CERIAS Tech Report 2003-48
Derived access control specification for XML
by Christopher Clifton
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

Derived Access Control Specification for XML

Siddhartha K. Goel Chris Clifton
Purdue University
Department of Computer Sciences
250 N University St
West Lafayette, IN 47907-2066
{skgoel, clifton}@cs.purdue.edu

Arnon Rosenthal
The MITRE Corporation, M/S K308
202 Burlington Rd
Bedford, MA 01730-1420
arnie@mitre.org

ABSTRACT

The growth in interchange of business and other sensitive data has led to increasing interest in access control. While broad-based access control may be adequate for library-style document bases, new applications demand different access rights on different documents, or different parts of a document. Methods have been developed that enforce fine-grained access control in XML, but the administrative complexity of hard-coding rules is still a challenge. We present an XQuery-based approach for deriving access control rules from schema-level rules, document or database content, or rules on other documents. This approach provides a novel capability to exploit non-structural information in broadly-applicable rules, making it feasible to specify data- and context-dependent rules for large document sets.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration — *Security, integrity, and protection*; H.2.3 [**Database Management**]: Languages—*Query Languages*; H.2.4 [**Database Management**]: Systems—*Textual databases*

General Terms

Security

Keywords

Access Control, XML

1. INTRODUCTION

Access control for XML is receiving significant attention as XML is gaining popularity for storing and exchanging information. Our work extends ideas from SQL security, and from researchers in XML security, to produce a more general mechanism for specifying access control for XML document bases.

XML-based approaches (a brief survey is given in Section 2) give a choice: either define a policy for each document, or (more feasibly) define policy in terms of the schema. Our concern is that while rules on schemas let administrators assign privileges on *parts* of documents, they do not give the ability to specify context-sensitive restrictions on specific documents or fragments. Hard-coding rules for each document or sub-document is supported, but administratively intractable. We explore how one can provide additional useful flexibility by allowing more general expressions, based on XQuery, to define the privileges granted. This allows schema level definitions that give a user different privileges on different documents, based on the content of documents and privileges already defined. Another advantage of using XQuery is that many of the constructs proposed for access control subsystems become unnecessary – they overlap (and should be provided by) the query language. We also explore the utility of making context information and the access policy itself available as queryable data (with restrictions, to avoid circularity). [1]

To demonstrate how this approach eases the administration of fine-grained access control rules, consider the following scenario. A data repository has maintenance records, technical manuals, parts orders, etc. for a fleet of naval ships. The technical staff continuously adds inspection and repair records to this repository. This is an invaluable knowledge store – shared information can aid technicians in fixing problems, provide predictive information on failures, and help improve maintenance procedures. However, much of the information is sensitive or classified. It is impractical to define a new policy for each document – we need broad rules. But rules that list particular nodes from a schema are not sufficiently flexible – there are several other factors that must be considered.

Suppose we want to say that technicians have read and write privileges to maintenance records of a part on their ship if they have the right to read the manual for that part. Instead of coding rules for each maintenance record and each part, we enable a single rule

[1]Ideally, this facility would span both SQL and XML data.

ACM Workshop on XML Security, October 31, 2003, Fairfax VA, USA.

on the maintenance record schema that captures precisely this policy, deriving the privileges for individual maintenance records based on a query determining if the technician has read access to the corresponding manual (and if the maintenance record applies to a part on that ship.) Furthermore, we want the capability to sanitize the result, e.g., (with the user's knowledge) to return only the parts of the document describing the variant of the part in use on the ship – a subset of the entire maintenance document. This can mean omitting nodes or field values. This models the common practice in the DOD of blacking out sections of documents, e.g., paragraphs.

The maintenance records are valuable to technicians elsewhere who work on the same equipment, but in some cases they may contain information that is too sensitive to reveal. For example, knowledge that a ship's anti-missile defense is out of order should be known only to a small circle. While an access control rule for normal maintenance records can make them available to many technicians, a rule for records showing equipment failure must be more restrictive. The access control rule could specify that on critical equipment, failure records are accessible to outsiders only if there is a corresponding record showing repair complete.

From SQL, one can borrow the idea of using an arbitrary query to define a view, and granting some users access to that view. Assuming the view can include the variable $User, the view can include different tuples for each user. However, this requires that applications be written against the view, i.e., it is not transparent to applications. Oracle thus provides policy functions that modify an existing view at run time, allowing arbitrary restriction predicates to be added to the Where clause. The basic ideas - using a query to govern the access rights - can be carried over into XML.

This paper describes how a generalized version of this Oracle capability can be supported for access control on XML, working within the framework of existing XML tools. (One extension will be needed – access rules must be made queryable, like ordinary data). Our contribution enables such high-level specification of flexible fine-grained access control. In the above scenario, even "right to read a particular manual" can be specified in terms of data content, for large sets of documents. A likely policy would be that the manual must describe equipment found on the ship, and the technician must have passed a training course for the equipment – both captured as a query on the data content.

Rules will have the power of XQuery to derive different access rights for different documents. The derivations can reference many kinds of information:

- data values in the requested document,
- data values in other documents,
- environmental information available to XQuery (e.g., $User or $Time from an application server or operating system),
- Other privileges possessed by this user, or by arbitrary users.

Previous work mostly expressed these rules in terms of schemas, or in limited query languages that did not provide sufficient flexibility for our examples.

In Section 3 we illustrate how we express derived access control rules, describing a language that builds on XQuery and a discussion how this enables our desired level of policy specification. Section 4 illustrates how privacy policies in a medical records database could be enforced using rules specified in this language. We discuss how this could be implemented using an XQuery parser and existing fine-grained access control methods in Section 5.2. We describe a prototype we developed in that shows we can capture this power entirely in XQuery, along with some pre- and post-processing of the XML, in Section 5. We also describe how this would be handled in a real world implementation. We conclude with a discussion of future work in Section 6.

