XML Access Control Using Static Analysis

MAKOTO MURATA, AKIHIKO TOZAWA, MICHIHARU KUDO,
and SATOSHI HADA
IBM Tokyo Research Lab

Access control policies for XML typically use regular path expressions such as XPath for specifying the objects for access-control policies. However such access-control policies are burdens to the query engines for XML documents. To relieve this burden, we introduce static analysis for XML access-control. Given an access-control policy, query expression, and an optional schema, static analysis determines if this query expression is guaranteed not to access elements or attributes that are hidden by the access-control policy but permitted by the schema. Static analysis can be performed without evaluating any query expression against actual XML documents. Run-time checking is required only when static analysis is unable to determine whether to grant or deny access requests. A side effect of static analysis is query optimization: access-denied expressions in queries can be evaluated to empty lists at compile time. We further extend static analysis for handling value-based access-control policies and introduce view schemas.

Categories and Subject Descriptors: H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; D.4.6 [**Operating Systems**]: Security and Protection—*Access controls*

General Terms: Algorithms, Performance, Experimentation, Security, Theory

Additional Key Words and Phrases: Access control, automaton, query optimization, schema, static analysis, value-based access control, view schema, XML, XQuery, XPath

1. INTRODUCTION

XML [Bray et al. 2004] has become an active area in the IT industry. XPath [Clark and DeRose 1999] and XQuery [Boag et al. 2003] from the W3C have come to be widely recognized as query languages for XML, and their implementations are actively in progress. In this paper, we are concerned with fine-grained (element- and attribute-level) access control for XML database systems.

An earlier version [Murata et al. 2003] of this paper was presented at the 10th ACM Conference on Computer and Communications Security (CCS), but it did not have value-based access control or the view schemas shown in Sections 5 and 6, respectively. In addition, the performance of static analysis has been significantly improved.
Authors' address: M. Murata, A. Tozawa, M. Kudo, and S. Hada, IBM Tokyo Research Lab, 1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan; email: {EB91801,atozawa,kudo, satoshih}@jp.ibm.com.

We believe that access control plays an important role in XML database systems, as it does in relational database systems. XML access control is also applicable to XML publish/subscribe systems and XML content management systems among others. Some early experiences [Kudo and Hada 2000; Damiani et al. 2000; Bertino et al. 2001] with access control for XML documents have already been reported.

Access control for XML documents should ideally provide expressiveness as well as efficiency, that is, (1) it should be easy to write fine-grained access-control policies, and (2) it should be possible to efficiently determine whether access to an element or an attribute is granted or denied according to those fine-grained access-control policies. It is difficult to fulfill both of these requirements, since XML documents have richer structures than relational databases. In particular, access-control policies, query expressions, and schemas for XML documents are required to handle an infinite number of paths, since there is no upper bound on the height of XML document trees.

Existing languages [e.g., Kudo and Hada 2000; Damiani et al. 2000] for XML access control achieve expressiveness by using XPath [Clark and DeRose 1999] as a simple and powerful mechanism for handling an infinite number of paths. For example, to deny accesses to `name` elements that are immediately or non-immediately subordinate to `article` elements, it suffices to specify a simple XPath expression `//article//name` as part of an access-control policy.

However, XPath-based access-control policies are additional burdens for XML query engines. Whenever an element or attribute in some XML document is accessed at run time, a query engine is required to check whether or not this access is granted by the access-control policies. Since such accesses are frequently repeated during query evaluation, naive implementations for checking access-control policies can lead to unacceptable performance. Many researchers have attempted to improve the performance of run-time checking of access-control policies (see Section 3.3).

In this paper, we introduce static analysis as a new approach for XML access-control. Static analysis examines access-control policies and query expressions as well as schemas, if present. Unlike the run-time checking described above, static analysis does not examine actual XML documents. Thus, static analysis does not have to be repeated even when a query is evaluated many times or when a query accesses many elements or attributes in the actual XML documents. Static analysis can often grant or deny access requests, thereby making run-time checking unnecessary. Run-time checking is required only when static analysis is unable to grant or deny access requests without examining the actual XML documents. Static analysis has two more advantages. First, it helps programmers to make sure that their queries do not cause access-control errors. Second, static analysis facilitates query optimization, since access-denied XPath expressions in queries can be rewritten as empty lists at compile time.

The key idea for our static analysis is to use automata for representing and comparing queries, access-control policies, and schemas. Our static analysis has two phases. In the first phase, we create regular expressions or automata from queries, access-control policies, and (optionally) schemas: (1) regular expressions created from queries, called *query regular expression*, represent paths

to elements or attributes as accessed by these queries; (2) those created from access-control policies, called *access-control automata*, represent paths to elements or attributes as exposed by these access-control policies; and (3) those created from schemas, called *schema automata*, represent paths to elements or attributes as permitted by these schemas. In the second phase, we compare these regular expressions and automata while applying the following rules: (1) accesses by queries are *always granted* if the intersection of the query regular expressions and the schema automata is subsumed by the access-control automata; (2) they are *always denied* if the intersection of the query regular expressions, schema automata, and access-control automata is empty; and (3) they are *statically indeterminate*, otherwise.

After introducing static analysis, we extend our framework by introducing view schemas. A view schema is derived from an original schema by enforcing an access-control policy statically. Unlike the original schema, the view schema allows only those elements or attributes that are exposed by the policy. Since the view schema hides superfluous information about access-denied elements or attributes, it is more programmer friendly than the original schema.

1.1 Related Works

Fine-grained access-control for XML documents has been studied by many researchers [Bertino et al. 1999, 2001; Kudo and Hada 2000; Damiani et al. 2000; Gabillon and Bruno 2001]. Their access-control policies are similar to ours. They all provide run-time checking of access-control policies, but do not consider static analysis. Their algorithms for run-time checking all assume that the XML documents are in the main memory and can be examined repeatedly.

Several researchers (e.g., [Gottlob et al. 2002; Altinel and Franklin 2000; Chan et al. 2002; Li and Moon 2001; Baeza-Yates and Navarro 2002]) have developed sophisticated techniques for evaluating XQuery or XPath expressions. Such techniques include XPath evaluation algorithms [e.g., Gottlob et al. 2002], stream processing [e.g., Altinel and Franklin 2000; Chan et al. 2002; Green et al. 2002; Barton et al. 2003], index files [e.g., Li and Moon 2001; Baeza-Yates and Navarro 2002], and so forth.

Access control for an RDBMS is driven by views, which hide some information (typically attributes in relations) in the RDBMS. Queries written by users do not access-actual databases, but rather access these views. View-driven access control is typically efficient, since view and user queries are optimized together and then executed. In other words, access control is provided partly by optimization at compile-time and partly by checking at run time.

Object-oriented database systems (OODBMS) provide richer structures than RDBMSs or XML. In fact, an OODBMS can provide network structures and class hierarchies. Access-control frameworks for OODBMSs have appeared in the literature [Bertino 1992; Rabitti et al. 1991]. Such frameworks typically rely on run-time checking and do not use static analysis.

Our static analysis for XML access-control is made possible by the tree-structured nature of XML. First, the schemas for XML are regular tree

grammars from which we can generate automata that represent the permissible paths. Second, both access-control policies and queries for XML can use regular path expressions (XPath) for locating elements or attributes. We can, therefore, use those automata and regular expressions for uniformly handling schemas, queries, and access-control policies.

