there are 11 standard functions that may not take arguments. Of the 11 functions, 4 functions are insecure. They are the `string()`, the `string-length()`, the `normalize-space()`, and the `number()`. We will show the proof for these 4 insecure functions in Section B.4.

Hypothesis:

Assume that the theorem holds for L(XPath$_i$), where $1 \leq i \leq k$ for some integer k greater than or equal to 1.

Induction Step:

We wish to prove that the theorem holds for $\mathrm{L}(\mathrm{XPath}_{k+1})$.

Recall that a $(k+1)$-level expression is an expression that applies *one* operation to a number of subexpressions whose maximum level is k. According to the hypothesis, we know that the theorem holds for all of the subexpressions. Therefore, to prove that the theorem holds on a $(k+1)$-level expression, we just need to show that the rewriting of that operation is correct.

The following list illustrates all of the possible operations an expression may take.

1. Comparison Operations: `=`, `!=`, `<`, `>`, `<=`, `>=`
2. Numeric Operations: `+`, `-`, `div`, `mod`, `*`
3. Logical Operations: `or`, `and`
4. Navigation Operators: `/`
5. Predicates: applying a "predicate" operation to an expression means appending the predicate at the end of that expression.
6. Function Calls: applying a "function call" operation to a number of expressions means calling that function with those expressions as arguments.

We classify the operations into six categories: the comparison operations, the numeric operations, the logical operations, the navigation operations, the predicates, and the function calls. In the following sections, we will prove that the rewriting of the operations in these six categories is correct.

B.4 Proof for Function Calls

Since the results of secure functions are solely determined by the evaluating context and the input arguments (if available), it is easy to see that evaluating a secure function against D'_{ur} equals evaluating that function against D, given the same evaluating context and arguments. That is, the theorem holds if the $(k+1)$-level expression is a function call to a *secure* function.

In the remainder of this section, we prove that the theorem holds if the $(k+1)$-level expression is a function call to an *insecure* function.

Of the 27 standard functions defined by the XPath specification, 6 are insecure functions: `id()`, `string()`, `string-length()`, `normalize-space()`, `number()`, and `sum()`. We prove them separately.

`string()` Function

Suppose q_u is `string(`p`)`, where p is a k-level expression. According to the query modification algorithm, the rewritten expression q'_u is `sec-string(`p'`)`, where p' is the rewritten expression for p. We prove that the theorem holds for the expression q_u in three steps. First step: suppose $p(D'_{ur})$ yields an object o which is of one of the following three types: a number, a boolean, or a string. According to the definition of `sec-string()`, the result of `sec-string`$(p')(D)$ is equal to the result of `string`$(p)(D)$. As the string value of o is independent of the database, the result of `string`$(p)(D)$ must be equal to the result of `string`$(p)(D'_{ur})$. Thus, we have `sec-string`$(p')(D) =$ `string`$(p)(D'_{ur})$.

Second step: suppose $p(D'_{ur})$ yields a node set. If the node set is empty, it is easy to see that the theorem holds, as both `string`$(p)(D'_{ur})$ and `sec-string`$(p')(D)$

will return an empty string. If the node set is not empty, we assume that n is the first node in the node set. Obviously, n is an accessible node that appears in both D and D'_{ur}.

Assume that `string`$(p)(D'_{ur})$ yields a string s_1, and `sec-string`$(p')(D)$ yields a string s_2. Let N_1 be the set of text node descendants of n in D'_{ur}, and N_2 be the set of *accessible* text node descendants of n in D. According to the definition of the `string()` function, s_1 is the concatenation of the string values of the nodes in N_1 in document order. Similarly, according to the definition of `sec-string()` function, s_2 is the concatenation of the string values of the nodes in N_2 in document order. Since D'_{ur} is a view of D, D'_{ur} and D must share the same document order. Therefore, to prove that s_1 is equal to s_2, we only need to show that N_1 is equal to N_2.

Pick an arbitrary node i from N_1. According to the definition of N_1, node i must satisfy the following conditions:

1. i is a node in D'_{ur}. That is, i is an accessible node in D.
2. i is a text node.
3. i is a descendant of n.

