Access Control Models for XML

S. De Capitani di Vimercati[1], S. Foresti[1], S. Paraboschi[2], and P. Samarati[1]

[1] University of Milan – 26013 Crema, Italy
{decapita,foresti,samarati}@dti.unimi.it
[2] University of Bergamo – 24044 Dalmine, Italy
parabosc@unibg.it

Summary. XML has become a crucial tool for data storage and exchange. In this chapter, after a brief introduction on the basic structure of XML, we illustrate the most important characteristics of access control models. We then discuss two models for XML documents, pointing out their main characteristics. We finally present other proposals, describing their main features and their innovation compared to the previous two models.

1 Introduction

The amount of information that is made available and exchanged on the Web sites is continuously increasing. A large portion of this information (e.g., data exchanged during EC transactions) is sensitive and needs to be protected. However, granting security requirements through HTML-based information processing turns out to be rather awkward, due to HTML's inherent limitations. HTML provides no clean separation between the structure and the layout of a document and some of its content is only used to specify the document layout. Moreover, site designers often prepare HTML pages according to the needs of a particular browser. Therefore, HTML markup has generally little to do with data semantics.

To the aim of separating data that need to be represented from how they are displayed, the World Wide Web Consortium (W3C) has standardized a new markup language: the *eXtensible Markup Language* (XML) [1]. XML is a markup meta-language providing semantics-aware markup without losing the formatting and rendering capabilities of HTML. XML's tags' capability of self-description is shifting the focus of Web communication from conventional hypertext to data interchange. Although HTML was defined using only a small and basic part of SGML (Standard Generalized Markup Language: ISO 8879), XML is a sophisticated subset of SGML, designed to describe data using arbitrary tags. As its name implies, extensibility is a key feature of XML; users and applications are free to declare and use their own tags and attributes. Therefore, XML ensures that both the logical structure and content

of semantically rich information is retained. XML focuses on the description of information structure and content as opposed to its presentation. Presentation issues are addressed by a separate language, XSL [2] (XML Stylesheet Language), which is also a W3C standard for expressing how XML-based data should be rendered.

Since XML documents can be used instead of traditional relational databases for data storage and organization, it is necessary to think of a security system for XML documents protection. In this chapter, we will focus on access control enforcement. Specifically, in the literature, different access control models have been proposed for protecting data stored in XML documents, exploiting the flexibility offered by the markup language. Even if traditionally access control models can be applied to XML documents, by simply treating them as files, a finer grained access control system is frequently necessary. As a matter of fact, an XML document may contain both sensitive and publicly available information, and it is necessary to distinguish between them when specifying the access control policy.

The remainder of the chapter is organized as follows. Section 2 discusses the basic XML concepts, by introducing DTD, XML Schema, XPath and XQuery syntax and semantics. Section 3 introduces the problem of access control for XML documents, points out the characteristics that an access control model for XML documents should have. Section 4 illustrates in the details two of the first access control models proposed for XML documents, and briefly describes other proposals. Finally, Sect. 5 concludes the chapter.

2 Preliminary Concepts

XML [1] (eXtensible Markup Language) is a markup language developed by the World Wide Web Consortium (W3C) and used for describing semi-structured information. We introduce some of the most important concepts related to XML, which are useful to define an access control system for protecting XML documents.

2.1 Well-Formed and Valid XML Documents

XML document is composed of a sequence of (possibly nested) *elements* and *attributes* associated with them. Basically, elements are delimited by a pair of start and end tags (e.g., <`request`> and </`request`>) or, if they have no content, are composed of an empty tag (e.g., <`request/`>). Attributes represent properties of elements and are included in the start tag of the element with which they are associated (e.g., <`request number=`“10”>). An XML document is said to be *well-formed* if its syntax complies with the rules defined by the W3C consortium [1], which can be summarized as follows:

- the document must start with the *prologue* `<?xml version=`“1.0”`?>`;

- the document must have a *root* element, containing all other elements in the document;
- all open tags must have a corresponding closed tag, provided it is not an empty tag;
- elements must be properly nested;
- tags are case-sensitive;
- attribute values must be quoted.

An *XML language* is a set of XML documents that are characterized by a syntax, which describes the markup tags that the language uses and how they can be combined, together with its semantics. A *schema* is a formal definition of the syntax of an XML language, and is usually expressed through a schema language. The most common schema languages, and on which we focus our attention, are *DTD* and *XML Schema*, both originating from W3C.

Document Type Definition

A DTD document may be either internal or external to an XML document and it is not itself written in the XML notation.

A DTD schema consists of definition of elements, attributes, and other constructs. An element declaration is of the form `<!ELEMENT` *element_name content*`>`, where *element_name* is an element name and *content* is the description of the content of an element and can assume one of the following alternatives:

- the element contains parsable character data (`#PCDATA`);
- the element has no content (`Empty`);
- the element may have any content (`Any`);
- the element contains a group of one or more subelements, which in turn may be composed of other subelements;
- the element contains parsable character data, interleaved with subelements.

When an element contains other elements (i.e., subelements or mixed content), it is necessary to declare the subelements composing it and their organization. Specifically, sequences of elements are separated by a comma "`,`" and alternative elements are separated by a vertical bar "|". Declarations of sequence and choices of subelements need to describe subelements' cardinality. With a notation inspired by extended BNF grammars, "`*`" indicates zero or more occurrences, "`+`" indicates one or more occurrences, "`?`" indicates zero or one occurrence, and no label indicates exactly one occurrence.

An attribute declaration is of the form `<!ATTLIST` *element_name attribute_def*`>`, where *element_name* is the name of an element, and *attribute_def* is a list of attribute definitions that, for each attribute, specify the attribute name, type, and possibly default value. Attributes can be marked as `#REQUIRED`, meaning that they must have an explicit value for each occurrence of the elements with which they are associated; `#IMPLIED`, meaning that

they are optional; `#FIXED`, meaning that they have a fixed value, indicated in the definition itself.

An XML document is said to be *valid* with respect to a DTD if it is syntactically correct according to the DTD. Note that, since elements and attributes defined in a DTD may appear in an XML document zero (optional elements), one, or multiple times, depending on their cardinality constraints, the structure specified by the DTD is not rigid; two distinct XML documents of the same schema may differ in the number and structure of elements.

XML Schema

An XML Schema is an XML document that, with respect to DTD, has a number of advantages. First, an XML Schema is itself an XML document, consequently it can be easily extended for future needs. Furthermore, XML Schemas are richer and more powerful than DTDs, since they provide support for data types and namespaces, which are two of the most significant issues with DTD.

An element declaration specifies an element name together with a simple or complex type. A *simple type* is a set of Unicode strings (e.g., decimal, string, float, and so on) and a *complex type* is a collection of requirements for attributes, subelements, and character data that apply to the elements assigned to that type. Such requirements specify, for example, the *order* in which subelements must appear, and the cardinality of each subelement (in terms of `maxOccurs` and `minOccurs`, with 1 as default value).

Attribute declarations specify the attributes associated with each element and indicate attribute `name` and `type`. Attribute declarations may also specify either a `default` value or a `fixed` value. Attributes can be marked as: `required`, meaning that they must have an explicit value for each occurrence of the elements with which they are associated; `optional`, meaning that they are not necessary.