The use of automata for XML is not new. Many researchers have used automata (string automata or tree automata) for handling queries, schemas, patterns, or integrity constraints. Furthermore, recent works apply Boolean operations (typically the intersection operation) to such automata. These works include type checking [e.g., Hosoya and Pierce 2003; Draper et al. 2004], query optimization using schemas [e.g., Fernández and Suciu 1998], query optimization using views [e.g., Papakonstantinou and Vassalos 1999; Deutsch and Tannen 2001; Miklau and Suciu 2004; Neven and Schwentick 2003; Wood 2003], or consistency between integrity constraints and schemas [e.g., Fan and Libkin 2002]. Our static analysis uses similar techniques. However, to the best of our knowledge, our static analysis is the first application of automata for XML access control.

XPath containment [Deutsch and Tannen 2001; Miklau and Suciu 2004; Neven and Schwentick 2003; Wood 2003] is similar to our static analysis, since we compare XPath expressions for queries and those for access-control policies. However, denial rules (shown in Section 3) in access-control policies require that our static analysis apply the negation operation to automata and use both over- and underestimation of access-control automata.

After the publication of an earlier version [Murata et al. 2003] of this paper, Fan et al. [2004] and Luo et al. [2004] were published. Fan et al. [2004] introduced view-based access-control for XML. Although their work is restricted to DTDs and XPath, it uses view schemas for eliminating access-denied information from schemas. Our view schemas are inspired by their work, but the construction algorithm is entirely different. Luo et al. [2004] provides static analysis similar to ours, but their implementation is significantly faster than our previous result. We have borrowed one of their ideas for improving the performance of our static analysis.

1.2 Outline

The rest of this paper is organized as follows. After presenting the fundamentals of XML, schemas, XPath, and XQuery in Section 2, we introduce access-control policies for XML documents in Section 3. We introduce static analysis in Section 4 and then extend it to handle value-based access control in Section 5. We then introduce view schemas in Section 6. In addition, we show a real-world example and demonstrate the scalability of static analysis in Section 7. In Section 8, we conclude.

2. PRELIMINARIES

In this section, we introduce the basics of XML, schema languages, XPath, and XQuery.

```
<record patientId="0003">
  <diagnosis>
    <pathology type="Gastric Cancer">
      Well differentiated adeno carcinoma
    </pathology>
    <comment>This seems correct</comment>
  </diagnosis>
  <chemotherapy>
    <prescription>5-FU 500 mg</prescription>
    <comment>Is this sufficient?</comment>
  </chemotherapy>
  <comment>How was the operation?</comment>
</record>
```

Fig. 1. An XML document example.

2.1 XML

An XML document consists of elements, attributes, and text nodes. These elements collectively form a tree. The content of each element is a sequence of elements or text nodes. An element has a set of attributes, each of which has a name and a value. We hereafter use Σ^E and Σ^A as a set of tag names and that of attribute names, respectively. To distinguish between the symbols in these sets, we prepend "@" to symbols in Σ^A.

An XML document representing a medical record is shown in Figure 1. This XML document describes diagnosis and chemotherapy information for a certain patient. Several comments have been inserted to this document. For the rest of this paper, we use this document as a motivating example.

2.2 Schema

A *schema* is a description of permissible XML documents. A *schema language* is a computer language for writing schemas. DTD, W3C XML Schema [Thompson et al. 2001], and RELAX NG [Clark and Murata 2001] from OASIS (and now ISO/IEC) are notable examples of schema languages.

We do not use any particular schema language in this paper, but rather use tree regular grammars [Comon et al. 1997] as a formal model of schemas. Murata et al. [2005] have shown that tree regular grammars can model DTD, W3C XML Schema, and RELAX NG.[1]

A *schema* is a five tuple $G = (N, \Sigma^E, \Sigma^A, S, P)$, where:

—N is a finite set of *nonterminals*,
—Σ^E is a finite set of *element names*,
—Σ^A is a finite set of *attribute names*,
—S (*start set*) is a subset of $\Sigma^E \times N$,
—P is a set of production rules $X \rightarrow r\ \mathcal{A}$, where $X \in N$, r is a regular expression over $\Sigma^E \times N$, and $\mathcal{A}$ is a subset of Σ^A.

[1]However, each schema language (most notably W3C XML Schema) has features that cannot be precisely captured by tree regular grammars. For more about this, see Murata et al. [2005].

Production rules collectively specify permissible element structures. We separate nonterminals and element names, since we want to allow elements of the same name to have different subordinates depending on where these elements occur. Although examples in this paper can be captured without separating nonterminals and element names, W3C XML Schema and RELAX NG require this separation. Unlike the definition in Murata et al. [2005], we allow production rules to have a set of permissible attribute names.[2]

For the sake of simplicity, we do not allow schemas to specify constraints on text nodes or attribute values. In the case of DTDs, this restriction amounts to the confusion of `#PCDATA` and `EMPTY`.

A schema is said to be *recursive* if it does not impose any upper bound on the height of XML documents. The above schema is recursive, since `record` elements are allowed to nest freely. Since most schemas (e.g., XHTML and DocBook) for narrative documents are recursive, our static analysis must handle recursive schemas and an infinite number of permissible paths.

A limitation of our approach is the omission of simple types such as integers. This omission is one of the reasons that our static analysis sometimes provide indeterminate answers.

2.2.1 *Example.* A schema for our motivating example is $G_1 = (N_1, \Sigma_1^E, \Sigma_1^A, S_1, P_1)$, where

$$
\begin{aligned}
N_1 &= \{\text{Record, Diag, Chem, Com, Patho, Presc}\},\\
\Sigma_1^E &= \{\texttt{record}, \texttt{diagnosis}, \texttt{chemotherapy},\\
&\qquad \texttt{comment}, \texttt{pathology}, \texttt{prescription}\},\\
\Sigma_1^A &= \{\texttt{@patientId}, \texttt{@type}\},\\
S_1 &= \{\texttt{record}[\text{Record}]\},\\
P_1 &= \{\text{Record} \rightarrow (\texttt{diagnosis}[\text{Diag}]^*,\\
&\qquad\qquad \texttt{chemotherapy}[\text{Chem}]^*,\\
&\qquad\qquad \texttt{comment}[\text{Com}]^*, \texttt{record}[\text{Record}]^*)\ \{\texttt{@patientId}\},\\
&\qquad \text{Diag} \rightarrow (\texttt{pathology}[\text{Patho}], \texttt{comment}[\text{Com}]^*)\ \emptyset,\\
&\qquad \text{Chem} \rightarrow (\texttt{prescription}[\text{Presc}]^*, \texttt{comment}[\text{Com}]^*)\ \emptyset,\\
&\qquad \text{Com} \rightarrow \epsilon\ \emptyset,\quad \text{Patho} \rightarrow \epsilon\ \{\texttt{@type}\},\quad \text{Presc} \rightarrow \epsilon\ \emptyset\}.
\end{aligned}
$$

An equivalent DTD is shown below.

```
<!ELEMENT record       (diagnosis*,chemotherapy*,
                        comment*,record*)>
<!ATTLIST record       patientID CDATA #REQUIRED>
<!ELEMENT diagnosis    (pathology,comment*)>
<!ELEMENT chemotherapy (prescription*,comment*)>
<!ELEMENT comment      (#PCDATA)>
<!ELEMENT pathology    (#PCDATA)>
```

[2]RELAX NG provides a more sophisticated mechanism for handling attributes [Hosoya and Murata 2002].

```
<!ATTLIST pathology     type CDATA #REQUIRED>
<!ELEMENT prescription (#PCDATA)>
```

2.3 XPath

Given an XML document, we often want to locate certain elements by specifying conditions on the elements and on their ancestor elements. For example, we may want to locate all anchors (e.g., <a ...>... in XHTML) occurring in paragraphs (e.g., <p ...>...</p> in XHTML). In this example, "anchor" is a condition on elements and "occurring in paragraphs" is a condition on ancestor elements. Such conditions can easily be captured by regular path expressions, which are regular expressions describing permissible paths from the root element to elements or attributes.