2. DEFINITIONS AND RELATED WORK

We first introduce some important terms from access control theory as used in this paper:

subject: An entity that may access objects (e.g., a user or group).

object: A document or part(s) of the document(s) (XML Fragment).

access-right: A particular type of operation that the Subject may perform on the Object. I.e., view tag set/read contents/delete contents/append to existing contents/overwrite contents.

privilege: A triple allowing a subject to perform an action on an object. This is the most basic kind of rule.

Objects affected by a privilege are described by an XPath expression that generates/returns the elements in the appropriate fragment(s).

request: An attempt by a subject to perform an action on a particular object that requires a particular access-right. I.e., If a corresponding privilege exists, the action can occur.

From these we get the concept of an access control rule.

rule: A set of privileges (the trivial case), or a function that produces a set of privileges on an object (denoted O). Its inputs may include a reference to O, and to any other database and context information that is available to XQuery requests.

A rule is *ordinary* or *derived*.

ordinary rule: A rule function that does not reference other rules.

derived rule: A rule whose function does reference other rules. The referencing must be noncircular, with respect to other derivation rules. (It is always permissible to reference privileges that were explicitly granted by administrators. More generally, if the

rule set is stratified,[2] it can also reference privileges generated by rules in lower strata.

rule evaluation: For any request, the rule evaluation on that request is given by: For each object on which access is requested, each rule is evaluated (conceptually) at the time of the request (and more precisely, within the environment of the request transaction.) It returns privileges for the request's subject, indicating what should/should not be allowed, by that rule.

access policy: A set of rules, intended to be applied to requests. The semantics are to evaluate rules in the policy, and see if they give privileges for every access made by the user request. That is: *If for each access in the request, some rule evaluates to OK, the request is approved.*

There have been numerous approaches to defining and enforcing access rights on XML documents. Initially, access control for markup languages like HTML was specified by physically embedding security related tags in the document. This approach does not identify or exploit semi-structured semantics.

More recent work allows different access control for different parts of documents, as well as flexibility for enforcing more complex policies. Kudo and Hada's XACL [11] is an access control policy specification language oriented around <*object, subject, action*> triplets (privileges). Fine grained access decisions aren't simply binary (allow or deny access). XACL allows more flexible provisional authorization to a document based on whether certain conditions are met; e.g., the subject allows accesses to be logged, signs an agreement, etc.

Bertino et al. [2, 1] defined access rules at the schema level that apply to all documents conforming to the schema. They define read element/attribute, navigate document, modify/delete contents of element/attribute and add/modify/delete element/attribute. Their Author-X system is a suite of tools focusing on access-control enforcement and security administration for XML documents. Our work is on security policy specification: high-level rules that generate the specific access privileges that a system such as Author-X would enforce.

Damiani et al., also specify a language for encoding access restrictions [4, 3, 5, 6]. Rules in the specification language can be defined for DTDs/schemas (applicable to all the documents that conform to the schema) or individual XML documents. A rule essentially is the five-tuple <*subject, object, action, sign, prop*>, allowing both negative privileges (with conflict resolution) and propagation to subtrees.

Gabillon and Bruno [8] add numeric priority to resolve conflicts when multiple rules apply to an object. They implement access control by converting their "authorization sheet" to an XSLT document that can then extract a view of the accessible part of the corresponding XML document.

XACML (eXtensible Access Control Markup Language) [12] is an OASIS specification that is gaining acceptance for expressing access control policy for XML. XACML is based on work including that of Kudo, Damiani, and Bertino. It standardizes access request/response format, architecture of the policy enforcement framework, etc., but it does not address deriving access control rules from the existing policy base. These approaches used the limited power query languages (XPath, XSL) then available for prototyping. The query languages were used only to specify which documents (or fragments) a privilege applied to; custom constructs were developed to specify conditions. Supporting the examples we have given requires hard-coded customized rules for each document (or fragment); rules at the DTD/schema level give similar privileges for all conforming documents. XQuery will allow much more flexibility in specifying conditions. This enables reuse of the training, GUIs, and implementation of the mainstream query languages; the security system need not provide special constructs for specifying portions of subtrees, nor predicates that range outside a given subtree. We remain neutral in the controversies about negative privileges and conflict resolution. If desired, they can be provided as part of policy semantics, included in the privileges derived from our more powerful rules.

Several authors have examined issues relevant to implementation.

Jagadish et al. [10] present a space efficient accessibility map that identifies the XML data items a user has access to by exploiting structural locality of accesses in tree-structured data and a time efficient map lookup algorithm. Their work is not on specification of access rules but about deciding accessibility given a set of access and conflict resolution rules.

Vimercati's authorization models for time-varying XML documents[7] show how one can precompute some of the privileges, as database contents change. In our approach, privileges are derived data, to be kept consistent with its sources. By this perspective, one should use a general purpose maintenance system, rather than one created just for access rules.

We concentrate on the specification of access control rather than how to enforce it.

3. DETAILS OF OUR METHOD

Semantically, a rule allows access to a fragment, determined by evaluating the function when the request is submitted. The policy takes the union of all these fragment privileges, and allows the request if this union suffices. The power we add comes from the ability to derive privileges from information beyond what XPath could use.

Our approach involves three parts:

1. Using XQueries to return a set of privileges
2. Providing a predicate access(subject, object) that returns the subject's access-rights on the object.

[2]We assume that the rule set is partitioned, and blocks of the partition (called strata) are partially ordered. Basic privileges that administrators define explicitly are in the lowest stratum. By default, all other rules fall into one stratum – but we hope to allow greater generality.