Example 1. Suppose that we need to define an XML-based language for describing bank account operations. Figure 1(a) illustrates a DTD stating that each `account_operation` contains a `request` element and one or more `operation` elements. Each `account_operation` is also characterized by two mandatory attributes: `bankAccN`, indicating the number of the bank account of the requester; and `id`, identifying the single update. Each `request` element is composed of `date`, `means`, and `notes` elements, where only `date` is required. Element `operation` is instead composed of: `type`, `amount`, `recipient`, and possibly one between `notes` and `value`.

Figure 1(b) illustrates an XML document valid with respect to the DTD in Fig. 1(a).

DTDs and XML documents can be graphically represented as trees.

A DTD is represented as a labeled tree containing a node for each element, attribute, and value associated with `fixed` attributes. To distinguish elements

```
<!DOCTYPE record[
  <!ELEMENT account_operation
        (request, operation+)>
  <!ATTLIST account_operation
        bankAccN CDATA #REQUIRED
        id CDATA #REQUIRED>
  <!ELEMENT request
        (date,means?,notes?)>
  <!ATTLIST request number CDATA #REQUIRED>
  <!ELEMENT operation
        (type, amount, recipient, (notes|value)?)>
  <!ELEMENT date (#PCDATA)>
  <!ELEMENT means (#PCDATA)>
  <!ELEMENT notes (#PCDATA)>
  <!ELEMENT type (#PCDATA)>
  <!ELEMENT amount (#PCDATA)>
  <!ELEMENT recipient (#PCDATA)>
  <!ELEMENT value (#PCDATA)>
]>
```

(a)

```
<?xml version="1.0" ?>
<!DOCTYPE record SYSTEM "record.dtd">
  <account_operation
      bankAccN="0012" id="00025">
      <request number="10">
          <date> 04-20-2007 </date>
          <means> Internet </means>
          <notes> urgent </notes>
      </request>
      <operation>
          <type> bank transfer </type>
          <amount> $ 1,500 </amount>
          <recipient> 0023 </recipient>
          <notes> Invoice 315 of 03-31-2007
          </notes>
      </operation>
  </account_operation>
```

(b)

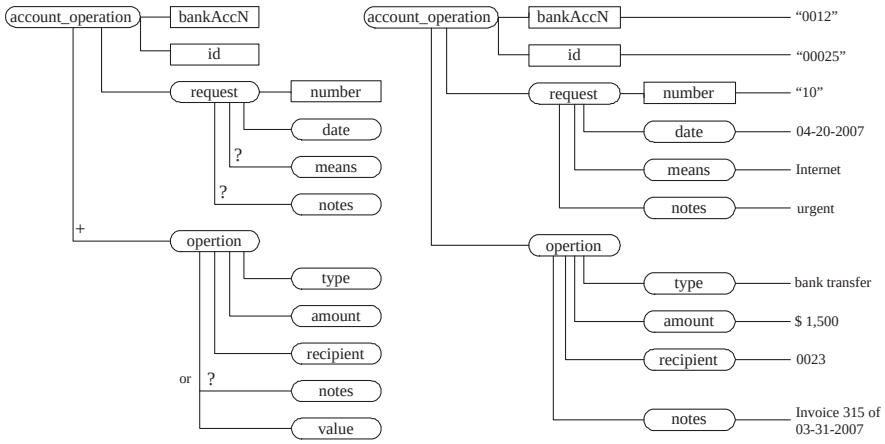
Fig. 1. An example of DTD (a) and a corresponding valid XML document (b)

and attributes in the graphical representation, elements are represented as ovals, while attributes as rectangles. There is an arc in the tree connecting each element with all the elements/attributes belonging to it, and between each `#FIXED` attribute and its value. Arcs connecting an element with its subelements are labeled with the cardinality of the relationship. Arcs labeled *or* and with multiple branching are used to represent a choice in an element declaration (|). An arc with multiple branching is also used to represent a sequence with a cardinality constraint associated with the whole sequence (`?`, `+`, `*`). To preserve the order between elements in a sequence, for any two elements e_i and e_j, if e_j follows e_i in the element declaration, node e_j appears below node e_i in the tree.

Each XML document is represented by a tree with a node for each element, attribute, and value in the document. There is an arc between each element and each of its subelements/attributes/values and between each attribute and each of its value(s). Each arc in the DTD tree may correspond to zero, one, or multiple arcs in the XML document, depending on the cardinality of the corresponding containment relationship. Note that arcs in XML documents are not labeled, as there is no further information that needs representation. Figure 2 illustrates the graphical representation of both DTD and XML document in Fig. 1.

2.2 Elements and Attributes Identification

The majority of the access control models for XML documents identify the objects under protection (i.e., elements and attributes) through the *XPath* language [3]. XPath is an expression language, where the basic building block is the path expression.



(a) DTD tree (b) XML document tree

Fig. 2. An example of graphical representation of DTD and XML document

A *path expression* on a document tree is a sequence of element names or predefined functions separated by character / (slash): $l_1/l_2/\ldots/l_n$. Path expressions may terminate with an attribute name as the last term of the sequence. Attribute names are syntactically distinguished by preceding them with special character `@`. A path expression $l_1/l_2/\ldots/l_n$ on a document tree represents all the attributes or elements named l_n that can be reached by descending the document tree along the sequence of nodes named $l_1/l_2/\ldots/l_{n-1}$. A path expression can be either *absolute*, if it starts from the root of the document (the path expression starts with /); or *relative*, if it starts from a predefined element in the document (the path expression starts with element name). The path expression may also contain operators (e.g., operator `.` represents the current node, operator `..` represents the parent node, operator `//` represents an arbitrary descending path), functions, and predicates (we refer the reader to [3] for more details).

XPath allows the association of conditions with nodes in a path; in this case the path expression identifies the set of nodes that satisfy all the conditions. Conditional expressions in XPath may operate on the "text" of elements (i.e., character data in elements) or on names and values of attributes. A condition is represented by enclosing it within square brackets, following a label l_i in a path expression $l_1/l_2/\ldots/l_n$. The condition is composed of one or more predicates, which may be combined via `and` and `or` boolean operators. Each predicate compares the result of the evaluation of the relative path expression (evaluated at l_i) with a constant or with another expression. Multiple conditional expressions appearing in the same path expression are considered to be `and`ed (i.e., all the conditions must be satisfied). In addition, conditional expressions may include functions `last()` and `position()` that permit

the extraction of the children of a node that are in given positions. Function `last()` evaluates to true on the last child of the current node. Function `position()` evaluates to true on the node in the evaluation context whose position is equal to the context position.

Path expressions are also the building blocks of other languages, such as XQuery [4] that allows to make queries on XML documents through `FLWOR` expressions. A `FLOWR` expression is composed of the following clauses:

- `FOR` declares variables that are iteratively associated with elements in the XML documents, which are identified via path expressions;
- `LET` declares variables associated with the result of a path expression;
- `WHERE` imposes conditions on tuples;
- `ORDER BY` orders the result obtained by `FOR` and `LET` clauses;
- `RETURN` generates the final result returned to the requester.

Example 2. Consider the DTD and the XML document in Example 1. Some examples of path expressions are the following.

- `/account_operation/operation`: returns the content of the `operation` element, child of `account_operation`;
- `/account_operation/@bankAccN`: returns attribute `bankAccN` of element `account_operation`;
- `/account_operation//notes`: returns the content of the `notes` elements, anywhere in the subtree rooted at `account_operation`; in this case, it returns both `/account_operation/request/notes` and `/account_operation/operation/notes`;
- `/account_operation/operation[./type=`"bank transfer"`]`: returns the content of the `operation` element, child of `account_operation`, only if the `type` element, child of `operation`, has value equal to "bank transfer".