XPath [Clark and DeRose 1999] provides a restricted variation of regular path expressions. XPath is widely recognized in the industry and is used by XSLT [Clark 1999] and XQuery. We focus on XPath in this paper, although our framework is applicable to any regular path expression.

XPath uses *axes* for representing the structural relationships between nodes. For example, the above example can be captured by the XPath expression `//p//a`, where `//` is an axis called "descendant-or-self." Although XPath provides many axes, we consider only three of them, namely, "descendant-or-self" (//), "child" (/), and "attribute" (@) in this paper. Extensions for handling other axes are discussed in Section 8. Namespaces and wild cards are outside the scope of this paper, although our framework can easily handle them.

XPath allows conditions on elements to have additional conditions. For example, we might want to locate `foo` elements such that their `@bar` attributes have `"abc"` as the values. Such additional conditions are called *predicates*. This example can be captured by the XPath expression `//foo[@bar = "abc"]`, where `[@bar = "abc"]` is a predicate.

2.4 XQuery

Several query languages for XML have recently emerged. Although they have different query algebras, most of them use XPath for locating elements or attributes. Our framework can be applied to any query language as long as it uses regular path expressions for locating elements or attributes. However, we focus on XQuery [Boag et al. 2003] in the rest of this paper.

FLWOR (For, Let, Where, Order by, Return) expressions are of central importance to XQuery. A FLWOR expression consists of a FOR, LET, WHERE, ORDER BY, and RETURN clause.

The FOR or LET clause associates one or more variables with XPath expressions. By evaluating these XPath expressions, the FOR and LET clauses in a FLWOR expression create tuples. The WHERE clause imposes additional conditions on tuples. Those tuples not satisfying the WHERE clause are discarded. The ORDER BY clause reorders the remaining tuples. Finally, for each of the tuples in sequence, the RETURN clause is evaluated and a value or sequence of values is returned.

The following query lists the pathology-comment pairs for gastric cancer.

```
<TreatmentAnalysis>
{
  for $r in document("medical_record")/record
  where $r/diagnosis/pathology/@type
          = "Gastric Cancer"
  return
    $r/diagnosis/pathology, $r//comment
}
</TreatmentAnalysis>
```

3. ACCESS CONTROL FOR XML DOCUMENTS

In this paper, access control for XML documents means element- and attribute-level access control for a certain XML instance. Each element and attribute is handled as a unit resource to which access is controlled by the corresponding access-control policies. In the following sections, we use the term *node-level* access-control when there is no need to separate the *element-level* access control from the *attribute-level* access control.

3.1 Syntax of Access-Control Policy

The access-control policy consists of a set of access-control rules. Each rule consists of an *object* (a target node), a *subject* (a human user or a user process), an *action*, and a *permission* (grant or denial) meaning that the *subject* is (or is not) allowed to perform the *action* on the *object*. The subject value is specified using a user ID, a role, or a group name, but is not limited to these. For the object value, we use an XPath expression. The action value can be either *read, update, create,* or *delete,* but we deal only with the *read* action in this paper because the current XQuery does not support other actions. The following is the syntax of our access-control policy:[3]

```
(Subject, +/-Action, Object)
```

The subject has a prefix indicating the type of the subject, such as role and group. " +" means grant access and "−" means deny access. In this paper, we sometimes omit specifying the subject if the subject is identical with the other rules.

Suppose there are three access-control rules for the document described in Section 2.1:

Role: *Doctor*
```
+R, /record
```

Role: *Intern*
```
+R, /record
-R, //comment
```

[3]The syntax of the policy can be represented in a standardized way using XACML [Godik and Moses (Eds) 2003], but we use our syntax for simplicity.

Each rule is categorized by the role of the requesting subject. The first rule says that "*Doctor* can read `record` elements." The second rule says that "*Intern* can read `record` elements." The third rule says that "*Intern* cannot read any `comment` elements," because `comment` nodes may include confidential information and should be hidden from access by *Intern*. (Please refer to Section 3.2 for more precise semantics.)

3.1.1 *Using X Path for XML access control.* Many reports [Kudo and Hada 2000; Damiani et al. 2000; Bertino et al. 2001; Gabillon and Bruno 2001] on the node-level access control for XML documents use XPath to locate the target nodes in the XML documents.

There are a couple of reasons why we use XPath for our access-control policy. First, XPath provides a sufficient number of ways to refer to the smallest unit of an XML document structure, such as an element, an attribute, a text node, or a comment node. Therefore, it allows a policy writer to write a policy in a flexible manner (e.g., grant access to a certain element but deny access to the enclosing attributes). In this paper, for simplicity, we limit target nodes of the policy to only the elements and attributes. We assume that other nodes, such as text and comment nodes, are governed by the policy associated with the parent element.

Second, it is often the case that the access to a certain node is determined by a value in the target XML document. For a medical record, a patient may be allowed to read his or her own record, but not another patient's record. Therefore, the access-control policy should provide a way to represent a necessary constraint on the record. By using an XPath predicate expression, such a policy could be specified as `(Role:patient, +R, /record[@patientId = $userid])`.[4] This policy says that the access to a `record` element and its subordinates is allowed if the value of the `patientId` attribute is equal to the user ID of the requesting subject. We use the term *value-based access control* to refer to an access-control policy (or rule) that includes such an XPath predicate that references a value. Given value-based policies, our static analysis sometimes, but not always, provide determinate answers (i.e., "always granted" or "always denied"), as shown in Section 5.

Powerful predicates of XPath allow more sophisticated access-control policies. For example, `(Role:somebody, +r, //x[foo/@a = bar/@b])` allows the access to an `x` element when the value of the `a` attribute of some `foo` child element is equal to the value of the `b` attribute of some `bar` child element. Given such sophisticated policies, our static analysis typically provides indeterminate answers, thus requiring run-time checking.