Obviously, $n \in N_2$, as it satisfies all of the criteria of N_2.

Similarly, pick an arbitrary text node i from N_2. According to the definition of N_2, node i must satisfy the following conditions:

1. i is a text node.

2. i is a node in D.

3. i is a descendant of n.

4. i is accessible.

Obviously, $n \in N_1$, as it satisfies all of the criteria of N_1.

Thus, we have $N_1 = N_2$.

A `string()` function may be called without arguments, in which case q_u is an atomic expression like `string()`, and the rewritten expression q'_u is `sec-string()`. According to the definition of `string()`, if the argument is omitted, it defaults to a node set which contains only the context node. That is, the expression `string()` is actually equivalent to the expression `string(`N`)`, where N is a node set that contains only the context node. In the previous proof, we have shown that the theorem holds for expression `string(`N`)`. Therefore, the theorem must also hold for the expression `string()`.

`id()` Function

Suppose q_u is `id(`p`)`, where p is a k-level expression. According to the query modification algorithm, the rewritten expression q'_u is `sec-id(`p'`)`, where p' is the rewritten expression for p. We prove that the theorem holds for the expression q_u in two steps.

First step: suppose $p(D'_{ur})$ yields an object o which is of one of the following three types: a number, a boolean, or a string. We assume that `id`$(p)(D'_{ur})$ yields the node set N_1, and `sec-id`$(p')(D)$ yields the node set N_2.

According to the definition of function `id()`, a node $n \in N_1$ must satisfy the following conditions.

1. $n \in D'_{ur}$. That is, n is an accessible node in D.
2. The id of n is a token that appears in the string value of $p(D'_{ur})$.

Similarly, according to the definition of function `sec-id()`, a node $n \in N_2$ must satisfy the following conditions.

1. n is a node in D.
2. n is accessible
3. The id of n is a token that appears in the string value of $p'(D)$.

According to hypothesis, we have $p(D'_{ur}) = p'(D) = o$. Since the string value of o in D is equal to the string value of o in D'_{ur}, we have $N_1 = N_2$.

Second step: suppose $p(D'_{ur})$ yields a node set N. According to the definition of `sec-id()`, a node $n \in$ `sec-id`$(p')(D)$ must satisfy the following conditions:

1. n is a node in D.
2. n is accessible.
3. The id of node n appears in the concatenation of the *secure* string values of the nodes in node set N in D.

Since the secure string value of a node in D is equal to its string value in D'_{ur}, n must also be a node in `id`$(p)(D'_{ur})$.

Similarly, according to the definition of `id()`, a node $n \in \texttt{id}(p)(D'_{ur})$ must satisfy the following conditions:

1. n is a node in D'_{ur}.
2. The id of node n appears in the concatenation of the string values of the nodes in node set N in D'_{ur}.

Since the secure string value of a node in D is equal to its string value in D'_{ur}, n must also be a node in $\texttt{sec-id}(p')(D)$.

Thus, we have $\texttt{sec-id}(p')(D) = \texttt{id}(p)(D'_{ur})$

`number()` Function

Suppose q_u is `number(`p`)`, where p is a k-level expression. According to the query modification algorithm, the rewritten expression q'_u is `sec-number(`p'`)`, where p' is the rewritten expression for p. We prove that the theorem holds for q_u in two steps.

First step: suppose $p(D'_{ur})$ yields an object o which is in one of the following three types: a number, a boolean, or a string. According to the definition of `sec-number()`, $\texttt{sec-number}(p')(D)$ converts the object o to a number by calling the standard `number()` function. As the number value of o is independent of the database, it is easy to see that

$$\texttt{sec-number}(p')(D) = \texttt{number}(p)(D'_{ur})$$

Second step: suppose $p(D'_{ur})$ yields a node set N. According to the definition

of function `sec-number()`, the `sec-number`$(p')(D)$ converts the secure string value of N to a number as if by calling the standard `number()` function. Since the secure string value of a node set in D is equal to the string value of a node set in D'_{ur}, we have

$$\texttt{sec-number}(p')(D) = \texttt{number}(p)(D'_{ur})$$

A `number()` function may be called without arguments, in which case q_u is an atomic expression like `number()`, and the rewritten expression q'_u would be `sec-number()`. According to the definition of function `number()`, if the argument is omitted, it defaults to a node set that contains only the context node. That is, the expression `number()` is actually equivalent to the expression `number(`N`)`, where N is a node set that contains only the context node. In the previous proof, we have shown that the theorem holds for the expression `number(`N`)`. Therefore, it is obvious that the theorem also holds for the expression `number()`.