The following XQuery extracts form the XML document in Fig. 1(b) all the `account_operation` elements with operation type equal to "bank transfer". For the selected elements, the `amount` and the `recipient` of the operation are returned, along with all `notes` appearing in the selected `account_operation` element.

```
<BankTransf>
{ FOR    $r in document("update_account")/account_operation
  WHERE  $r/operation/type="bank transfer"
  RETURN $r/operation/amount, $r/operation/recipient, $r//notes
}
</BankTransf>
```

3 XML Access Control Requirements

Due to the peculiar characteristics of the XML documents, they cannot be protected by simply adopting traditional access control models, and specific

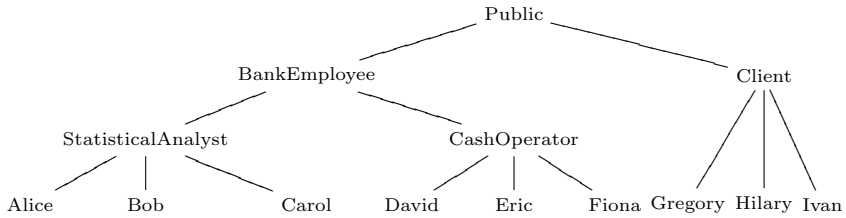


Fig. 3. An example of user-group hierarchy

models need to be defined. By analyzing the existing proposals, it is easy to see that they are all based on the definition of a set of authorizations that at least specify the subjects on which they apply, the objects to be protected, and the action to be executed. The existing XML-based access control models differentiate on the basis of the subjects, objects, and actions they can support for access control specification and enforcement.

- *Subject* Subjects are usually referred to on the basis of their *identities* or of the network *location* from which requests originate. Locations can be expressed with reference to either the numeric IP address (e.g., `150.100.30.8`) or the symbolic name (e.g., `bank.com`) from which the request comes.
  It often happens that the same privilege should be granted to sets of subjects, which share common characteristics, such as the department where they work, or the role played in the company where they work. To the aim of simplifying the authorizations definition, some access control models allow the specification of authorizations having as subject:
  - a *group of users*, which is a statically defined set of users; groups can be nested and overlapping;
  - a *location pattern*, which is an expression identifying a set of physical locations, obtained by using the wild character $*$ in physical or symbolic addresses;
  - a *role*, which is a set of privileges that can be exploited by any user while playing the specific role; users can dynamically decide which role to play, among the ones they are authorized to play.

  Also, subjects are often organized in hierarchies, where an authorization defined for a general subject propagates to its descendants.
  A hierarchy can be pictured as a *directed acyclic graph* containing a node for each element in the hierarchy and an arc from element x to element y, if x directly dominates y. Dominance relationships holding in the hierarchy correspond to paths in the graph. Figure 3 shows an example of user-group hierarchy.
  Recently proposed models [5] for access control on XML documents introduce the possibility of specifying authorizations on the basis of subject's

characteristics, called *credentials*, without even knowing the user's identity and/or location.

- *Object Granularity.* The identification of the object involved in a specific authorization can exploit the possibility given by XML of identifying elements and attributes within a document through path expressions as defined by the XPath language.
  Consequently, XML allows the specification of authorizations at a fine grained level. Any portion of a document that can be referred by a path expression can be the object of an authorization. For instance, a single element or a single attribute are objects as well as a whole XML document. It is important to note that not all models support entirely XPath syntax, since it is very expressive and may be difficult to manage. For instance, some models impose restrictions on the number of times that the // operator can appear in a path expression [6], other proposals do not allow predicates to be specified after the // operator [7].
- *Action.* Most of the proposed XML access control models support only *read* operations, since there is not a standard language for XML update. Furthermore, the management of *write* privileges is a difficult task, which needs to take into account both the access control policy and the DTD (or XML Schema) defined for the document. In fact, the DTD may be partially hidden to the user accessing the document, as some elements/attributes may be denied by the access control policy. For instance, when adding an element to the document, the user may even not be aware of the existence of a required attribute associated with it, as she is not entitled to access the attribute itself.
  However, some approaches try to also support write privileges that are usually classified as: insert operations, update operations, and delete operations.
  In [8], the author proposes to differentiate also read privileges in two categories: the privilege of reading the content of an element, from the privilege of knowing that there is an element in a certain position of the XML document (without knowing the name and content of the element itself). The former authorization class is modeled as `read` action, while the latter is modeled as `position` action. In the same paper, the author proposes also to add the possibility, for the security administrator, to propagate privileges *with-grant option*, as in typical database contexts.

We now discuss the basic peculiar features that are supported by the existing XML-based access control models.

- *Support for Fine and Coarse Authorizations.* The different protection requirements that different documents may have call for the support of access restrictions at the level of each specific document. However, requiring the specification of authorizations for each single document would make the authorization specification task too heavy. The system may then support,

beside authorizations on single documents (or portions of documents), authorizations on collections of documents [9]. The concept of DTD can be naturally exploited to this end, by allowing protection requirements to refer to DTDs or XML documents, where requirements specified at the level of DTD apply to all those documents instance of the considered DTD. Authorizations specified at DTD level are called *schema level* authorizations, while those specified at XML document level are called *instance level* authorizations.

Furthermore, it is important to be able to specify both organization-wide and domain authorizations, which apply only to a part of the whole organization. To this purpose, some systems [9] allow access and protection requirements to be specified both at the level of the enterprize, stating general regulations, and at the level of specific domains where, according to a local policy, additional constraints may need to be enforced or some constraints may need to be relaxed. Organizations specify authorizations with respect to DTDs; domains can specify authorizations with respect to specific documents as well as to DTDs.

- *Propagation Policy.* The structure of an XML document can be exploited by possibly applying different propagation strategies that allow the derivation of authorizations from a given set of authorizations explicitly defined over elements of DTD and/or XML documents. Some proposals therefore distinguish between two kinds of authorizations: *local*, and *recursive* [9]. Local authorizations defined on an element apply to all its attributes only. A recursive authorization defined on an element applies to its whole content (both attributes and subelements). Recursive authorizations represent an easy way for specifying authorizations holding for the whole structured content of an element (for the whole document if the element is the root node of the document).

  The models proposed in [6, 7] assume that negative authorizations are always recursive, while positive authorizations may be either local or recursive.

  Besides downward propagation, upward propagation methods have been introduced [10]. Here, the authorizations associated with a node in the XML tree propagate to all its parents.

  Some of the most common propagation policies (which include also some resolution policies for possible conflicts) are described in the following [11].

  – *No propagation.* Authorizations are not propagated. This is the case of local authorizations.
  – *No overriding.* Authorizations of a node are propagated to its descendants, but they are all kept.
  – *Most specific overrides.* Authorizations of a node are propagated to its descendants, if not overridden. An authorization associated with a

node n overrides a contradicting authorization[3] associated with any ancestor of n for all the descendants of n.
- *Path overrides.* Authorizations of a node are propagated to its descendants, if not overridden. An authorization associated with a node n overrides a contradicting authorization associated with an ancestor n' for all the descendants of n only for the paths passing from n. The overriding has no effect on other paths.

These policies can be adopted also for the authorization subject hierarchy.