3.2 Semantics of Access-Control Policy

Access-control policies, in general, should satisfy the following requirements: *succinctness*, *least privilege*, and *soundness*. Succinctness means that the policy semantics should provide a way to specify a smaller number of rules rather than to specify rules on every single node in the document. Least privilege means

[4]We use a variable $user id to refer to the identity of the requesting user in the access-control policies.

```
<record patientID="0003">
  <diagnosis>
    <pathology type="Gastric Cancer">
      Well differentiated adeno carcinoma
    </pathology>
  </diagnosis>
  <chemotherapy>
    <prescription>5-FU 500 mg</prescription>
  </chemotherapy>
</record>
```

Fig. 2. The XML document that *Intern* can see.

that the policy should grant the minimum privilege to the requesting subject. Soundness means that the policy evaluation must always generate either a grant or a denial decision in response to any access request.

To satisfy the above requirements, we adopt the "denial-takes-precedence" principle[5] and define the semantics of our access-control policies as follows:

1. An access-control rule with `+R` or `-R` (capital letter) propagates downward through the XML document structure. An access-control rule with `+r` or `-r` (small letter) does not propagate and just describes the rule on the specified node.
2. A rule with denial permission for a node overrules any rules with grant permission for the same node.
3. If no rule is associated with a certain node, the default denial permission "-" is applied to that node.

Now we informally describe an algorithm to generate an access decision according to the above definitions. First, the algorithm gathers every grant rule with `+r` and marks "+" on the target nodes referred to by the XPath expression. If the node type is an element, the algorithm marks "+" on immediate children nodes (e.g., a text and comment nodes), except for the attributes and the elements. It also marks a "+" on all the descendant nodes if the action is `R`. Next, the algorithm gathers the remaining rules (denial rules) and marks "−" on the target nodes in the same way. The "−" mark overwrites the "+" mark if any. Finally, the algorithm marks "−" on every node that is not yet marked. This operation is performed for each subject and action independently.

For example, the access-control policy in Section 3.1 is interpreted as follows: The first rule marks the entire tree with "+" and, therefore, *Doctor* is allowed to read every node (including attributes and text nodes) equal to or below any `record` element. The second and third rules are policies for *Intern*. The second rule marks the entire tree with "+" as the first rule does and the third rule marks `comment` elements and subordinate text nodes with "−," which overwrites + marks. Thus, three `comment` elements and text nodes are determined as "access denied." The XML document that *Intern* can see is shown in Figure 2.

[5]We could adopt the "grant-takes-precedence" principle. That is, we could make *grant* rules overrule denial ones and use the default *grant* permission "+." In Section 4, we show how our static analysis is modified accordingly.

A rule that uses +R or −R can be converted to the rule with +r or −r. For example, (*Sbj*, +R,/a) is semantically equivalent to a set of three rules: (*Sbj*, +r,/a), (*Sbj*, +r,/a//*), and (*Sbj* ,+r,/a//@*). Thus, +R and −R are technically syntactic sugar, but enable a more succinct representation of the policy specification.

3.3 Run-Time Checking of Access-Control Policies

For the integration of access control and query processing, we assume that if there are access-denied nodes in a target XML document, the query processor behaves as if they do not exist in the document.[6]

We will now explain how the semantics described above are enforced by the access-control system at run-time. The sample scenario is the following: Whenever an access to a node (and its descendant nodes) is requested, the node-level access controller makes an access decision on each node. The controller first retrieves the access-control rules applicable to the requested node(s). The controller then computes the access decision(s) according to the rules and returns *grant* or *denial* for each node. A naive implementation of this scenario can lead to poor performance, since the XPath expressions in the rules are evaluated whenever a node is accessed. Many researchers (including the third author of the present paper) have attempted to improve the performance of run-time checking. Some create index files or other data structures from XML documents and access-control policies in advance, and use them for efficient run-time checking. Others [Naishin Qi 2005] convert access-control policies to other devices, which can be efficiently evaluated when an XML document is given.

Our static analysis is *not* intended to entirely eliminate run-time checking, but rather intended to complement it. When static analysis cannot provide determinate answers, we rely on run-time checking.

4. STATIC ANALYSIS

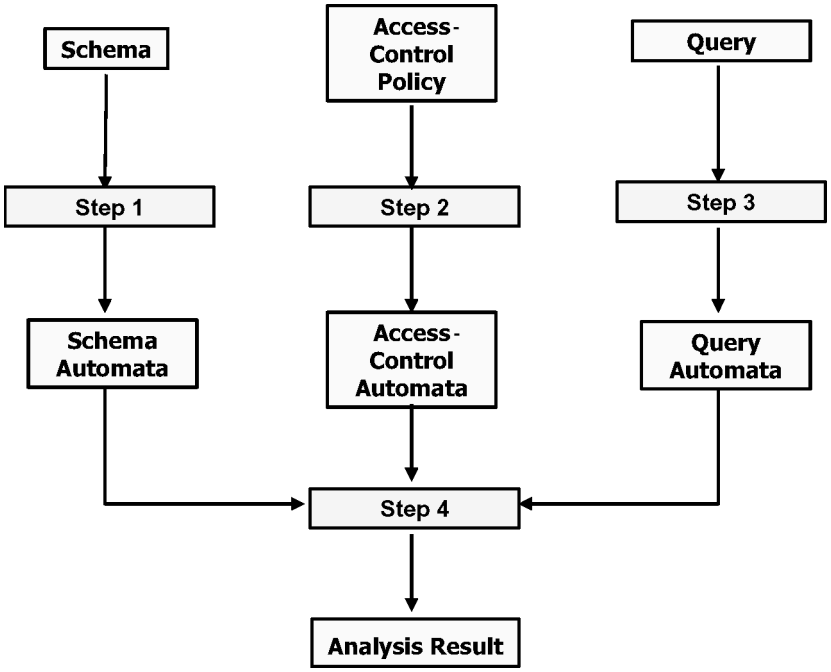
In this section, we introduce our framework for static analysis. The key idea is to use automata for comparing schemas, access-control policies, and regular query expressions.

Figure 4 depicts an overview of our static analysis. Static analysis has four steps:

Step 1: Creating schema automata from schemas
Step 2: Creating access-control automata from the access-control policies
Step 3: Creating query regular expressions from the XQuery queries
Step 4: Comparing the schema automata, access-control automata, and query regular expressions

Schema automata represent paths to elements or attributes as permitted by the schemas. Access-control automata represent paths to elements or attributes

[6]Another semantic model is to raise an access violation error whenever the query processor encounters an access-denied node. With the exception of query optimization in Section 4.6, our static analysis and view schemas are applicable to this semantic model as well.



Step 1: Construct an schema automaton $M^G = (\Sigma^E \cup \Sigma^A, N \cup \{q^{\text{ini}}, q^{\text{fin}}\}, \{q^{\text{ini}}\}, N \cup \{q^{\text{fin}}\}, \delta)$ from a schema $G = (N, \Sigma^E, \Sigma^A, S, P)$, where δ is a transition function shown in 4.2.

Step 2: After replacing +R and−R rules with +r and−r rules, respectively (see 4.3), compute an underestimation access control automaton $\underline{M^\Gamma}$ and an overestimation access-control automaton $\overline{M^\Gamma}$ as shown in 4.3.

Step 3: Create E^r from each XPath expression r by first creating a regular expression from the overestimation $\overline{r}$ of r and then appending $\cdot(\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})$ when r occurs in a RETURN clause.