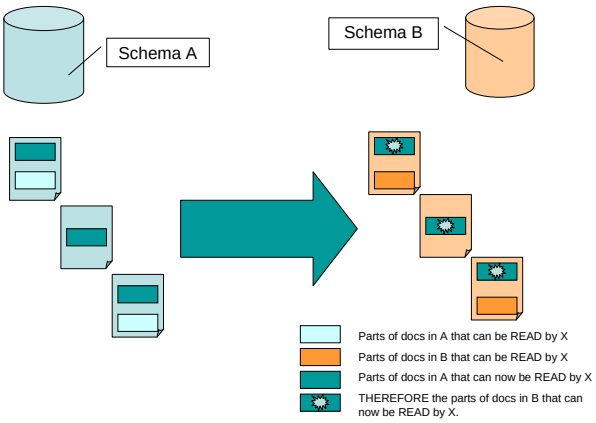

Figure 1: Documents with transitive access control relationship

3. Providing a functionality to recursively derive access based on other derivable privileges.

We start with the an XACL privilege triple:<*Object*, *Subject*, *Access-right*> [11]. The specification of *Object* is an XPath – a document or document fragment – as defined above. *Subject* is simply an identifier. *Access-right* is one of view_tag_set, delete, read, append, or overwrite. We assume a mechanism exists to match requests against privileges, e.g., one of the systems described in the preceding section. (It is possible to incorporate additional features beyond those supported by XACL into privileges, such as negative permissions, but this would unnecessarily complicate the discussion.) In practice the access control rules could be embedded in an XML document (making the *object* specification implicit), however for clarity we treat them as an external "privilege set".

The basic information used in a rule is:

1. A query defining the set of objects to which the Rule applies, the same as a basic privilege;

2. A set of objects from which the access is derived, identified by a function whose inputs may (as listed in Section 2) include the object to which access is being requested and other information available to the query.

3. A function returning an access right; its potential inputs are the same as in the previous item.

   In the case of a derived (as opposed to ordinary) rule, the function may also be based on the access that the user has to the objects in (2).

Note that a user need not be granted the same rights on all objects specified in (1). The objects in (2) may change depending on the contents of the requested object. The function in (3) will thus operate on different data (from both (1) and (2)), giving access customized to the particular user and request.

Consider Figure 1. The documents conforming to schema B derive access control from the policy base on schema A. Note only one rule is required for schema B to derive access from documents conforming to schema A, even though user X obtains different rights on each document in schema B. Each document conforming to schema B derives a different access control policy based on its own content and structure and the access control of the corresponding document(s) conforming to schema A. To achieve the same effect from previous work one needs to define access controls separately for each document (administratively intractable), since a definition on the schema would provide the same rights on all documents or document parts conforming to the schema.

Changes in the data, or in such environmental information as the system time, may change the privileges that can be derived from a rule. The access granted by the rule is dynamic, even if the rule (and entire policy) is statically defined. This leads to some interesting implementation considerations, discussed in Section 5.2.

First we describe the language for defining a rule. Although we speak of a rule-specification language, policy specification rules can actually be specified entirely in terms of XQuery and XML. This enables privileges to be generated by XQuery, with some pre- and post-processing. We discuss this further in Section 5. However, we first present the rule-specification language as a modification of XQuery; we feel this gives a clearer view of the concepts and the power of the method.

An example rule-specification is:

```
let $hospital =
    document("http://www.example.com/hospital.xml")
let $office =
    document("http://www.example.com/office.xml")

for $a in $hospital/PatientRecords/Patient/Medical return
    for $b in $office/Staff/Employee return
        for $c in $office/Staff/Employee
            where $b/@Name = $a/Doctor and
                  $c/@Name = $b//AccountableTo
                return <$c/@Name, $a,
                        access($b/@Name, $a)>
```

This captures the policy: *A supervisor should have the access permissions to the records of patients of doctors they supervise that the supervised doctor does.* This example will be worked out in more detail in Section 4, but for now we use this to point out the basic structure of a rule. The first part identifies the relevant documents – in this example the first line identifies the document on which privileges will be given, and the second a document containing information used in deriving the privileges. The next part binds variables to specific document fragments. The third line iterates over all patient medical records. The fourth and fifth iterate over all hospital employees. The sixth line binds $b to the patient's doctor, and $c to the doctor's supervisor. Finally, the last line generates a privilege giving $c the same access on the record $a as that enjoyed by the patient's doctor $b.

The key difference is that instead of returning a document or document fragment, the query returns a set of privilege triples. While we have not shown the syntactic

details, in Section 5 we describe how to represent these triples in an XML document. A special predicate is also used: **access(Subject, Object)**. This returns the maximum access-right that *Subject* has for *Object*. Intersection, union, addition, exception operations, value based comparisons, and other XQuery operations can be performed on access-rights, allowing rights to be based on complex functions of other rights, as well as data.

We assume support for enforcing access control from the work above, i.e., given a set of privileges specifying access control, implementations by the groups above will enforce access control in the corresponding XML. (We expand on this in Section 5.2.) We concentrate on providing the administrative ability to program these rules once, instead of specifying them for each document conforming to schema B (Figure 1) as they are added to the repository.

4. FULL EXAMPLE

We now give a detailed example based on a medical record scenario. All examples use two XML documents: office.xml (Figure 2) storing data about hospital staff, and hospital.xml (Figure 3) storing patient records for the hospital.

We first demonstrate how the previously described rule – *supervisor has any access to a supervisee's patients records a supervisee does* – works:

```
let $hospital =
    document("http://www.example.com/hospital.xml")
let $office =
    document("http://www.example.com/office.xml")

for $a in $hospital/PatientRecords/Patient/Medical return
    for $b in $office/Staff/Employee return
        for $c in $office/Staff/Employee
            where $b/@Name = $a/Doctor and
                  $c/@Name = $b//AccountableTo
                  return <$c/@Name, $a,
                           access($b/@Name, $a)>
```

The tuple <*subject, object, access-right*> is a privilege that implies that the *subject* specified has privileges equivalent to *access-right* to access the *object* in the rule. The referenced previous work essentially expresses access control for XML as more complex versions of the above privilege tuple. Here, *access()* is a function that looks up the policy base of <*subject, object, access-right*> tuples for *(subject, object)* and returns the corresponding *access-right* for the *subject* and *object*.