- *Support of Exceptions.* The support of authorizations at different granularity levels allows for easy expressiveness of both fine and coarse grained authorizations. Such an advantage would remain however very limited without the ability of the authorization model to support exceptions, since the presence of a granule (document or element/attribute) with protection requirements different from those of its siblings would require the explicit specification of authorizations at that specific granularity level. For instance, the situation where a user should be granted access to all documents associated with a DTD but one specific instance, would imply the need of stating the authorizations explicitly for all the other documents as well; thereby ruling out the advantage of supporting authorizations at the DTD level. A simple way to support exceptions is by using both *positive* (permissions) and *negative* (denials) authorizations, where permissions and denials can override each other.

  The combined use of positive and negative authorizations brings to the problem of how the two specifications should be treated when conflicting authorizations are associated with the same node element for a given subject and action. This requires the support for conflict resolution policies [11].

  Most of the models proposed for XML access control adopt, as a conflict resolution policy, the "denials take precedence" policy, meaning that, in case of conflict, access is denied.

  Note that, when both permissions and denials can be specified, another problem that naturally arises is the *incompleteness* problem, meaning that for some accesses neither a positive nor a negative authorization exists. The incompleteness problem is typically solved by applying a default *open* or *closed* policy [12].

4 XML Access Control Models

Several access control models have been proposed in the literature for regulating access to XML documents. We start our overview of these models by presenting the first access control model for XML [9], which has then inspired

[3] Two authorizations (s, o, a) and (s', o', a') are contradictory if $s = s'$, $o = o'$, and $a = a'$, but one of them grants access, while the other denies it.

many other subsequent proposals. We then illustrate the Kudo et al. [13] model that introduced the idea of using a static analysis system for XML access control. Finally, we briefly describe other approaches that have been studied in the literature to the aim of supporting write privileges and adopting cryptography as a method for access control enforcement.

4.1 Fine Grained XML Access Control System

Damiani et al [9] propose a fine grained XML access control system, which extends the proposals in [14, 15, 16], exploiting XML's own capabilities to define and implement an authorization model for regulating access to XML documents.

We now present the authorizations supported by the access control model and illustrate the authorizations enforcement process.

Authorizations Specification

Access authorization determines the accesses that the system should allow or deny. In this model, access authorizations are defined as follows.

Definition 1 (Access Authorization). *An* access authorization $a \in \mathsf{Auth}$ *is a five-tuple of the form:* ⟨*subject, object, action, sign, type*⟩*, where:*

- subject $\in \mathsf{AS}$ *is the subject for which the authorization is intended;*
- object *is either a URI*$\in\mathsf{Obj}$ *or is of the form URI:PE, where URI*$\in\mathsf{Obj}$ *and PE is a path expression on the tree of document URI;*
- action=`read` *is the action being authorized or forbidden;*
- sign $\in \{+,-\}$ *is the sign of the authorization, which can be positive (allow access) or negative (forbid access);*
- type $\in \{$`LDH`, `RDH`, `L`, `R`, `LD`, `RD`, `LS`, `RS`$\}$ *is the type of the authorization and regulates whether the authorization propagates to other objects and how it interplays with other authorizations (exception policy).*

We now discuss in more detail each of the five elements composing an access authorization.

- *Subject.* This model allows to identify the subject of an authorization by specifying both her identity and her location. This choice provides more expressiveness as it is possible to restrict the subject authorized to access an object on the basis of her identity and of the location from which the request comes.
  Subjects are then characterized by a triple ⟨`user-id`,`IP-address`,`sym-address`⟩, where `user-id` is the identity with which the user connected to the system, and `IP-address` (`sym-address`, respectively) is the numeric (symbolic, respectively) identifier of the machine from which the user connected. The proposed model supports

also *user-groups* and *location patterns* and the corresponding hierarchies. Location patterns are however restricted by imposing that multiple wild characters must be continuous, and that they must always appear as rightmost elements in IP patterns and as leftmost elements in symbolic patterns. As a consequence, location pattern hierarchies are always trees. The user-group hierarchy and the location pattern hierarchies need to be merged in a unique structure: the *authorization subject hierarchy* AS, obtained as Cartesian product of the user-group hierarchy, the IP hierarchy, and the symbolic names hierarchy. Any element in the hierarchy is then associated with a user-id (or group), an IP address (or pattern), and a symbolic name (or pattern). When one of these three values corresponds to the top element in the corresponding hierarchy, the characteristics it defines are not relevant for access control purposes, as any value is allowed.

- *Object.* The set of objects that should be protected is denoted as Obj and is basically a set of URIs (Uniform Resources Identifiers) referring to XML documents or DTDs. Reference to the finer element and attribute grains is supported through path expressions, which are specified in the XPath language.
- *Action.* The authors limit the basic model definition to read authorizations only. However, the support of write actions such as insert, update, and delete does not complicate the authorization model. In [9] the authors briefly introduce a method to handle also write operations, using a model similar to the one proposed for read operations.
- *Sign.* Authorizations can be either positive (permissions) or negative (denials), to provide a simple and effective way to specify authorizations applicable to sets of subjects/objects with support for *exceptions*.
- *Type.* The type defines how the authorizations must be treated with respect to propagation at a finer granularity and overriding.
  Authorizations specified on an element can be defined as applicable to the element's attributes only (*local* authorizations) or, in a recursive approach, to its subelements and their attributes (*recursive* authorizations). To support exceptions (e.g., the whole content, except a specific element, can be read), recursive propagation from a node applies until stopped by an explicit conflicting (i.e., of different sign) authorization on the descendants, following the "most specific overrides" principle. Authorizations can be specified on single XML documents (instance level authorizations) or on DTDs (schema level authorizations). Authorizations specified on a DTD are applicable (i.e., are propagated) to all XML documents that are instances of the DTD. According to the "most specific overrides" principle, schema level authorizations being propagated to an instance are overridden by possible authorizations specified for the instance. To address situations where this precedence criterion should not be applied, the model allows users to specify instance level authorizations as *soft* (i.e., to be applied unless otherwise stated at the schema level) and schema level authorizations

Table 1. Authorization types

	Propagation	
Level/Strength	Local	Recursive
Instance	`L`	`R`
Instance (soft statement)	`LS`	`RS`
DTD	`LD`	`RD`
DTD (hard statement)	`LDH`	`RDH`

as *hard* (i.e., to be applied independently from instance level authorizations). Besides the distinction between instance level and schema level authorizations, this model allows the definition of two types of schema level authorizations: *organization* and *domain* schema level authorizations. Organization schema level authorizations are stated by a central authority and can be used to implement corporate wide access control policies on document classes. Domain schema level authorizations are specified by departmental authorities and describe department policies complementing the corporate ones. For simplicity, these two classes of authorizations are merged by performing a *flat union* (i.e., they are treated in the same way). The combination of the options above (i.e., local vs recursive, schema vs instance level, and soft vs hard authorizations) introduces the eight authorization types summarized in Table 1. Their semantics dictates a priority order among the authorization types. The priority order from the highest to the lowest is: `LDH` (local hard authorization), `RDH` (recursive hard authorization), `L` (local authorization), `R` (recursive authorization), `LD` (local authorization specified at the schema level), `RD` (recursive authorization specified at the schema level), `LS` (local soft authorization), and `RS` (recursive soft authorization).

Access Control Enforcement

Whenever a user makes a request for an object of the system, it is necessary to evaluate which portion of the object (if any) she is allowed to access. To this aim, the system builds a *view* of the document for the requesting subject [9]. The view of a subject on each document depends on the access permissions and denials specified by the authorizations and on their priorities. Such a view can be computed through a *tree labeling* process, followed by a *transformation* process.