Step 4: The XPath expression r is *always granted* when $L(E^r) \cap (L(M^G) \setminus L(\underline{M^\Gamma}))$ is empty, *always denied* when $L(E^r) \cap (L(M^G) \cap L(\overline{M^\Gamma}))$ is empty, and *statically indeterminate* otherwise.

Fig. 3. Framework of the analysis. Summary of Steps 1 thru 4.

as exposed by the access-control policies. Query regular expressions represent paths to elements or attributes as accessed by the queries.

When schemas are not available, we skip Step 1 and do not use the schema automata in Step 4.

4.1 Automata and XPath Expressions

In preparation, we introduce automata and show how we capture XPath expressions by using automata. Note that automata in this paper are not tree automata but rather traditional automata, which accept strings.

A *nondeterministic finite state automaton* (NFA) M is a tuple $(\Omega, Q, Q^{\text{init}}, Q^{\text{fin}}, \delta)$, where Ω is an alphabet, Q is a finite set of *states*, $Q^{\text{init}} \subseteq Q$ is a set of

initial states, $Q^{\text{fin}} \subseteq Q$ is a set of *final states*, and δ is a *transition function* from $Q \times \Omega$ to the power set of Q [Hopcroft and Ullman 1979]. The set of strings accepted by M is denoted $L(M)$.

Recall that we have allowed only three axes of XPath (see Section 2.3). This restriction allows us to capture XPath expressions with automata. As long as an XPath expression contains no predicates, we can easily construct an automaton from it. We first create a regular expression by replacing "/" and "//" with "$\cdot$" and "$\cdot\,(\Sigma^E)^*\cdot$," respectively, where "$\cdot$" denotes the concatenation of two regular sets, and then create an automaton from this regular expression. The constructed automaton accepts a path if, and only if, it matches the XPath expression.

When an XPath expression r contains predicates, we cannot capture its semantics exactly by using an automaton. However, we can still approximate r by constructing an *overestimation* $\overline{r}$ and an *underestimation* $\underline{r}$ and then construct automata for them. To construct $\overline{r}$, we assume that predicates are always satisfied. That is, $\overline{r}$ is created by removing all of the predicates from r. Obviously, $\overline{r}$ accepts all paths matching r and may accept other paths (overestimation). For example, if r is `/record[...]`, then $L(\overline{r}) = \{$`/record`$\}$.

Meanwhile, to construct $\underline{r}$, we assume that the predicates occurring in r are never satisfied. That is, if some step in r contains one or more predicates, $\underline{r}$ is $()$, which is an XPath 2.0 expression denoting the empty set (underestimation). If no steps in r contain any predicates, $\underline{r}$ is r. For example, if r is `/record[...]`, then $L(\underline{r})$ is the empty set.

As a special case, when r does not contain any predicates, $\overline{r}$ is identical to $\underline{r}$, and we simply write r to denote either expression.

4.2 Step 1: Creating Schema Automata

Since we are interested in permissible paths rather than permissible trees, we construct a *schema automaton* from a schema. A schema automaton accepts permissible paths rather than permissible documents.

Let $G = (N, \Sigma^E, \Sigma^A, S, P)$ be a schema. To construct a schema automaton from G, we use all nonterminals (i.e., N) of G as final states. We further introduce an additional final state q^{fin} and a start state q^{ini}. Formally, the schema automaton for G is

$$M^G = (\Sigma^E \cup \Sigma^A, N \cup \{q^{\text{ini}}, q^{\text{fin}}\}, \{q^{\text{ini}}\}, N \cup \{q^{\text{fin}}\}, \delta),$$

where δ is a transition function from $(N \cup \{q^{\text{ini}}, q^{\text{fin}}\}) \times (\Sigma^E \cup \Sigma^A)$ to the power set of $N \cup \{q^{\text{ini}}, q^{\text{fin}}\}$ such that

$$\delta(x, \mathtt{e}) = \{x' \mid \mathtt{e}[x'] \text{ occurs in } r \text{ for some } x \to r\mathcal{A} \text{ in } P\} \cup \{x' \mid x = q^{\text{ini}}, \mathtt{e}[x'] \in S\},$$
$$\delta(x, \mathtt{a}) = \{q^{\text{fin}} \mid \mathtt{a} \in \mathcal{A} \text{ for some } x \to r\ \mathcal{A} \text{ in } P\},$$

where e is an element name in Σ^E and a is an attribute name in Σ^A.

For example, consider the example schema in Section 2. The schema automaton for this schema is

$$M^G = (\Sigma^E \cup \Sigma^A, N \cup \{q^{\text{ini}}, q^{\text{fin}}\}, \{q^{\text{ini}}\}, N \cup \{q^{\text{fin}}\}, \delta)$$

where

$\Sigma^E = \{$`record`, `diagnosis`, `chemotherapy`, `comment`, `pathology`, `prescription`$\}$,
$\Sigma^A = \{$`@type`$\}$,
$N = \{$Record, Diag, Chem, Com, Patho, Presc$\}$,
$\delta(q^{\text{ini}},$ `record`$) = \{$Record$\}$, $\delta($Record, `diagnosis`$) = \{$Diag$\}$,
$\delta($Record, `chemotherapy`$) = \{$Chem$\}$, $\delta($Record, `comment`$) = \{$Com$\}$,
$\delta($Record, `record`$) = \{$Record$\}$, $\delta($Diag, `pathology`$) = \{$Patho$\}$,
$\delta($Diag, `comment`$) = \{$Com$\}$, $\delta($Chem, `prescription`$) = \{$Presc$\}$,
$\delta($Chem, `comment`$) = \{$Com$\}$, $\delta($Patho, `@type`$) = \{q^{\text{fin}}\}$.

Observe that this automaton accepts the following paths.

```
/record,
/record/comment,
/record/diagnosis,
/record/diagnosis/pathology,
/record/diagnosis/pathology/@type,
/record/diagnosis/comment,
/record/chemotherapy,
/record/chemotherapy/prescription,
/record/chemotherapy/comment,
/record/record,
/record/record/comment,...
/record/record/record,
/record/record/record/comment,...
```

Since the example schema in Section 2 allows `record` elements to nest freely, this automaton allows an infinite number of paths.

4.3 Step 2: Creating Access-Control Automata

An access-control policy consists of rules, each of which applies to some roles. For each role, we create a pair of automata: an *underestimation access-control automaton* and an *overestimation access-control automata*. This pair captures the set of those paths to elements or attributes that are exposed by the access-control policy.

In preparation, we replace +R and −R rules with +r and −r rules, respectively (see Section 3.2). Let $r_1, \ldots, r_m$ be the XPath expressions occurring in the grant rules (`+r`) and let $r'_1, \ldots, r'_n$ be the XPath expressions occurring in the denial rules (`-r`).