`sum()` Function

Suppose q_u is `sum(`p`)`, where p is a k-level expression that returns a node set. According to the query modification algorithm, the rewritten expression q'_u must be `sec-sum(`p'`)`, where p' is the rewritten expression for p.

Suppose $p(D'_{ur})$ yields a node set N. According to the definition of function

`sec-sum()`, we have

$$\texttt{sec-sum}(p')(D) = \sum_{n \in N} \texttt{sec-number}(n)(D)$$

Since we have proven that

$$\texttt{sec-number}(n)(D) = \texttt{number}(n)(D'_{ur})$$

it is easy to see that the following equations hold.

$$\begin{aligned} \texttt{sec-sum}(p')(D) &= \sum_{n \in N} \texttt{sec-number}(n)(D) \\ &= \sum_{n \in N} \texttt{number}(n)(D'_{ur}) \\ &= \texttt{sum}(p)(D'_{ur}) \end{aligned}$$

`string-length()` Function

Suppose q_u is `string-length(`p`)`, where p is a k-level expression that returns a string. According to the query modification algorithm, the rewritten expression q'_u must be `sec-string-length(`p'`)`, where p' is the rewritten expression for p.

Suppose $q_u(D'_{ur})$ yields a string s. According to the hypothesis, we have $q_u(D'_{ur}) = q'_u(D) = s$. According to the definition of `sec-string-length()`, the `sec-string-length`$(p)(D)$ will return the number of characters in string

s. It is easy to see that

$$\texttt{string-length}(p)(D'_{ur}) = \texttt{sec-string-length}(p')(D)$$

A `string-length()` function may be called without arguments, in which case q_u is an atomic expression like `string-length()`, and the rewritten expression q'_u would be `sec-string-length()`. According to the definition of function `string-length()`, if the argument is omitted, it defaults to the string value of the context node. That is, assuming n is the context node and s is the string value of n in D'_{ur}, the expression `string-length()`(D'_{ur}) will return the number of characters in s. Similarly, according to the definition of function `sec-string-length()`, if the argument is omitted, it defaults to the secure string value the context node. In the previous proof, we have shown that the string value of the context node in D'_{ur} is equal to the *secure* string value of the context node in D, *i.e.*, $\texttt{string}(n)(D'_{ur}) = \texttt{sec-string}(n)(D)$. Therefore, it is easy to see that the theorem also holds for the expression `string-length()`.

`normalize-space()` Function

Suppose q_u is `normalize-space(`p`)`, where p is a k-level expression that returns a string. According to the query modification algorithm, the rewritten expression q'_u must be `sec-string-length(`p'`)`, where p' is the rewritten expression for p.

Suppose $p(D'_{ur})$ yields a string s. According to the hypothesis, we have

$p(D'_{ur}) = p'(D) = s$. According to the definition of `sec-normalize-space()`, the `sec-normalize-space`$(p)(D)$ will return the normalized string of s as if by a call to the standard `normalize-space()` function. It is easy to see that the following equation holds.

$$\texttt{normalize-space}(p)(D'_{ur}) \;=\; \texttt{sec-normalize-space}(p')(D)$$

A `normalize-space()` function may be called without arguments, in which case q_u is an atomic expression like `normalize-space()`, and the rewritten expression q'_u would be `sec-normalize-space()`. According to the definition of function `normalize-space()`, if the argument is omitted, it defaults to the string value of the context node. That is, assuming n is the context node and s is the string value of n in D'_{ur}, the expression `normalize-space()`(D'_{ur}) will return the normalized string value of s. Similarly, according to the definition of function `sec-normalize-space()`, if the argument is omitted, it defaults to the secure string value the context node. In the previous proof, we have shown that the string value of the context node in D'_{ur} is equal to the *secure* string value of the context node in D, *i.e.*, `string`$(n)(D'_{ur}) =$ `sec-string`$(n)(D)$. Therefore, it is easy to see that the theorem also holds for the expression `normalize-space()`.