Given an access request rq and the requested XML document URI, the tree labeling process considers the tree corresponding to URI and, for each of its nodes, tries to identify if the requesting subject is allowed or denied access. Each node n in the considered tree is associated with a vector $n.veclabel[t]$ that, for each authorization type $t \in$ {`LDH`, `RDH`, `L`, `R`, `LD`, `RD`, `LS`, `RS`}, stores

the users for which there is a positive ($n.veclabel[t].Allowed$) and negative ($n.veclabel[t].Denied$) authorization of type t that applies to n. The algorithm mainly executes the following steps.

- *Step 1: Authorization retrieval.* Determine the set A of authorizations defined for the document URI at the instance and schema levels and applicable to the requester in rq (i.e., the subject of the authorization is the same, or a generalization of the requested subject).
- *Step 2: Initial labeling.* For each authorization $a{=}\langle subject, object, action, sign, type\rangle{\in}\ A$, determine the set N of nodes that are identified by $a.object$. Then, for each node n in N, $a.subject$ is added to the list $n.veclabel[a.type].Allowed$ or to the list $n.veclabel[a.type].Denied$ depending if $a.sign$ is $+$ or $-$, respectively.
  Since several authorizations, possibly of different sign, may exist for each authorization type, the application of a conflict resolution policy is necessary. The final sign $n.veclabel[t].sign$ applicable to node n for each type t is then obtained by combining the two lists according to the selected conflict resolution policy. The model is applicable and adaptable to different conflict resolution policies. However, for simplicity it is assumed that conflicts are solved by applying the "most specific subject takes precedence" principle together with the "denials take precedence" principle.
- *Step 3: Label propagation.* The labels (signs) associated with nodes are then propagated to their subelements and attributes according to the following criteria: (1) authorizations on a node take precedence over those on its ancestors, and (2) authorizations at the instance level, unless declared as soft, take precedence over authorizations at the schema level, unless declared as hard. The nodes whose sign remains undeterminate (ϵ) are associated with a negative sign since the closed policy is applied.
- *Step 4: View computation.* Once the subtree associated with the request has been properly labeled with $+$ $-$ signs, it is necessary to compute the document's view to be returned to the requester. Note that, even if the requester is allowed access to all and only the elements and attributes whose label is positive, the portion of the document visible to the requester includes also start and end tags of elements with a negative label, but that have a descendant with a positive label. Otherwise, the structure of the document would change, becoming non compliant with the DTD any more. The view of the document can be obtained by *pruning* from the original document tree all the subtrees containing only nodes with a negative or undefined label. The pruned document may be not valid with respect to the DTD referenced by the original XML document. This may happen, for instance, when attributes marked as `#REQUIRED` are deleted because the final user cannot access them. To avoid this problem, a *loosening* transformation is applied to the DTD, which simply defines as optional all the elements (and attributes) marked as required in the original DTD. DTD loosening

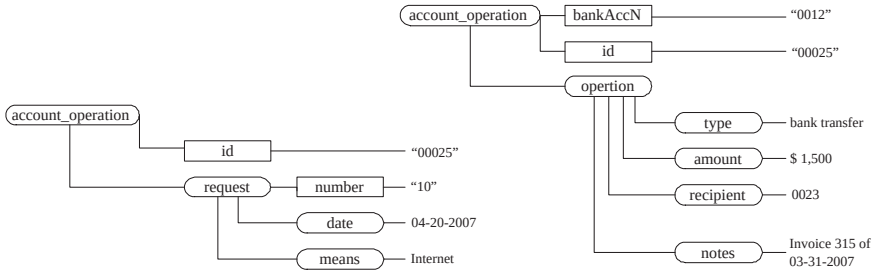
Table 2. An example of access control policies

	Subject	Object	Sign	Action	Type
1	Public,*,*	/account_operation/@bankAccN	−	read	LD
2	BankEmployee,*,*	/account_operation	+	read	RD
3	StatisticalAnalyst,*,*	/account_operation	+	read	RD
4	StatisticalAnalyst,*,*	//notes	−	read	LD
5	StatisticalAnalyst,*,*	/account_operation/operation [./type="bank transfer"]	−	read	RD
6	Client,*,*	/account_operation [./@bankAccN=$userAcc]	+	read	R
7	BankEmployee,150.108.33.*,*	/account_operation/@bankAccN	+	read	L
8	StatisticalAnalyst,*,*.bank.com	/account_operation//notes	+	read	L
9	CashOperators,*,*	/account_operation/ request[./means="Internet"]	−	read	R

prevents users from detecting whether information has been hidden by the security enforcement or was simply missing in the original document [14].

Example 3. Consider the DTD and the XML document in Fig. 1, and the user-group hierarchy in Fig. 3. Table 2 shows a list of access control policies. The first schema-level authorization states that nobody can access attribute `@bankAccN` of element `account_operation` (1). Users belonging to `BankEmployee` and `StatisticalAnalyst` groups can access the `account_operation` element (2 and 3), but `StatisticalAnalyst` group is denied access to `//notes` (4). Since the fourth authorization is `LD`, while third authorization is `RD`, the fourth policy overrides the third one. Furthermore, `StatisticalAnalyst` group is denied access to `/account_operation/operation[./type=`"bank transfer"`]`, meaning that users belonging to the group cannot access `/account_operation/operation` if the operation is a bank transfer (5). Consider now the instance-level authorizations. Users belonging to `Client` group can access the `account_operation` element, if condition `./@bankAccN=$userAcc` holds (variable `$userAcc` represents the variable containing the bank account number for the requesting user) (6). Also, members of the `BankEmployee` group and connected from `150.108.33.*` can access `@bankAccN` attribute (7). This authorization overrides the first authorization in the table. Members of the `StatisticalAnalyst` group and connected from `*.bank.com` can read `/account_operation//notes` for the specific instance (8). Finally, `CashOperators` group is denied access to `/account_operation/request[./means=`"Internet"`]` (9).

Suppose now that Alice and David submit a request to read the document in Fig. 1(b). Figure 4 illustrates the views returned to Alice and David at the end of the access control process.



(a) Alice's view (b) David's view

Fig. 4. Examples of views

4.2 Kudo et al. Static Analysis

Most of the access control systems proposed for XML documents are based on a run-time policy evaluation, that is, any time an access request is submitted to the system, the access control policies are evaluated. However, this run-time policy evaluation may be quite expensive [13]. To avoid this problem, Kudo et al. proposed an access control system based on *static analysis*, which is complemented by a run-time analysis when needed [13].

Authorization Specification

Access authorizations are defined as triples of the form $(s, \pm a, o)$, stating that authorization subject s is (or not, depending on the sign) allowed to perform action a on object o.

An authorization subject may be a user-id, a role, or a group name: the subject name is preceded by a prefix indicating its type. Note that hierarchical structures are not supported by this model. The XPath language is used to define objects in an authorization rule, but functions are not handled by the considered model. Like in [9], the authors limit the basic model definition to read authorizations only, and support both positive and negative authorizations to easily handle exceptions. However, this model does not distinguish between schema and instance level authorizations.