We first assume that none of $r_1, \ldots, r_m, r'_1, \ldots, r'_n$ contain predicates. Recall that we interpret the policy according to the "denial-takes-precedence" principle[7] M^Γ accepts those paths, which are allowed by one of $r_1, \ldots, r_m$, but

[7]If we adopt the "grant-takes-precedence" principle, M^Γ is modified as follows:

$$L(M^\Gamma) \;=\; L(M[r_1]) \cup \cdots \cup L(M[r_m]) \cup \neg\,(L(M[r'_1]) \cup \cdots \cup L(M[r'_n]))$$

The rest of our static analysis remains the same.

are denied by any of $r'_1, \ldots, r'_n$. Formally,

$$L(M^\Gamma) = (L(M[r_1]) \cup \cdots \cup L(M[r_m])) \setminus (L(M[r'_1]) \cup \cdots \cup L(M[r'_n]))$$

where $\Sigma = \Sigma^E \cup \Sigma^A$ and "\" denotes the set difference. We can construct M^Γ by applying Boolean operations to $M[r_1], \ldots, M[r_m], M[r'_1], \ldots, M[r'_n]$.

We demonstrate this construction for the access-control policy in Section 3.1. For the role *Intern*, this policy contains a grant rule and a denial rule, both of which propagate downward. The grant rule contains an XPath `/record`, while the denial rule contains an XPath `//comment`. Thus,

$$L(M^\Gamma) = \{\texttt{record}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \setminus (\Sigma^E)^* \cdot \{\texttt{comment}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})$$

Now, let us consider the case that predicates occur in $r_1, \ldots, r_m, r'_1, \ldots, r'_n$. Since predicates cannot be captured by automata, we have to construct an overestimation access-control automaton $\overline{M^\Gamma}$ as well as an underestimation access-control automaton $\underline{M^\Gamma}$. Rather than exactly accepting the set of exposed paths, these automata overestimate and underestimate this set, respectively. Observe that $L(M[r_1]), \ldots, L(M[r_m])$ are positive atoms and $L(M[r'_1]), \ldots, L(M[r'_n])$ are negative atoms in the above equation. To construct an under estimation access-control automaton $\underline{M^\Gamma}$, we underestimate *positive* atoms and overestimate *negative* atoms. On the other hand, to construct an overestimation access-control automaton $\overline{M^\Gamma}$, we overestimate *positive* atoms and underestimate *negative* atoms. Formally,

$$L(\underline{M^\Gamma}) = (L(M[\underline{r_1}]) \cup \cdots \cup L(M[\underline{r_m}])) \setminus (L(M[\overline{r'_1}]) \cup \cdots \cup L(M[\overline{r'_n}]))$$
$$L(\overline{M^\Gamma}) = (L(M[\overline{r_1}]) \cup \cdots \cup L(M[\overline{r_m}])) \setminus (L(M[\underline{r'_1}]) \cup \cdots \cup L(M[\underline{r'_n}]))$$

Again, we can construct $\underline{M^\Gamma}$ and $\overline{M^\Gamma}$ by applying Boolean operations to the automata occurring in the right-hand side of the above equations.

Suppose that the grant rule and denial rules in the example policy use `/record[...]` and `//comment[...]`, respectively. Then,

$$\begin{aligned} L(\underline{M^\Gamma}) &= \emptyset \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \setminus (\Sigma^E)^* \cdot \{\texttt{comment}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \\ &= \emptyset, \\ L(\overline{M^\Gamma}) &= \{\texttt{record}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \setminus (\Sigma^E)^* \cdot \emptyset \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \\ &= \{\texttt{record}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \end{aligned}$$

4.4 Step 3: Creating Query Regular Expressions from the XQuery Queries

Given a FLWOR expression of XQuery, we first extract the XPath expressions occurring in it. If an XPath expression contains variables, we replace each of them with the XPath expression associated with that variable.

It is important to distinguish XPath expressions in RETURN clauses and those in other (FOR, LET, ORDER, and WHERE) clauses. XPath expressions in FOR, LET, ORDER, or WHERE clauses examine elements or attributes, but do not access their subordinate elements. On the other hand, XPath expressions in RETURN clauses return subtrees including subordinate elements and attributes.

Let us create a query regular expression E^r for each r of the extracted XPath expressions. If r occurs in a FOR, LET, ORDER, or WHERE clause, then E^r is a regular expression created from $\overline{r}$. Observe that we overestimate r, since we would like to err on the safe side in our static analysis. When r occurs in a RETURN clause, we want to capture those paths which reference to descendants of nodes matching $\overline{r}$. Thus, E^r is created by appending $\cdot(\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})$ to the regular expression created from $\overline{r}$.

As an example, consider the XQuery expression given in Section 2.4. From this XQuery expression, we extract the following XPath expressions. Observe that the variable `$r` is expanded.

FOR-LET-ORDER-WHERE:

```
/record
/record/diagnosis/pathology/@type
```

RETURN.

```
/record/diagnosis/pathology
/record//comment
```

Let r be the query regular expression for `/record//comment`, which is the second XPath expression occurring in the RETURN clause. Then, E^r accepts `/record/comment`, `/record/comment/@type`, `/record/comment/record`, `/record/comment/diagnosis`, and so forth.

It is not always possible to extract a finite set of XPath expressions from a given XQuery query, since the query may be recursive or invoke itself for different nodes. Such recursive queries cannot be handled by our static analysis.

4.5 Step 4: Comparing the Schema Automata, Access-Control Automata, and Query Regular Expressions

We are now ready to compare schema automata, access-control automata, and query regular expressions. For simplicity, we first assume that predicates do not appear in the access-control policy.

The XPath expression r is *always granted* if every path accepted by both the query regular expression E^r and schema automaton M^G is accepted by the access-control automaton M^Γ, that is,

$$L(E^r) \cap L(M^G) \subseteq L(M^\Gamma)$$

However, we use an equivalent formula shown below, since it can be much more efficiently tested (see Appendix).

$$L(E^r) \cap (L(M^G) \setminus L(M^\Gamma)) = \emptyset$$

When schemas are unavailable, we assume that M^G allows all paths and examine whether

$$L(E^r) \cap (\neg L(M^\Gamma)) = \emptyset$$

where $\neg$ denotes the complement set.

The XPath expression r is *always denied* if no path is accepted by all of the query regular expression E^r, schema automaton M^G, and access-control automaton M^Γ. That is,

$$L(E^r) \cap L(M^G) \cap L(M^\Gamma) = \emptyset$$

When schemas are unavailable, we examine whether

$$L(E^r) \cap L(M^\Gamma) = \emptyset$$

The path expression r is *statically indeterminate* if it is neither always granted nor always denied.

As an example, we use the XQuery expression in Section 2.4, the DTD in Section 2.2, and the access-control policy in Section 3.1. We have already constructed a schema automaton in Section 4.2, an access-control automaton in Section 4.3, and a query automaton for `/record//comment` in Section 4.4. It can be easily seen that $L(M^r) \cap L(M^G)$ is a singleton set containing `/record/comment` and that $L(M^\Gamma)$ does not contain this path. Thus, the last XPath expression (`$r//comment`) in the example query is always denied.