Thus if the policy base has the rules:

```
<Brian, hospital.xml/PatientRecords/Patient[@Name=Aaron]/
                                        Medical, OVERWRITE >
<David, hospital.xml/PatientRecords/Patient[@Name=Christy]/
                                        Medical, OVERWRITE >
<Fred, hospital.xml/PatientRecords/Patient[@Name=Emily]/
                                        Medical, OVERWRITE >
<Greg, hospital.xml/PatientRecords/Patient/Medical, READ >
```

the following additional privileges are derived:

```
<David, hospital.xml/PatientRecords/Patient[@Name=Aaron]/
                                        Medical, OVERWRITE>
<David, hospital.xml/PatientRecords/Patient[@Name=Emily]/
                                        Medical, OVERWRITE>
```

We now give some other examples to show the power and flexibility of the approach.

The following gives any *subject* READ access to a patient's medical prescription if he/she can at least READ the diagnosis of the patient.

```
let $hospital =
    document("http://www.example.com/hospital.xml")
let $office =
    document("http://www.example.com/office.xml")

for $u in distinct($hospital//patient/@Name,
                   $office//Employee/@Name) return
    for $med in $hospital//patient/Medical
        where access($u, $med/Diagnosis) >= READ
              return <$u, $med/Prescription, READ>
```

Other XQuery features (e.g., wildcards) can be used to generalize such rules, e.g., using a wildcard for "Diagnosis" in the above would give access if any part of the medical record can be read – arbitrarily general rules are possible, limited only by the power of the query language.

The following example demonstrates how it is possible to use a complex XQuery to derive rights from multiple documents. The policy being implemented is that if a subject can read the contents of any patient's record, then the subject has access to all the hospital's *Employee* records (perhaps a result of a very strong patient's rights law.)

```
let $hospital =
    document("http://www.example.com/hospital.xml")
let $office =
    document("http://www.example.com/office.xml")

for$u in distinct($hospital//patient/@Name,
                  $office//Employee/@Name) return
  if (not (empty (
    for $p in $hospital/PatientRecords/Patient/Medical
        return
        if (access ($u, $p) > VIEW_TAG_SET)
          then <$u, $p, access ($u, $p)>
          else ()
  ))) then
    for $d in $office/Staff/Employee return
        recursiveSpec( $u, $d, READ )
  else ()
```

The function recursiveSpec takes care of the fact that a privilege could be defined only for the node in consideration and not for the sub-tree below it. This would remove the requirement for looking up all the privileges that otherwise propagate negative or positive access to a node and resolving conflicts among them. If the above XQuery simply provided access to the node *office.xml/Staff/Employee*, then the function recursiveSpec would recursively carry similar permissions to selected nodes of its sub-tree:

```
<Staff>
    <Employee Name="Brian">
        <Personal> <SSN> 666-66-6666 </SSN> </Personal>
        <StaffInfo>
            <Position> Doctor </Position>
            <AccountableTo> David </AccountableTo>
        </StaffInfo>
    </Employee>
    <Employee Name="David">
        <Personal> <SSN> 555-55-5555 </SSN> </Personal>
        <StaffInfo>
            <Position> Doctor </Position>
            <AccountableTo />
        </StaffInfo>
    </Employee>
    <Employee Name="Fred">
        <Personal> <SSN> 777-77-7777 </SSN> </Personal>
        <StaffInfo>
            <Position> Doctor </Position>
            <AccountableTo> David </AccountableTo>
        </StaffInfo>
    </Employee>
    <Employee Name="Greg">
        <Personal> <SSN> 888-88-8888 </SSN> </Personal>
        <StaffInfo>
            <Position> Nurse </Position>
            <AccountableTo>
            <Doctor>David</Doctor> <Doctor>Brian</Doctor> <Doctor>Fred</Doctor>
            </AccountableTo>
        </StaffInfo>
    </Employee>
</Staff>
```

Figure 2: www.example.com/office.xml

```
define function recursiveSpec (text $u, element $x,
                               right $y) returns privilege{
    for $a in $x/@* return
        if name($a) = "Name"
          then <$u, $a, $y>
          else ();
    for $e in $x/* return
        if name($e) = "Position" or
          name($e) = "AccountableTo"
          then <$u, $e, $y>
          else ();
    for $e in $x/* return recursiveSpec ($u, $e, $y)
}
```

This particular example could also have been written more simply using the *some* quantifier of XQuery. The complex version above is given to show some of the possibilities of the language. Combined with an appropriate enforcement mechanism for sanitizing documents, this shows how XQuery can be used to provide a general mechanism for specifying rules governing the sanitization.

5. PROTOTYPE AND EXPERIMENTATION

As mentioned above, we can describe rule derivation entirely in XQuery. We now discuss a prototype that generates all privileges that can be derived from a set of derived privileges, a document base, and a prespecified set of privileges. The prototype works by pre- and post-processing the XML documents and rules (see Figure 4). The XQuery modifications suggested before are simplified until all the power can be subsumed without changing XQuery. Privileges are encoded as XML format. (Privileges in XML format can easily be post processed to the format acceptable by access control enforcement systems similar to Author-X.) The privilege set becomes an XML document making it accessible to the XQuery. Thus rules that invoke **access(subject, object)** are altered to include the access-control privileges as an XML document and a subquery that looks up the appropriate access right, specifying the **Subject** and **Object** in XPath. With renaming to give the proper lexical ordering, 1_VIEW_TAG_SET, 2_READ, ... 5_OVERWRITE, access-rights can be operated on using XQuery string operations.