Authorizations specified on an element can be defined as applicable to the element's attributes only (*local* authorizations) or, in a recursive approach, to its subelements and their attributes (*recursive* authorizations). To solve conflicts that may arise on a node, the proposed model can adopt either the "denials take precedence" or the "permissions take precedence" principles, independently from the node on which the conflicting authorizations have been specified. For security reasons, the model presented in the paper limits the analysis to "denials take precedence" principle adoption. The default closed policy is applied when no authorizations are specified.

The framework proposed for static analysis is based on the use of *automata* to compare schemas, authorizations, and queries. The static analysis tries to evaluate anything that does not depend on the specific XML instance and that can be evaluated simply on the basis of the schema and of the access control policy. Formally, an automaton is defined as follows.

Definition 2 (Non deterministic finite state automaton). *A* non deterministic finite state automaton *(NFA) M is a five-tuple of the form $(\Omega, Q, Q^{init}, Q^{fin}, \delta)$ where:*

- *Ω is the alphabet;*
- *Q is a finite set of states of M;*
- *$Q^{init} \subseteq Q$ is the set of* initial *states of M;*
- *$Q^{fin} \subseteq Q$ is the set of* final *states of M;*
- *$\delta : Q \times \Omega \rightarrow Q$ is the* transition function *of M.*

The set of strings accepted by M, denoted $L(M)$, is the language of the automaton.

Given the definition of non deterministic finite state automaton, it is possible to build a NFA corresponding to an arbitrary XPath expression r that does not contain conditions. The NFA accepts a path iff it matches with r. This correspondence is possible since XPath is limited to // operator and conditions are not considered while building the NFA. However, if an XPath expression contains conditions, it is possible to partially capture their semantics by building two NFAs for the given XPath expression r: an *overestimation* $\overline{M[r]}$ and an *underestimation* $\underline{M[r]}$. The former automaton is obtained by assuming all conditions satisfied, while the latter is obtained by assuming all conditions not satisfied.

Static Analysis

The static analysis exploits the definition of automaton and is composed of the following four steps.

- *Step 1: Create schema automata.* Given a schema (DTD or XML Schema) that a document should follow, a *schema automaton* M^G is built. This automaton accepts all and only the paths that are allowed by the schema.
- *Step 2: Create access control automata.* For each role (group) in the system, a pair of automata is defined: an *underestimate access-control automaton* $\underline{M^\Gamma}$ and an *overestimate access-control automaton* $\overline{M^\Gamma}$. For each role, this pair of automata should accept the set of paths to elements and/or attributes that the role is authorized to access. It is necessary to define both an underestimate and an overestimate automaton since conditions may be added to correctly handle the propagation of positive and negative authorizations along the XML tree. In particular, since the "denials take precedence" principle is adopted, an element is accessible only if it is the

descendant of an authorized node, and it is not the descendant of any denied node.

- *Step 3: Create query regular expressions.* Given a query expressed in XQuery, the XPath expressions appearing in the query are translated in equivalent *regular expressions* E^r. XPath expressions appearing as argument for the clauses `FOR`, `LET`, `ORDER`, and `WHERE` are translated in equivalent (possibly overestimated) regular expressions. XPath expressions appearing in the `RETURN` clause are overestimated and the regular expression generated captures also any descendant of the nodes defined by the XPath expression. Note that recursive queries cannot be handled, since the corresponding regular expression would not be defined.
- *Step 4: Compare schema and access control automata with query regular expressions.* Given an XPath expression r, it may be:
  - *always granted*, if every path accepted by the query regular expression E^r and by the schema automaton M^G is accepted by the (underestimated) access control automaton $\underline{M^\Gamma}$;
  - *always denied*, if no path is accepted by all of the query regular expression E^r, the schema automaton M^G, and the (overestimated) access control automaton $\overline{M^\Gamma}$;
  - *statically indeterminated*, otherwise.

  Note that, if the schema is not defined, the schema automaton M^G accepts any path.

The proposed static analysis method does not support conditions involving values specified in the XML documents. However, it is possible to extend the model to the aim of partially handling value-based access control. Intuitively, if an access control policy and a query specify the same predicate, it is possible to incorporate the predicate in the underlying alphabet adopted to build NFAs. To this aim, it is necessary a pre-processing phase of the static analysis method that identifies and substitutes predicates with symbols. Even if this solution does not eliminate predicates completely, it improves query efficiency by anticipating some predicate evaluations.

The main advantage of static analysis is that queries can be rewritten on the basis of the XPath expressions they consider. If the query contains a path expression classified as always denied by the fourth step of the static analysis process, it can be removed from the query without evaluation. By contrast, path expressions classified as always granted, simply need to be returned to the requester. Those path expressions that are classified as statically indeterminate have to be run-time evaluated, on the basis of the specific instance they refer to.

The authors provide also a way for easily building a schema (DTD or XML Schema), which can be released without security threats, depending on the authorizations of the requesting user. This method is based on the automata structures previously described. The *view schema* contains only elements vis-

ible to the final user, while non accessible elements containing accessible ones are renamed as `AccessDenied` elements [13].

As a support for the proposal, experimental results are presented demonstrating the efficiency gain due to static analysis with respect to run-time analysis proposed by other approaches.

Example 4. Consider the DTD and the XML document in Fig. 1 and suppose that there are three user-groups: `BankEmployee`, which are employees of the considered bank institute, `StatisticalAnalyst`, which are bank employees who make statistics about clients and their operations, and `Client`, which are people having a bank account at the institute.

Consider a set of authorizations stating that the members of the `BankEmployee` group can access the whole content of the `account_operation` element, members of the `StatisticalAnalyst` group can access the content of the `account_operation` element but the `notes` elements, and each client can access the `account_operation` elements about their bank account. Formally, these authorizations can be expressed as follows.

- group: `BankEmployee`, `/account_operation`, + `read`, recursive
- group: `StatisticalAnalyst`, `/account_operation`, + `read`, recursive
- group: `StatisticalAnalyst`, `//notes`, - `read`, recursive
- group: `Client`, `/account_operation[./@bankAccN=$userAcc]`, + `read`, recursive

We first define the schema automaton corresponding to the considered DTD. It is first necessary to define two sets of symbols, representing elements and attributes, respectively.

- Σ^E={`account_operation`, `request`, `operation`, `date`, `means`, `notes`, `type`, `amount`, `recipient`, `value`}
- Σ^A={`@bankAccN`, `@Id`, `@number`}

Given Σ^E and Σ^A, it is now possible to define the schema automaton M^G as follows.

- Ω=$\Sigma^E \cup \Sigma^A$
- Q={Account_Operation, Request, Operation, Date, Means, Notes, Type, Amount, Recipient, Value}$\cup\{q^{init}\}\cup\{q^{fin}\}$
- Q^{init}=$\{q^{init}\}$
- Q^{fin}={Date, Means, Notes, Type, Amount, Recipient, Value}$\cup\{q^{fin}\}$
- $\delta(q^{init}$,`account_operation`)=Account_Operation
  δ(Account_Operation,`request`)=Request
  δ(Account_Operation,`operation`)=Operation
  δ(Request,`date`)=Date
  δ(Request,`means`)=Means
  δ(Request,`notes`)=Notes
  δ(Operation,`type`)=Type
  δ(Operation,`amount`)=Amount;

δ(Operation,`recipient`)=Recipient
δ(Operation,`notes`)=Notes;
δ(Operation,`value`)=Value
δ(Account_Operation,`@bankAccN`)=q^{fin}
δ(Account_Operation,`@Id`)=q^{fin}
δ(Request,`@number`)=q^{fin}

The schema automaton defined accepts the same paths allowed by the considered DTD. Specifically, $L(M^G)$ is equal to: `/account_operation`, `/account_operation/@Id`, `/account_operation/@bankAccN`, `/account_operation/request`, `/account_operation/request/@number`, `/account_operation/request/date`, `/account_operation/request/means`, `/account_operation/request/notes`, `/account_operation/operation`, `/account_operation/operation/type`, `/account_operation/operation/amount`, `/account_operation/operation/recipient`, `/account_operation/operation/notes`, `/account_operation/operation/value`.