When predicates appear in the access-control policy, we have to use $\overline{M^\Gamma}$ and $\underline{M^\Gamma}$ rather than M^Γ. We use an underestimation $\underline{M^\Gamma}$ when we want to determine whether or not a query is always granted. That is, we examine whether

$$L(E^r) \cap (L(M^G) \setminus L(\underline{M^\Gamma})) = \emptyset$$

When schemas are unavailable, we examine whether

$$L(E^r) \cap (\neg L(\underline{M^\Gamma})) = \emptyset$$

Likewise, we use an overestimation $\overline{M^\Gamma}$ when we determine whether or not a path expression is always denied. That is, we examine whether

$$L(E^r) \cap (L(M^G) \cap L(\overline{M^\Gamma})) = \emptyset$$

When schemas are unavailable, we examine if

$$L(E^r) \cap L(\overline{M^\Gamma}) = \emptyset$$

4.6 Query Optimization

When an XPath expression r in a XQuery expression is always denied, we can replace r by an empty list. This rewriting makes it unnecessary to evaluate r as well as unnecessary to perform run-time checking of the access-control policy for r, and may trigger further optimizations if we have an optimizer for XQuery.

Recall our example XQuery expression in Section 2.4. When the role is *Doctor*, static analysis reports that every XPath expression is always granted. Run-time checking is thus unnecessary. If the role is *Intern*, static analysis reports that the last XPath expression is always denied. We can thus rewrite the query as follows. Observe that comments are not returned by this rewritten query.

```
<TreatmentAnalysis>
{
  for $r in document("medical_record")/record
  where $r/diagnosis/pathology/@type="Gastric Cancer"
  return
    $r/diagnosis/pathology
}
</TreatmentAnalysis>
```

5. VALUED-BASED ACCESS CONTROL

In this section, we extend static analysis to handle value-based access control.

Value-based access-control requires that XPath expressions in the access-control policy contain predicates, which, in turn, contain values. Query expressions may also use XPath predicates containing values. We have approximated such access-control policies and query expressions by creating underestimation and overestimation automata. However, these approximations make some queries statically indeterminate.

For example, recall the example of value-based access-control policy in Section 3.1.1. That access-control policy is shown again here.

Role: *patient*
`+R, /record[@patientId = $userid]`

Suppose that a patient wants to issue this query:

```
<AboutMe>
{
  for $r in document("medical_record")/record[@patientId = $userid]
  return
    $r/diagnosis
}
</AboutMe>
```

It is obvious that this query does not access those elements or attributes that are hidden by the access-control policy. However, our static analysis fails to report that the query is always granted. The reason is over- and underestimation. As we have already seen in Section 4.3, the underestimation access-control automata accepts no paths; i.e., $L(\underline{M^{\Gamma}}) = \emptyset$. On the other hand, the query regular expression E^r for `/record[@patientId = $userid]` in the FOR clause is computed by overestimation, i.e., $L(E^r) = \{\texttt{record}\}$. Thus, $L(E^r) \cap (L(M^G) \setminus L(\underline{M^{\Gamma}})) = \emptyset$ does not hold.

However, when an access-control policy and a query expression specify the same predicate, we can incorporate it into the underlying alphabet. In the above example, if we consider `record[@patientId = $userid]` as a "symbol" (say `ownRecord`), we have $L(M^{\Gamma}) = \{\texttt{ownRecord}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})$ and $L(M^r) = \{\texttt{ownRecord}\}$, thus ensuring $L(E^r) \cap (L(M^G) \setminus L(\underline{M^{\Gamma}})) = \emptyset$.

On the basis of this observation, we extend our static analysis for handling value-based access control. This extension is effected as a preprocessing step before executing Steps 1 thru 4, as shown in Section 4.

1. First, we find "symbols" from the access-control policy and query. For each tag name t, we try to find a predicate p such that
   —$t[p]$ appears in both of the access-control policy and query,
   —$t[\text{not}(p)]$ appears in both of the access-control policy and query,
   —$t[p]$ appears in the access-control policy while $t[\text{not}(p)]$ appears in the query, or
   —$t[\text{not}(p)]$ appears in the access-control policy while $t[p]$ appears in the query.
2. Second, we introduce symbols t_1 and t_2 for capturing $t[p]$ and $t[\text{not}(p)]$, respectively.
3. Third, we rewrite the schema. This is done by replacing t with t_1 and t_2. Formally, we replace each $t[x]$ in the production rules with $t_1[x] \mid t_2[x]$.
4. Fourth, we rewrite the access-control policy and query. We replace t without predicates by $t_1 \mid t_2$, and replace $t[p]$ and $t[\text{not}(p)]$ with t_1 and t_2, respectively.

As an example, let us consider the schema in Section 2.2 as well as the access-control policy and query shown above. First, we find `record[@patientId = $userid]` in this access-control policy, and choose this and `record[not (@patientId = $userid)]` as "symbols," denoted by `record1` and `record2`, respectively.

Second, we rewrite the schema in Section 2.2 as $G_2 = (N_2, \Sigma_2^E, \Sigma_2^A, S_2, P_2)$, where

$$
\begin{aligned}
N_2 &= \{\text{Record, Diag, Chem, Com, Patho, Presc}\},\\
\Sigma_2^E &= \{\texttt{record1}, \texttt{record2}, \texttt{diagnosis}, \texttt{chemotherapy},\\
&\qquad \texttt{comment}, \texttt{pathology}, \texttt{prescription}\},\\
\Sigma_2^A &= \{\texttt{@patientId}, \texttt{@type}\},\\
S_2 &= \{\texttt{record1}[\text{Record}] \mid \texttt{record2}[\text{Record}]\},\\
P_2 &= \{\text{Record} \to (\texttt{diagnosis}[\text{Diag}]^*,\\
&\qquad\qquad \texttt{chemotherapy}[\text{Chem}]^*,\\
&\qquad\qquad \texttt{comment}[\text{Com}]^*,\\
&\qquad\qquad (\texttt{record1}[\text{Record}] \mid \texttt{record2}[\text{Record}])^*\ \{\texttt{@patientId}\},\\
&\quad \text{Diag} \to (\texttt{pathology}[\text{Patho}], \texttt{comment}[\text{Com}]^*)\ \emptyset,\\
&\quad \text{Chem} \to (\texttt{prescription}[\text{Presc}]^*,\\
&\qquad\qquad \texttt{comment}[\text{Com}]^*)\ \emptyset,\\
&\quad \text{Com} \to \epsilon\ \emptyset,\ \ \text{Patho} \to \epsilon\ \{\texttt{@type}\},\ \ \text{Presc} \to \epsilon\ \emptyset\}
\end{aligned}
$$

Third, we rewrite the access-control policy as

```
Role: patient
      +R, /record1/diagnosis
```

and further rewrite the query as

```
<AboutMe>
{
  for $r in document("medical_record")/record1
  return
    $r/diagnosis
}
</AboutMe>
```

The rest of static analysis is straightforward. The access-control automaton M^{Γ} accepts $\{\texttt{record1}\}\cdot(\Sigma^{E})^{*}\cdot(\Sigma^{A}\cup\{\epsilon\})$. The query automaton M^{r} for the XPath expression `/record1` in the FOR clause accepts $\{\texttt{record1}\}$ and the query regular expression $E^{r'}$ for the XPath expression `$r/diagnosis` in the RETURN clause accepts $\{\texttt{record1}\}\cdot(\Sigma^{E})^{*}\cdot(\Sigma^{A}\cup\{\epsilon\})$, respectively. Since $L(M^{r})\cap(L(M^{G})\setminus L(M^{\Gamma}))=\emptyset$ and $L(M^{r'})\cap(L(M^{G})\setminus L(M^{\Gamma}))=\emptyset$ obviously hold, our static analysis can successfully report that the query is always granted.