The prototype also does post-processing of the privileges document returned by XQuery to handle issues

```
<PatientRecords>
    <Patient Name="Aaron">
        <Personal>
            <SSN> 999-99-9999 </SSN>
            <DoB> <Month>January</Month><Date>01</Date><Year>1991</Year> </DoB>
        </Personal>
        <Medical>
            <Doctor> Brian </Doctor>
            <Diagnosis> Cancer </Diagnosis>
            <Prescription> Chemo medicine </Prescription>
            <Bill> 500.00 </Bill>
        </Medical>
    </Patient>
    <Patient Name="Christy">
        <Personal>
            <SSN> 444-44-4444 </SSN>
            <DoB> <Month>February</Month> <Date>02</Date> <Year>1972</Year> </DoB>
        </Personal>
        <Medical>
            <Doctor> David </Doctor>
            <Diagnosis> Diabetes </Diagnosis>
            <Prescription> Insulin </Prescription>
            <Bill> 100.00 </Bill>
        </Medical>
    </Patient>
    <Patient Name="Emily">
        <Personal>
            <SSN> 222-22-2222 </SSN>
            <DoB> <Month>March</Month> <Date>03</Date> <Year>1983</Year> </DoB>
        </Personal>
        <Medical>
            <Doctor> Fred </Doctor>
            <Diagnosis> SARS </Diagnosis>
            <Prescription> Unknown </Prescription>
            <Bill> 1000.00 </Bill>
        </Medical>
    </Patient>
</PatientRecords>
```

Figure 3: www.example.com/hospital.xml

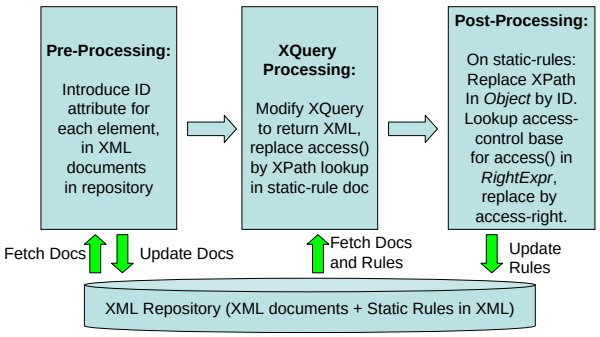

Figure 4: Prototype implementation the pre/postprocessing of rules

introduced by derived rules. Checking 'equality' or 'belongs to' for XPaths, to see if an access condition and privilege match, is non-trivial. For example, *//Patient[@Name="Aaron"]/Medical* and *//Patient//Medical[Doctor = "Brian"]* both point to Aaron's *Medical* records. We dealt with this in our prototype by adding a unique numeric identifier attribute to each element during pre-processing. Post-processing replaces the XPath for the **Object** in each **Privilege** with the unique id(s) of the element(s) the XPath references. This simplifies lookup in an XQuery because a simple string match on the subject, string match on the document name, and integer match on the id is enough to accurately determine the corresponding access-right. (The systems in Section 2 also face this issue; a real system would incorporate their solutions.)

Post-processing is also needed to resolve occurrences of **access(subject, object)** in the returned **Privilege**. This is handled with a scan of the privilege document (including privileges derived from processing rules).

We started with open source implementation for XQuery (GNU's Qexo) in Java. For efficient post-processing lookup of access-rights we used the HashMap data-structure (of the java.util package). To parse the XML documents JDom (the org.jdom package) was used.

5.1 Experimentation

Experiments were conducted to evaluate the semantic correctness of our approach and to test the time and space complexity of the query- and post-processing rule generation as an upper bound on possible implementation methods. For experimenting, a database of two XML files following the schema of Section 4 was generated. *hospital.xml* contains patient records for 50 patients. *office.xml* contains the employee records for 16 doctors and 4 nurses. Each nurse is accountable to 4 doctors; no two are assigned to the same doctor. Each patient is assigned a doctor. Seven rule specifications were executed on the database. They granted patient's and employee's access to their own and each others records. The full rule specifications are listed in the appendix.

Rule 1 gives the doctor assigned to each patient READ access to the patient's record, in particular the Personal/DoB, and OVERWRITE access to the patient's Medical record. It generates 150 rules (3 rules/patient) in 2 seconds. Rule 2 gives nurses the same access to a patient's medical records as the doctors they are accountable to have. It also takes little time, and returns 50 rules (1 rule/patient since only the Medical record is in consideration).

Rule 3 gives the same access-rights cascading down the subtrees of the Medical and DoB elements in the patient's records to the nurses and doctors. It is a recursive function, producing 6000 rules (50 patients, 16 doctors + 4 nurses, 6 elements: Month, Date, Year under DoB and Doctor, Diagnosis, Bill under Medical), and takes 6 seconds. Post-processing reduces this to 450 (50 patients, 6 elements to the serving doctor and 3 elements - only the Medical record - to the concerned nurse).

Rule 4 gives every doctor's supervisor READ access to the Name, READ access to the Medical/Diagnosis and OVERWRITE access to the Medical/Bill for each patient that the doctor serves. It takes time comparable to Rule 1 and produces 141 rules (50 patients; each doctor reports to a supervisor except one, who serves 3 patients).

Rule 5 gives each patient READ access to the doctor(s) and nurse(s) StaffInfo who have OVERWRITE access to that patient's medical records. This query looks up the access-control policy and hence demonstrates transitivity different from Rule 2 and Rule 3 where the lookup is performed during post-processing. This produces 100 rules (1 doctor and 1 nurse have OVERWRITE access to each patient's Medical records) and takes 120 seconds. It takes time to lookup the access-policy base (650 rules from Rule 1 + Rule 2 + Rule 3) for each patient-employee combination to see if the employee has access to the patient's Medical records.