B.5 Proof for Comparison Operators

XPath 1.0 defines 6 comparison operators which include `=`, `!=`, `<`, `>`, `<=`, and `>=`. Since the proof for comparison operators are quite similar, we show one example proof for the `=` operator.

Suppose q_u is an expression like $p_1 = p_2$, where p_1 and p_2 are two path expressions whose levels are less than or equal to k. According to the query modification algorithm, the rewritten expression q'_u is `sec-eq(`p'_1`,` p'_2`)`, where p'_1 and p'_2 are rewritten expressions of p_1 and p_2 respectively.

Assume $p_1(D'_{ur})$ yields an object o_1 and $p_2(D'_{ur})$ yields an object o_2. We prove that the theorem holds for the expression q_u in three steps.

First step: suppose neither o_1 nor o_2 is a node set. If one of o_1 and o_2 is a boolean, according to the definition of the operator `=`, the result of $(p_1 = p_2)(D'_{ur})$ is equal to the result of $\texttt{boolean}(o_1)(D'_{ur}) = \texttt{boolean}(o_2)(D'_{ur})$. According to the definition of `sec-eq()`, the result of $\texttt{sec-eq}(o_1, o_2)(D)$ is equal to the result of $\texttt{boolean}(o_1)(D) = \texttt{boolean}(o_2)(D)$. Since

$$\begin{aligned} \texttt{boolean}(o_1)(D'_{ur}) &= \texttt{boolean}(o_1)(D) \\ \texttt{boolean}(o_2)(D'_{ur}) &= \texttt{boolean}(o_2)(D) \end{aligned}$$

it is easy to see that

$$(p_1 = p_2)(D'_{ur}) = \texttt{sec-eq}(p'_1, p'_2)(D)$$

Similarly, if one of o_1 and o_2 is a number or a string, we can also prove that

$$(p_1 = p_2)(D'_{ur}) = \texttt{sec-eq}(p'_1, p'_2)(D)$$

Second step: suppose that both o_1 and o_2 are node sets. If the result of $(p_1 = p_2)(D'_{ur})$ is true, according to the definition of the operator `=`, there must exist a node $n_1 \in o_1$ and a node $n_2 \in o_2$ such that the string value of n_1 in D'_{ur} is equal to the string value of n_2 in D'_{ur}. Since we have shown that the string value of a node in D'_{ur} is equal to its secure string values in D, it is easy to see that the result of $\texttt{sec-eq}(p'_1, p'_2)(D)$ must be true. Similarly, if the result of $\texttt{sec-eq}(p'_1, p'_2)(D)$ is true, according to the definition of `sec-eq()`, there must exist a node $n_1 \in o_1$ and a node $n_2 \in o_2$ such that the secure string value of n_1 in D is equal to the secure string value of n_2 in D. Again, since the secure string value of a node in D is equal to its string values in D'_{ur}, it is easy to see that the result of $(p_1 = p_2)(D'_{ur})$ must be true. Thus, we have $(p_1 = p_2)(D'_{ur}) = \texttt{sec-eq}(p'_1, p'_2)(D)$.

Third step: suppose that one of o_1 and o_2 is a node set. Without loss of generality, we assume that o_1 is a node set.

Suppose o_2 is a number. If the result of $(p_1 = p_2)(D'_{ur})$ is true, according to the definition of the operator `=`, there must exist a node $n_1 \in o_1$ such that the result of $\texttt{number}(\texttt{string}(n_1))(D'_{ur}) = o_2$ is true. Since $\texttt{string}(n_1)(D'_{ur}) = \texttt{sec-string}(n_1)(D)$, we know that $\texttt{number}(\texttt{sec-string}(n_1))(D) = o_2$ must be true. Consequently, the result of $\texttt{sec-eq}(p'_1, p'_2)(D)$ must be true. Similarly, if the result of $\texttt{sec-eq}(p'_1, p'_2)(D)$ is true, according to the definition of `sec-eq`, there must exist a node $n_1 \in o_1$ such that the result of $\texttt{number}(\texttt{sec-string}(n_1))(D) = o_2$ is true.