The second step of the static analysis method consists in building the access control automata $\underline{M^\Gamma}$ and $\overline{M^\Gamma}$, for each of the three groups of users considered. For the sake of simplicity, we represent only the language of the automaton.

- *BankEmployee* $L(M^\Gamma)$=$\{$`account_operation`$\}\cdot(\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})$
- *StatisticalAnalyst* $L(M^\Gamma)$=$\{$`account_operation`$\}\cdot(\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})\setminus$ $\{$`notes`$\}\cdot(\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})$
- *Client* $L(\underline{M^\Gamma})=\emptyset\cdot(\Sigma^E)^*\cdot(\Sigma^A\cup\{\epsilon\})$; $L(\overline{M^\Gamma})$=$\{$account_operation$\}\cdot(\Sigma^E)^*\cdot$ $(\Sigma^A \cup \{\epsilon\})$

Here, $\cdot$ is the concatenation operator, $\setminus$ is the set difference operator, ϵ is the nil character, and $(\Sigma^E)^*$ represents any string in Σ^E.

Consider now the XQuery expression introduced in Example 2. The corresponding XPath expressions, classified on the basis of the clause they are represented in, are:

- `FOR, LET, ORDER BY ,WHERE`: `/account_operation`; `/account_operation/operation/type`
- `RETURN`: `account_operation/operation/amount`; `account_operation/operation/recipient`; `account_operation//notes`

Here `record//notes` implies both `record/request/notes` and `record/operation/notes`.

On the basis of the static analysis, it is possible to classify the requests submitted by users. As an example, consider the following requests.

- `BankEmployee` requests `/account_operation/operation/type`: the request is *always granted*;

- `StatisticalAnalyst` requests `/account_operation//notes`: the request is *always denied*;
- `Client` requests `/account_operation/operation/amount`: the request is *statically indeterminate*.

The last request introduced by the example is statically indeterminate as the path expression `/account_operation[./@bankAccN=$userAcc]` in the access control policy cannot be statically captured by an automaton. To solve this problem, it is possible to rewrite the policy, and all the statical analysis tools, adding two new symbols to the considered alphabet:
`account_operation1=/account_operation[./@bankAccN=$userAcc]` and
`account_operation2=/account_operation[not ./@bankAccN=$userAcc]`.

4.3 Other Approaches

Besides the two access control models described above, a number of other models have been introduced in the literature for controlling access to XML documents.

The first work of Kudo et al. [10] introduce provisional authorizations in XML access control. A *provisional authorization* is an authorization allowing the specification of a security action that the user (and/or the system) has to execute to gain access to the requested resource. A security action may be for example, the encryption of a resource with a given key, or the recording in the log of an access control decision. Due to the problem of run-time policy evaluation, Kudo et al. [6] present a different access control model, based on the definition of an *Access-Condition-Table* (ACT). An ACT structure is statically generated from an access control policy. The ACT contains, for each target path in the XML document, an access condition and a subtree access conditions, which are the conditions that have to be fulfilled to gain access to the node and to its subtree, respectively. By using the ACT, the run-time evaluation of requests is reduced from the whole policy to an access condition. The proposed model has however some disadvantages: it does not scale well, and it imposes limitations on XPath expressions. To overcome these issues the authors propose an alternative structure to ACT, the *Policy Matching Tree* (PMT) [7], which supports real-time updates of both policy and data. In this case, the pre-processing phase consists in building the tree structure on the basis of the access control policy. Whenever a user makes a request, an algorithm visits the path in the tree that matches the request, to compute the correct answer stored in the leaf. To further improve computational efficiency, the authors propose a *function-based* access control model that has a rule function for each authorization in the policy [17]. A rule function is a piece of executable code, which is run any time an access request matches with the rule, and returning the answer for the final user. Function rules can be organized on the basis of the subject or object they refer to: the first solution has been empirically proven to be more efficient.

An alternative solution to the static analysis proposed by Kudo et al. is presented in [18], where the authors propose to store the access control policy in a space and time efficient data structure, called *compressed accessibility map* (CAM). This structure is obtained by exploiting the structural locality of access authorizations, that is, by grouping object having similar access profiles.

Another model proposing a pre-processing phase for access control purposes is introduced in [19], where the pre-processing algorithm, called *QFilter* rewrites these queries by pruning any part that violate access control rules.

The concept of view as the portion of an XML document that can be released to the user (introduced first by Damiani et al. [9]) has been exploited by different models.

The solution proposed by Fan et al. [20] is based on the concept of *security view*. A security view of an XML document provides with each user group both a view of the XML document with all and only the information that the group can access, and a view of the DTD, compliant with the released portion of the XML document. It is important to note that, concretely, each document has one security view, obtained by marking the XML document according with the access control policy. Authorized users are then supposed to make queries over their security view. In the paper, the authors propose both an algorithm for computing security views from an access control policy, and an algorithm for reformulating queries posed on security views to be evaluated on the whole XML document, avoiding materialization.

An alternative method for view generation has also been proposed [21]. This model uses an *authorization sheet* to collect all the authorizations. The authorization sheet is then translated in an XSLT sheet, which grants the generation of the correct view to the user when she asks for (a portion of) the document.

In [22] the authors propose an alternative method to the tree labeling process for view generation, since it may be inefficient if the size of the tree and the number of requests increase. The alternative model stores XML documents in a relational database, which is used to select data on users' request, and to check only selected data against the access control policy, instead of labeling the whole XML tree.

Bertino et al. proposed different works aimed at access control enforcement in XML documents [23, 24, 25, 5]. In particular, they propose a model supporting the use of *credentials* (i.e., sets of attributes concerning a specific user) for subject definition.

Since XML documents represent an alternative to the traditional relational database model, some models adopt solutions proposed for relational databases [8, 26]. In [8] the author proposes to adopt SQL syntax and semantics to XML documents. Each user manages all privileges on her files, and `grants` or `revokes` them to other users, possibly along with the `grant option`.

The model proposed in [26] does not use SQL syntax, but exploits the concept of view as in relational databases to restrict access to data. In this case, views are defined by using the XQuery language, and may be authorization objects. The model supports not only structure-based authorizations, but also rules depending on the context or content of the considered documents by adding conditions in XQuery expressions.

Since relationship among elements/attributes may reveal sensitive information, in [27] the authors propose the definition of access control rules on the relationship among XML elements and attributes (i.e., on arcs in the XML tree). It is then presented a technique to control the view that can be released of the *path* leading to any authorized node in an XML document. The authors introduce also a rule-based formulation of the new class of authorizations.

To the aim of adding semantic meaning to authorizations, RDF (Resource Description Framework) is used as a new way for expressing access control policies [28]. The paper focuses also on the problem of controlling *data associations*, and adds a new object type to the classical model: the association security object. An association security object is an XML subtree whose elements can be accessed only separately. To solve the problem of data associations, the model uses temporal data.