Rule 6 gives each doctor/nurse OVERWRITE access to their own Personal records, APPEND access to their own StaffInfo and READ access to all the other doctors/nurses StaffInfo records unless the former doctor is the latter doctor's/nurse's supervisor in which case he/she gets OVERWRITE access to the doctor's/nurse's StaffInfo. This generates 420 rules (20 rules: each doctor's/nurse's OVERWRITE access to their ownPersonal records; plus each doctor/nurse gets 20 access rules: APPEND access to their own StaffInfo and 19 OVERWRITE/READ access rules to the other doctors/nurses StaffInfo records depending on whether they are supervisor to the doctor/nurse) and takes the same order of time as Rule 1. Rule 7 gives each patient READ access to his/her own Medical record unless the patient has AIDS or cancer and is under 18 years of age. This generates 49 rules since one patient doesn't satisfy the criteria and takes 1 second to execute. A detailed discussion of the XQueries and the resulting privileges can be found in [9].

Table 5.1 summarizes the time to generate all privileges, and the number of privileges that can be derived, for the sample rules. With the exception of Rule 5, all were reasonably fast. Rule 5 required multiple passes over the privilege base; the large number of privileges viewed increased the time required for rule generation.

Table 1: Time taken and privileges derived from sample rules

	Rule 1	*Rule 2*	*Rule 3*	*Rule 4*	*Rule 5*	*Rule 6*	*Rule 7*
# of patients/ employees/ privileges accessed	1000	1000	12000	1000	325000	400	50
time in seconds	2	2	6	2	120	2	1
# of privileges generated	150	50	450	141	100	420	49

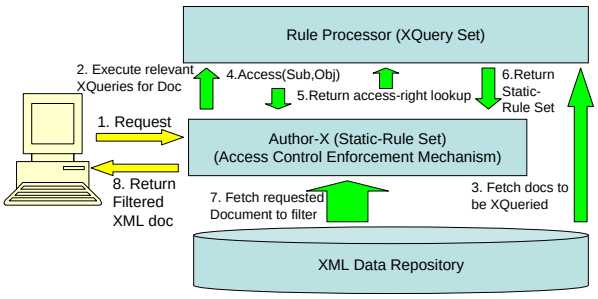

Figure 5: Possible architecture for a real implementation

The space required is dependent on the number of privileges produced. Given M document fragments over which fine grained access control is desired and N users there would be at most $N*M$ privileges. In the example given, the number of privileges generated is small and reflects the selective nature of the policies. Rules of this order can be enforced by a system that applies access control to filter XML document fragments. If the XML repository updates frequently, then instead of executing the queries to generate the privileges set, it would be more efficient to ask the question whether the queries together allow access to the requested document fragment(s), i.e. will some combination of variable bindings produce the privilege(s) to give the *subject* the desired *access-right* on the requested document.

5.2 Issues in a Real-World Implementation

Figure 5 shows a potential architecture for a real world implementation. A rule derivation engine would be built either on top of XQuery as in our prototype, or as a system capable of processing the rule language. This rule deriver would produce the needed rules, and use a system such as Author-X [1] for enforcement.

The techniques for evaluating policies against a document base fall along a spectrum, between no precomputation and complete precomputation. We discuss the ends of the spectrum; an optimal solution would combine both. One approach is to derive privileges for a schema when the data server receives a request for a document conforming to the schema. This returns a set of privileges; existing mechanisms then decide if access to the document is allowed. Only rules pertaining to the documents being requested need be evaluated. It is important that the rules are evaluated in the order of their dependency. A directed acyclic graph structure can capture the dependencies of the rules. Alternatively, a theorem-proving approach could be used to attempt to derive privileges that would allow access to the requested document fragment(s), i.e., will some combination of variable bindings produce the privilege(s) to give the *subject* the desired *access-right* on the requested document? This is similar to the unification procedure in Prolog like languages.

The other approach is to cache, or even precompute, the privileges (as in the prototype). While more efficient on document access, it introduces additional complexity because changes to the data may invalidate old privileges or generate new ones. This is essentially the same problem as materialized derived data maintenance in an XML database, similar solutions will apply.

The rule processor would use an existing mechanism to enforce access control policy on the document base, e.g., Author-X [2]. The rule processor would interface with this mechanism to implement **access(subject, object)**, a lookup in the access control policy base. The privileges returned by the rule processor would be fed into the mechanism to generate the policy base.

5.2.1 Security Issues in Applying Policies

We now discuss a few other challenges – the user interface and ensuring that the security system does not itself create security threats.

A privilege grant should not tacitly give the user all privileges the security system needs to evaluate relevant security policies. Yet in our system, a user's request needs not just the apparent resources, but also the resources that the security system uses to make a decision – and may reveal information (e.g., the existence of a Repair record).[3] Administrators cannot be expected to check all relevant policies and analyze covert channels. We therefore suggest that the security tools attempt to infer the predicate "all users who might satisfy the policy have the right to execute the security queries". If this cannot be proved, the tool should show the administrator a form that can grant the needed privileges (also allowing the choice of no more privileges). Candidates to be presented include:

- Have the security system execute using the privileges of the calling user. This may cause denials in cases where access might otherwise be justified.
- Give just the right to have the security system execute its policy. Administrators will have difficulty judging the power of this covert channel, but the vulnerability is bounded by the next item below.

[3]Solutions referenced in Section 2 could not reference other resources, so the difficulty did not arise.

- Give user the right to see the result of predicates the security system evaluates (e.g., value > threshold), but not the underlying data referenced by those predicates; and
- Give all potential users the right to read all data the security system needs to read. This might be a handy reminder about privileges the users are likely to need for their other work.

Another interesting implementation option is to have policies that produce not a Boolean value, but a modified query with portions omitted.