Since $\texttt{sec-string}(n_1)(D) = \texttt{string}(n_1)(D'_{ur})$, we know that $\texttt{number}(\texttt{string}(n_1))(D'_{ur}) = o_2$ must be true. Consequently, the result of $(p_1 = p_2)(D'_{ur})$ must be true. Thus, we have $(p_1 = p_2)(D'_{ur}) = \texttt{sec-eq}(p'_1, p'_2)(D)$.

Similarly, if o_2 is a string or a boolean, we can also prove that $(p_1 = p_2)(D'_{ur}) = \texttt{sec-eq}(p'_1, p'_2)(D)$.

B.6 Proof for Numeric Operators

XPath 1.0 defines 5 numeric operators which include `+`, `-`, `*`, `div`, and `mod`. Since the proof for these numeric operators are quite similar, we show one example proof for the `+` operator.

Suppose q_u is an expression like $p_1 + p_2$, where p_1 and p_2 are two path expressions whose levels are less than or equal to k. According to the query modification algorithm, the rewritten expression q'_u is $\texttt{sec-add}(p'_1, p'_2)$, where p'_1 and p'_2 are rewritten expressions of p_1 and p_2 respectively.

According to the definition of numeric operators, $(p_1 + p_2)(D'_{ur})$ will convert its operands to numbers as if by a call to the standard `number()` function and return the addition of the two numbers. According to the previous proof, we have

$$\begin{aligned} \texttt{number}(p_1)(D'_{ur}) &= \texttt{sec-number}(p'_1)(D) \\ \texttt{number}(p_2)(D'_{ur}) &= \texttt{sec-number}(p'_2)(D) \end{aligned}$$

Therefore, we have

$$\begin{aligned}(p_1 + p_2)(D'_{ur}) &= \texttt{number}(p_1)(D'_{ur}) + \texttt{number}(p_2)(D'_{ur}) \\ &= \texttt{sec-number}(p'_1)(D) + \texttt{sec-number}(p'_2)(D) \\ &= \texttt{sec-add}(p'_1, p'_2)(D)\end{aligned}$$

B.7 Proof for Navigation Operators

Suppose q_u is an expression like p_1/p_2, where p_1 and p_2 are two path expressions whose levels are less than or equal to k. According to the query modification algorithm, the rewritten expression q'_u is p'_1/p'_2, where p'_1 and p'_2 are rewritten expressions of p_1 and p_2 respectively.

According to the definition, the result of $(p_1/p_2)(D'_{ur})$ is computed in two steps. First, $p_1(D'_{ur})$ is evaluated to generate a node set, say N_1. Then, for each node in N_1, the expression $p_2(D'_{ur})$ is evaluated with that node as the context node. The union of the sets of the nodes identified by $p_2(D'_{ur})$ is the final result set. According to the hypothesis, we have

$$\begin{aligned}p_1(D'_{ur}) &= p'_1(D) \\ p_2(D'_{ur}) &= p'_2(D)\end{aligned}$$

Thus, we have

$$(p_1/p_2)(D'_{ur}) = (p'_1/p'_2)(D)$$

B.8 Proof for Predicates

Suppose q_u is an expression like $p_1[p_2]$, where p_1 and p_2 are two path expressions whose levels are less than or equal to k According to the query modification algorithm, the rewritten expression q'_u is $p'_1[p'_2]$, where p'_1 and p'_2 are rewritten expressions of p_1 and p_2 respectively.