All the models introduced above for access control of XML documents are based on the discretionary access control model [12]. In [29], the authors propose a role-based access control model (RBAC) for XML documents, which exploits the main characteristics of XML data.

In [30] the authors propose the first access control model for XML documents operating client-side. The main difference with respect to the previous proposals is that this method needs to operate on stream data and it is supposed to operate on a system where the server storing data may not be trusted for access control enforcement.

Recently, a new class of methods have been also proposed for access control enforcement for XML documents [5, 31, 32]. These methods consider a data outsourcing scenario, where XML documents are stored on a possibly not trusted server, and are not under the data owner's direct control. In these cases, XML documents themselves should enforce access control, since this task cannot either be executed by the owner or by the storing server. Access control is enforced through selective encryption, that is, by encrypting different portions of the XML tree by using different encryption keys. Consequently, a correct key distribution to users ensures that access control enforcement is correct.

5 Conclusions

The role of XML in the representation and processing of information in current information systems is already significant and is certainly going to see a considerable increase in the next years. The design and implementation of

an access control model for XML promises to become an important tool for the construction of modern applications. The research of the last few years presented in this chapter has produced several proposals for the construction of an access control solution for XML data. These results are a robust basis for the work of a standard committee operating within one of the important consortia involved in the definition of Web standards. Thanks to the availability of such a standard, it is reasonable to expect that XML access control models will be used to support the data protection requirements of many applications, making XML access control a common tool supporting the design of generic software systems.

Acknowledgements

This work was supported in part by the European Union under contract IST-2002-507591, and by the Italian Ministry of Research, within programs FIRB, under project "RBNE05FKZ2", and PRIN 2006, under project "Basi di dati crittografate" (2006099978).

References

1. Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F.: Extensible markup language (XML) 1.0 (fourth edition) (August 2006) W3C Recommendation.
2. Berglund, A.: Extensible stylesheet language (XSL) version 1.1 (December 2006) W3C Recommendation.
3. Clark, J., DeRose, S.: XML path language (XPath) version 1.0 (November 1999) W3C Recommendation.
4. Boag, S., Chamberlin, D., Fernndez, M.F., Florescu, D., Robie, J., Simon, J.: XQuery 1.0: An XML query language (January 2007) W3C Recommendation.
5. Bertino, E., Ferrari, E.: Secure and selective dissemination of XML documents. ACM Transaction Information System Security **5**(3) (August 2002) 290–331
6. Qi, N., Kudo, M.: Access-condition-table-driven access control for XML databases. In: Proc. of the 9th European Symposium on Research in Computer Security, Sophia Antipolis, France (September 2004)
7. Qi, N., Kudo, M.: XML access control with policy matching tree. In: Proc. of the 10th European Symposium on Research in Computer Security, Milan, Italy (September 2005)
8. Gabillon, A.: An authorization model for XML databases. In: Proc. of the 2004 Workshop on Secure Web Service (SWS04), Fairfax, Virginia (November 2004)
9. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: A fine-grained access control system for XML documents. ACM Transaction Information System Security **5**(2) (May 2002) 169–202
10. Kudo, M., Hada, S.: Xml document security based on provisional authorization. In: Proc. of the 7th ACM Conference on Computer and Communications Security (CCS00). (November 2000)

11. Jajodia, S., Samarati, P., Sapino, M., Subrahmanian, V.: Flexible support for multiple access control policies. ACM Transactions on Database Systems **26**(2) (June 2001) 214–260
12. Samarati, P., di Vimercati, S.D.C.: Access control: Policies, models, and mechanisms. In Focardi, R., Gorrieri, R., eds.: Foundations of Security Analysis and Design. LNCS 2171. Springer-Verlag (2001)
13. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML access control using static analysis. ACM Transaction Information System Security **9**(3) (August 2006) 292–324
14. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: Design and implementation of an access control processor for XML documents. Computer Networks **33**(1-6) (June 2000) 59–75
15. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: Securing XML documents. In: Proc. of the 7th International Conference on Extending Database Technology (EDBT00), Konstanz, Germany (March 2000)
16. Damiani, E., Samarati, P., De Capitani di Vimercati, S., Paraboschi, S.: Controlling access to XML documents. IEEE Internet Computing **5**(6) (November/December 2001) 18–28
17. Qi, N., Kudo, M., Myllymaki, J., Pirahesh, H.: A function-based access control model for XML databases. In: Proc. of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany (October - November 2005)
18. Yu, T., Srivastava, D., Lakshmanan, L.V.S., Jagadish, H.V.: Compressed accessibility map: Efficient access control for XML. In: Proc. of the 28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China (August 2002)
19. Luo, B., Lee, D., Lee, W.C., Liu, P.: QFilter: fine-grained run-time XML access control via NFA-based query rewriting. In: Proc. of the 2004 ACM CIKM International Conference on Information and Knowledge Management, Washington, DC, USA (November 2004)
20. Fan, W., Chan, C.Y., Garofalakis, M.: Secure XML querying with security views. In: Proc. of the 2004 ACM SIGMOD International Conference on Management of Data, Paris, France (June 2004)
21. Gabillon, A., Bruno, E.: Regulating access to XML documents. In: Proc. of the Fifteenth Annual Working Conference on Database and Application Security (Das01), Niagara, Ontario, Canada (July 2002)
22. Tan, K.L., Lee, M.L., Wang, Y.: Access control of XML documents in relational database systems. In: Proc. of the 2001 International Conference on Internet Computing, Las Vegas, Nevada, USA (June 2001)
23. Bertino, E., Braun, M., Castano, S., Ferrari, E., Mesiti, M.: Author-X: A Java-based system for XML data protection. In: Proc. of the IFIP TC11/ WG11.3 Fourteenth Annual Working Conference on Database Security, Amsterdam, The Netherlands (August 2000)
24. Bertino, E., Castano, S., Ferrari, E.: Securing XML documents with Author-X. IEEE Internet Computing **5**(3) (May/June 2001) 21–31
25. Bertino, E., Castano, S., Ferrari, E., Mesiti, M.: Specifying and enforcing access control policies for XML document sources. World Wide Web **3**(3) (June 2000) 139–151

26. Goel, S.K., Clifton, C., Rosenthal, A.: Derived access control specification for XML. In: Proc. of the 2003 ACM Workshop on XML Security (XMLSEC-03), New York (October 2003)
27. Finance, B., Medjdoub, S., Pucheral, P.: The case for access control on XML relationships. In: Proc. of the 2005 ACM CIKM International Conference on Information and Knowledge Management, Bremen, Germany (October - November 2005)
28. Gowadia, V., Farkas, C.: RDF metadata for XML access control. In: Proc. of the 2003 ACM Workshop on XML Security (XMLSEC-03), New York (October 2003)
29. Hitchens, M., Varadharajan, V.: RBAC for XML document stores. In: Proc. of the Third International Conference on Information and Communications Security (ICICS01), Xian, China (November 2001)
30. Bouganim, L., Ngoc, F.D., Pucheral, P.: Client-based access control management for XML documents. In: Proc of the 30th VLDB Conference, Tornoto, Canada (September 2004)
31. Miklau, G., Suciu, D.: Controlling access to published data using cryptography. In: Proc. of the 29th VLDB Conference, Berlin, Germany (September 2003)
32. Wang, H., Lakshmanan, L.V.S.: Efficient secure query evaluation over encrypted XML databases. In: Proc. of the 32nd VLDB Conference, Seoul, Korea (September 2006)