5.2.2 User Interface

We would not expect a security administrator to directly generate rules in the language we have given. A template/forms/wizard-like abstraction to generate the underlying XQuery code is needed. The underlying similarity of the language to XQuery should enable easy adaptation of XQuery user interfaces to security administration, particularly in combination with interfaces used to specify the underlying rules. Facilities that help in creating new queries by modifying previous ones would seem particularly valuable.

6. CONCLUSION

We have presented a method to enable specifying fine-grained access control in XML based on the relationship of a document to other documents. While no more powerful than existing access control methods, it provides significant *administrative* advantages. Access control on rapidly changing parts of a database can be specified in terms of access on static portions of the database, and content relationships between documents can be used to specify policy at the schema level that generates fine-grained enforcement mechanisms.

In summary, the approach is to:

- Use a powerful, standard query language (XQuery) to define not only the documents and document fragments to which a policy applies, but also to express the policy itself. This enables referencing outside the current document.
- Make the contents of the privilege repository accessible for queries, so one can derive new privileges from old. One may then wish to restrict what a user can see – a natural candidate is to let the user see *only their own privileges.*

More work is needed to demonstrate that this method could be a realistic component of an XML Database. Key open research issues (discussed previously) are

- User interface, and
- Efficient implementation.

Another issue is analysis of access control rules, e.g., what is the effect of a change to a rule. We believe this approach to specifying access control provides significant opportunities for mapping policy into mechanism, enabling a higher level of security and privacy for information stored in XML databases.

Acknowledgments

We would like to thank Elisa Bertino for her suggestions that have helped aim the work at making fine-grained access control easier to administer.

7. REFERENCES

[1] E. Bertino, M. Braun, S. Castano, E. Ferrari, and M. Mesiti. AuthorX: A java-based system for XML data protection. In *Proceedings of the 14th Annual IFIP WG 11.3 Working Conference on Database Security*, Aug. 2000.

[2] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Specifying and enforcing access control policies for XML document sources. *World Wide Web Journal*, 3(3), 2000.

[3] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Design and Implementation of an Access Control Processor for XML Documents. *Computer Networks*, 33(1-6):59–75, 2000.

[4] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Proceedings of the International Conference on Extending Database Technology (EDBT)*, Mar. 27-31 2000.

[5] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Controlling access to XML documents. *IEEE Internet Computing*, 5(6):18–28, Nov. 2001.

[6] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security (TISSEC)*, 5(2):169–202, May 2002.

[7] S. D. C. di Vimercati. An authorization model for temporal XML documents. In *Proceedings of the 2002 ACM Symposium on Applied Computing (SAC)*, pages 1088–1093, Mar. 2002.

[8] A. Gabillon and E. Bruno. Regulating access to XML documents. In *Proceedings of the 15th Annual IFIP WG 11.3 Working Conference on Database Security*, July 15-18 2001.

[9] S. K. Goel. Transitive access control specification for XML. Master's thesis, Purdue University, West Lafayette, IN, Aug. 2003. http://www.cs.purdue.edu/~skgoel/paper.pdf.

[10] H. Jagadish, L. V. Lakshmanan, D. Srivastava, and T. Yu. Compressed accessibility map: Efficient access control for XML. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, Sept. 2002.

[11] M. Kudo and S. Hada. XML document security based on provisional authorization. In *7th ACM Conference on Computer and Communication Security (CCS)*, Nov. 2000.

[12] eXtensible Access Control Markup Language (XACML) version 1.0, Feb. 18 2003. http://www.oasis-open.org/committees/xacml/repository/oasis-xacml-1.0.pdf.

APPENDIX

A. RULE 1

Give the doctor assigned to each patient READ access to the patient's record, in particular the *Personal/DoB*, and OVERWRITE access to the patient's *Medical* record.

```
<access-rights>
let $office := document ( "office.xml" )
let $hospital := document ( "hospital.xml" )
for $doc1 in $hospital//Patient return
    for $doc2 in $office//Employee
        where $doc2//Position = "Doctor" and $doc1//Doctor = $doc2/@Name return
<Rules>
    <Rule>
        <Subject>{$doc2/@Name}</Subject>
        <Object>
                <File>hospital.xml</File>
                <Path>//Patient[{ $doc1/@Name }]</Path>
        </Object>
        <Right>READ</Right>
    </Rule>
    <Rule>
        <Subject>{$doc2/@Name}</Subject>
        <Object>
                <File>hospital.xml</File>
                <Path>//Patient[{ $doc1/@Name } ]//DoB</Path>
        </Object>
        <Right>READ</Right>
    </Rule>
    <Rule>
        <Subject>{$doc2/@Name}</Subject>
        <Object>
                <File>hospital.xml</File>
                <Path>//Patient[{ $doc1/@Name }]/Medical</Path>
        </Object>
        <Right>OVERWRITE</Right>
    </Rule>
</Rules>
</access-rights>
```

B. RULE 2

Give nurses the same access to a patient's medical records as the doctors they are accountable to have.

```
<access-rights>
let $office := document ( "office.xml" )
let $hospital := document ( "hospital.xml" )
for $doc1 in $hospital//Patient return
    for $doc2 in $office//Employee
        where $doc2//Position="Nurse" and $doc1//Doctor = $doc2//Doctor return
<Rule>
    <Subject>{$doc2/@Name}</Subject>
    <Object>
        <File>hospital.xml</File>
        <Path>//Patient[{ $doc1/@Name } ]/Medical</Path>
    </Object>
    <Right>access
        <Subject>{$doc1//Doctor/text()}</Subject>
        <Object>
                <File>hospital.xml</File>
                <Path>//Patient[{ $doc1/@Name }]/Medical</Path>
        </Object>
```

```
    </Right>
</Rule>
</access-rights>
```