According to the definition, the result of $p_1[p_2](D'_{ur})$ is computed in two steps. First, $p_1(D'_{ur})$ is evaluated to generate a node set, say N_1. Then, for each node in N_1, the expression $p_2(D'_{ur})$ is evaluated with that node as the context node; if the $p_2(D'_{ur})$ evaluates to true, then the node is included in the final result set. According to the hypothesis, we have

$$\begin{aligned} p_1(D'_{ur}) &= p'_1(D) \\ p_2(D'_{ur}) &= p'_2(D) \end{aligned}$$

Thus, we have

$$p_1[p_2](D'_{ur}) = p'_1[p'_2](D)$$

B.9 Proof for Logical Operations

XPath 1.0 defines 2 logical operators: the operator `or` and the operator `and`. Since the proof for these 2 logical operators are quite similar, we show one example proof for the `or` operator.

Suppose q_u is an expression like p_1 `or` p_2, where p_1 and p_2 are two path expressions whose levels are less than or equal to k. According to the query modification

algorithm, the rewritten expression q'_u is p'_1 `or` p'_2, where p'_1 and p'_2 are rewritten expressions of p_1 and p_2 respectively.

According to the definition of the operator `||`, the expression $(p_1$ `or` $p_2)(D'_{ur})$ is TRUE if at least one of $\texttt{boolean}(p_1)(D'_{ur})$ and $\texttt{boolean}(p_2)(D'_{ur})$ is TRUE. Similarly, the expression $(p'_1$ `or` $p'_2)(D)$ is TRUE if at least one of $\texttt{boolean}(p'_1)(D)$ and $\texttt{boolean}(p'_2)(D)$ is TRUE.

In the previous proofs, we have shown that

$$\begin{aligned} \texttt{boolean}(p_1)(D'_{ur}) &= \texttt{boolean}(p'_1)(D) \\ \texttt{boolean}(p_2)(D'_{ur}) &= \texttt{boolean}(p'_2)(D) \end{aligned}$$

Obviously, we have

$$(p_1 \texttt{ or } p_2)(D'_{ur}) = (p'_1 \texttt{ or } p'_2)(D)$$

■

Bibliography

[1] D. Ellion Bell and Leonard J. LaPadula. *secure computer systems: mathematical foundations*. `http://citeseer.nj.nec.com/548063.html`, March 1973.

[2] E. Bertino, S. Castano, and E. Ferrari. Securing XML documents with Author-X. *IEEE Internet Computing*, 5(3):21–31, 2001.

[3] Elisa Bertino, Silvana Castano, Elena Ferrari, and Marco Mesiti. Controlled access and dissemination of XML documents. In *Proceedings of the second international workshop on Web information and data management*, pages 22–27. ACM Press, 1999. `http://doi.acm.org/10.1145/319759.319770`.

[4] Elisa Bertino, Sushil Jojodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *IEEE Symposium on Security and Privacy*, pages 94–109, `citeseer.nj.nec.com/article/bertino96supporting.html`, May 1996.

[5] Meadows C. and Landwehr C.E. Designing a trusted application in an object-oriented data model. In *Research Directions in Database Security*, Berlin, 1992. Springer-Verlag.

[6] S. Castano, M. G. Fugini, and P. Samarati. *Database Security*. ACM Press/Addison-Wesley, New York, NY., 1995.

[7] SungRan Cho, Laks V.S. Lakshmanan, Sihem Amer-Yahia, and Divesh Srivastava. Optimizing the secure evaluation of twig queries. In *Proceedings of the 28th VLDB Conference*, Hong Kong, China, 2002.

[8] E. Damiani, S. di Vimercati, S. Paraboschi, and P. Samarati. XML access control systems: a component-based approach. In *In 14th IFIP 11.3 Working Conference in Database Security*, 2000.

[9] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Securing XML documents. In *Advances in Database Technology - EDBT 2000, 7th International Conference on Extending Database Technology, Konstanz, Germany, March 27-31, 2000, Proceedings*, volume 1777, pages 121–135, `http://link.springer.de/link/service/series/0558/bibs/1777/17770121.htm`.

[10] Ernesto Damiani, Sabrina De Capitani di Vimercati, Stefano Paraboschi, and Pierangela Samarati. Design and implementation of an access control processor for XML documents. *WWW9 / Computer Networks*, 33(1-6):59–75, 2000.

[11] L.C. Dion. A complete protection model. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, April 1981.

[12] Bertino E. A view mechanism for object-oriented databases. In *Proc. Int. Conf. on Extending Database Technology (EDBT)*, Vienna, 1992. Springer-Verlag.

[13] E.Bertino, M.Braun, S.Castano, E.Ferrari, and M.Mesiti. Author-X: a Java-based system for XML data protection. In *Proc. of the 14th Annual IFIP WG 11.3 Working Conference on Database Security*, Netherlands, August 2000.

[14] D.E. Denning et al. The sea view security model. In *Proc. IEEE Symp. on Security and Privacy*, Oakland, CA, April 1988.

[15] D. Ferraiolo, R. Sandhu, S. Gavrila, D.R. Kuhn, and R. Chandramouli. A proposed standard for role based access control. *ACM Transactions on Information and System Security*, 4, August 2001.

[16] Fort George and G. Meade. *A guide to understanding discretionary access control in trusted Systems.* `http://www.dsinet.org/textfiles/rainbow-books/neon_orange.html`, September 1987.

[17] G.S. Graham and P.J. Denning. Protection - principles and practice. In *Proc. Spring Joint. Comp. Conf*, page 40. AFIPS Press, 1972.

[18] Satoshi Hada and Michiharu Kudo. *XML access control language: provisional authorization for XML documents.* IBM, `http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html`, October 2000.

[19] Michael A. Harrison, Walter L. Ruzzo, and Jeffrey D. Ullman. Protection in operating systems. *Communications of the ACM*, 19(8):461–471, 1976.

[20] H.V. Jagadish, Divesh Srivastava, Laks V. S. Lakshmanan, and Ting Yu. Compressed accessibility map: Efficient access control for XML. In *Proceedings of*

the 28th VLDB Conference, `citeseer.nj.nec.com/yu02compressed.html`, 2002.

[21] S. Jajodia, P. Samarati, and V.S. Subrahmanian. A logical language for expressing authorizations. In *Proc. IEEE Symposium on Security and Privacy*, 1997.

[22] S. Jajodia, P. Samarati, V.S. Subrahmanian, and Elisa Bertino. A unified framework for enforcing multiple access control policies. In *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*, pages 474–485, May 1997.

[23] Michiharu Kudo and Satoshi Hada. XML document security based on provisional authorization. In *Proceedings of the 7th ACM conference on Computer and communications security*, pages 87–96, `http://doi.acm.org/10.1145/352600.352613`, 2000. ACM Press.

[24] B. W. Lampson. Protection. In *Proceedings of the 5th Princeton Symposium on Information Science and Systems, reprinted in ACM Operating Systems Review*, volume 8, 1974.

[25] John Levine, Tony Mason, and Doug Brown. *lex & yacc*. O'Reilly, second edition, 1992.

[26] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, March 1991.

[27] R. Sandhu, D.F. Ferraiolo, and D.R. Kuhn. The NIST model for role based access control: Towards a unified standard. In *Proceedings, 5th ACM Workshop on Role Based Access Control*, `http://csrc.nist.gov/rbac/`, July 2000.

[28] Ravi S. Sandhu and Pierrangela Samarati. Access control: principles and practice. *IEEE Communications Magazine*, 32(9):40–48, 1994.

[29] World Wide Web Consortium, `http://www.w3.org/TR/xpath`. *XML Path Language (XPath)*, verson 1.0 edition, November 1999.

[30] World Wide Web Consortium, `http://www.w3.org/TR/xslt`. *XSL Transformations (XSLT) Version 1.0*, W3C recommendation edition, November 1999.

[31] World Wide Web Consortium, `http://www.w3.org/TR/REC-xml`. *Extensible Markup Language (XML) 1.0*, second edition, October 2000.

[32] World Wide Web Consortium, `http://www.w3.org/TR/xquery`. *XQuery 1.0: An XML Query Language*, W3C working draft edition, November 2002.

[33] George Kingsley Zipf. *human behavior and the principle of least effort*. Hafner Publishing Company, Inc., 866 Third Avenue, New York. N.Y. 10022, 1972.