C. RULE 3

Give the same access-rights cascading down the subtrees of the Medical and DoB elements in the patient's records to the nurses and doctors.

```
define function recursiveSpec($f, $o, $x, $u) {
        for $e in $x/* return
        <Rules>
                <Rule>
                        <Subject> {$u}</Subject>
                        <Object>
                                <File>{$f}</File>
                                <Id>{$e/@NewID}</Id>
                        </Object>
                        <Right>access
                                <Subject>{$u}</Subject>
                                <Object>
                                        <File>{$f}</File>
                                        <Id> {$o/@NewID}</Id>
                                </Object>
                        </Right>
                </Rule>
        { for $e in $x/* return
                        recursiveSpec($f, $o, $e, $u) }
        </Rules>
}
<access-rights>
{
for $u in document("office.xml")//Employee return
        for $hospital in document("hospital.xml")//Patient//Medical return
                recursiveSpec("hospital.xml", $hospital, $hospital, $u/@Name),

for $u in document("office.xml")//Employee return
        for $hospital in document("hospital.xml")//Patient//DoB return
                recursiveSpec("hospital.xml", $hospital, $hospital, $u/@Name)
}
</access-rights>
```

D. RULE 4

Give every doctor's supervisor READ access to the Name, READ access to the Medical/Diagnosis and OVERWRITE access to the Medical/Bill for each patient that the doctor serves.

```
<access-rights>
let $office := document ( "office.xml" )
let $hospital := document ( "hospital.xml" )
for $doc1 in $hospital//Patient return
    for $doc2 in $office//Employee
        where $doc2//Position = "Doctor" and $doc1//Doctor = $doc2/@Name return
<Rules>
    <Rule>
        <Subject>{$doc2//AccountableTo/text()}</Subject>
        <Object>
              <File>hospital.xml</File>
              <Path>//Patient[{ $doc1/@Name } ]</Path>
        </Object>
```

```
        <Right>READ</Right>
    </Rule>
    <Rule>
        <Subject>{$doc2//AccountableTo/text()}</Subject>
        <Object>
                <File>hospital.xml</File>
                <Path>//Patient[ { $doc1/@Name } ]/Medical/Diagnosis</Path>
        </Object>
        <Right>READ</Right>
    </Rule>
    <Rule>
        <Subject>{$doc2//AccountableTo/text()}</Subject>
        <Object>
                <File>hospital.xml</File>
                <Path>//Patient[ { $doc1/@Name } ]/Medical/Bill</Path>
        </Object>
        <Right>OVERWRITE</Right>
    </Rule>
</Rules>
</access-rights>
```

E. RULE 5

Give each patient READ access to the doctor(s) and nurse(s) StaffInfo who have OVERWRITE access to that patient's medical records.

```
<access-rights>
let $office := document ( "office.xml" )
let $hospital := document ( "hospital.xml" )
let $accesspolicy := document ( "ruleBase1-2-3.xml" )
for $a in $hospital//Patient return
    for $b in $office//Employee
        where $accesspolicy//Rule[Subject = $b/@Name
        and Object/Id = $a//Medical/@NewID
        and Object/File = "hospital.xml"]/Right = "OVERWRITE" return
<Rule>
    <Subject>{$a/@Name}</Subject>
    <Object>
        <File>office.xml</File>
        <Path>//Employee[{$b/@Name}]//StaffInfo</Path>
    </Object>
    <Right> READ </Right>
</Rule>
</access-rights>
```

F. RULE 6

Give each doctor/nurse OVERWRITE access to their own Personal records, APPEND access to their own StaffInfo and READ access to all the other doctors/nurses StaffInfo records unless the former doctor is the latter doctor's/nurse's supervisor in which case he/she gets OVERWRITE access to the doctor's/nurse's StaffInfo.

```
<access-rights>
let $office := document ( "office.xml" )
for $b in $office//Employee return
    for $c in $office//Employee return
    if($b/@Name = $c/@Name) then (
        <Rules>
                <Rule>
                        <Subject>{$c/@Name}</Subject>
                        <Object>
                                <File>office.xml</File>
```

```
                        <Path>//Employee[{ $b/@Name } ]/Personal</Path>
                    </Object>
                    <Right>OVERWRITE</Right>
                </Rule>
                <Rule>
                    <Subject>{$c/@Name}</Subject>
                    <Object>
                        <File>office.xml</File>
                        <Path>//Employee[{ $b/@Name } ]/StaffInfo</Path>
                    </Object>
                    <Right>APPEND</Right>
                </Rule>
        </Rules>
    )
    else (
        if(($b//Position = "Doctor" and $c/@Name = $b//AccountableTo)
        or ($b//Position = "Nurse" and $c/@Name = $b//AccountableTo/Doctor)) then (
                <Rule>
                    <Subject>{$c/@Name}</Subject>
                    <Object>
                        <File>office.xml</File>
                        <Path>//Employee[{ $b/@Name } ]/StaffInfo</Path>
                    </Object>
                    <Right>OVERWRITE</Right>
                </Rule>
        )
        else (
                <Rule>
                    <Subject>{$c/@Name}</Subject>
                    <Object>
                        <File>office.xml</File>
                        <Path>//Employee[{ $b/@Name } ]/StaffInfo</Path>
                    </Object>
                    <Right>READ</Right>
                </Rule>
        )
    )
</access-rights>
```

G. RULE 7

Give each patient READ access to his/her own Medical record unless the patient has AIDS or cancer and is under 18 years of age.

```
<access-rights>
let $hospital := document("hospital.xml")
for $p in $hospital//Patient
    where (1985 > $p//Year) or
    not($p//Diagnosis = "AIDS" or $p//Diagnosis = "Cancer") return
<Rule>
        <Subject>{$p/@Name}</Subject>
        <Object>
                <File>hospital.xml</File>
                <Path>//Patient[{$p/@Name}]/Medical</Path>
        </Object>
        <Right>READ</Right>
</Rule>
</access-rights>
```