In this example, the rewritten access-control policy and the query do not have predicates and, thus, we did not need over- or underestimation. However, the preprocessing for value-based access control does not always eliminate predicates completely. Thus, we may still need over- and underestimation, but these estimation are made more proper.

Our preprocessing technique may lead to indeterminate results when values are not normalized. For example, consider another policy shown below:

Role: *patient*
`+R, /record[@patientId = "100"]`

This policy specifies `"100"` as the value of the attribute `parentId`. Meanwhile, the following query specifies `" 0100."`

```
<AboutMe>
{
  for $r in document("medical_record")/record[(@patientId = " 0100 ")]
  return
    $r/diagnosis
}
</AboutMe>
```

Since `"100"` are `"0100"` are different values, our preprocessing fails to find common "symbols." Thus, our static analysis has to rely on over- and underestimation, thereby failing to report that the new query is always granted.

To remedy this problem, values in documents, queries, and access-control policies have to be normalized in advance. Several XML normalizations [Boyer 2001; Atkinson 2002] are already available.

6. VIEW SCHEMA

Hitherto we have concentrated on static access control, which eases the burden of run-time access control. In this section, we address a different, but equally

important problem. Specifically, we attempt to help the programmer by eliminating superfluous information from schemas.

Recall that a schema defines the set of permissible XML documents in terms of elements, attributes, and their structural relationships. When access control is present, however, the elements or attributes permitted by the schemas are not always accessible to the programmer. In other words, the exposed documents are *different* from the documents permitted by the schema. Such a schema is not only confusing, but may also allow malicious programmers to guess hidden information [Fan et al. 2004].

To overcome this problem, we derive a *view schema* from an input schema. A view schema is equivalent to the input schema, except that it does not allow those elements and attributes that are hidden by the policy.

The key idea for constructing view queries is to simulate the execution of the access-control automaton, as well as the derivation of the schema.[8] This is done by using (nonterminal, state) pairs as "nonterminals," where nonterminals are taken from the schema and states are borrowed from the access-control automaton. Observe that a "nonterminal" comprising a nonfinal state is used only for deriving access-denied elements or attributes. A view schema can then be obtained by renaming access-denied elements and deleting access-denied attributes.

Formally, the view schema G^Γ is defined as follows. Let the access-control automaton M^Γ be a deterministic automaton $(\Sigma^A \cup \Sigma^E, Q, q_0, \delta, Q_F)$, where $q_0 \in Q$, $Q_F \subseteq Q$, and δ is a function from $Q \times (\Sigma^A \cup \Sigma^E)$ to Q. We first construct the product of G and M^Γ as below:

1. The set of nonterminals is the cross product of the nonterminal set N of G and the state set Q of M^Γ.
2. The underlying alphabets (namely, Σ^E and Σ^A) for elements and attributes are borrowed from G, but terminals not appearing in any production rules or start sets are deleted.
3. The start set is constructed from S as well as q_0 and δ; for every $e[x]$ in S, we introduce $e[(x, \delta(q_0, e))]$, where $\delta(q_0, e)$ simulates the execution of M^Γ from q_0 via e.
4. The set of production rules is constructed from P as well as δ. For every production rule $x \to r\ \mathcal{A}$ of G and every state q in M^Γ, a production rule is introduced. Its left-hand side is (x, q) and its right-hand side is obtained by replacing each $e[x']$ in r with $e[(x', \delta(q, e))]$, which simulates the execution of M^Γ from q via e.
5. Element or attribute names not appearing in any production rules are removed.

To create a view schema from this product, we only have to make the access-denied elements and attributes invisible. First, we rename elements that leads M^Γ to nonfinal states. The new names for invisible elements are `accessDenied`. Second, we delete attributes that leads M^Γ to nonfinal states.

[8]This approach is a special case of the schema transformation shown by the first author in Murata [2001].

We now formally introduce view schemas. Let

$$e^q = \begin{cases} e & (\delta(q,e) \in Q_F) \\ \texttt{accessDenied} & (\text{otherwise}). \end{cases}$$

A view schema is

$$G^\Gamma = (N', (\Sigma^E)', (\Sigma^A)', S', P')$$

where:

$$\begin{aligned}
N' &= N \times Q, \\
S' &= \{e^q[(x,q)] \mid e[x] \in S, q = \delta(q_0, e), q \in Q_F\}, \\
P' &= \{(x,q) \to r'\ \mathcal{A}' \mid x \to r\ \mathcal{A} \in P, q \in Q, \\
&\qquad r' \text{ is the regular expression obtained from } r \text{ by replacing} \\
&\qquad \text{each } e[x] \text{ occurring in } r \text{ with } e^q[(x, \delta(q,e))], \text{ and} \\
&\qquad \mathcal{A}' = \{a \in \mathcal{A} \mid \delta(q,a) \in Q_F\}\ \}, \\
(\Sigma^E)' &= \{e \in \Sigma^E \cup \{\texttt{accessDenied}\} \mid e \text{ occurs in } S' \text{ or } P'\}, \\
(\Sigma^A)' &= \{a \in \Sigma^A \mid a \text{ occurs in } P'\}.
\end{aligned}$$

6.1 Example

A view schema for our motivating example is $G_1 = (N_1, \Sigma_1^E, \Sigma_1^A, S_1, P_1)$, where

$$\begin{aligned}
N_1 &= \{\text{Record}_1, \text{Diag}_1, \text{Chem}_1, \text{Com}_1, \text{Patho}_1, \text{Presc}_1\}, \\
\Sigma_1^E &= \{\texttt{record}, \texttt{diagnosis}, \texttt{chemotherapy}, \\
&\qquad \texttt{pathology}, \texttt{prescription}, \texttt{accessDenied}\}, \\
\Sigma_1^A &= \{\texttt{@patientId}, \texttt{@type}\}, \\
S_1 &= \{\texttt{record}[\text{Record}_1]\}, \\
P_1 &= \{\text{Record}_1 \to (\texttt{diagnosis}[\text{Diag}_1]^*, \\
&\qquad\qquad \texttt{chemotherapy}[\text{Chem}_1]^*, \\
&\qquad\qquad \texttt{accessDenied}[\text{Com}_1]^*, \texttt{record}[\text{Record}_1]^*)\ \{\texttt{@patientId}\}, \\
&\qquad \text{Diag}_1 \to (\texttt{pathology}[\text{Patho}_1]^*, \texttt{accessDenied}[\text{Com}_1]^*)\ \emptyset, \\
&\qquad \text{Chem}_1 \to (\texttt{prescription}[\text{Presc}]_1^*, \texttt{accessDenied}[\text{Com}_1]^*)\ \emptyset, \\
&\qquad \text{Com}_1 \to \epsilon\ \emptyset, \quad \text{Patho}_1 \to \epsilon\ \{\texttt{@type}\}, \quad \text{Presc}_1 \to \epsilon\ \emptyset\}.
\end{aligned}$$

Note that the tag name `comment` is replaced with `accessDenied` in this view schema, since our access-control policy (shown in Section 3) has a denial rule (`-R, //comment`).

One might argue that `accessDenied` elements in view schemas hamper readability and that they be removed from the view schema. Indeed, we could have removed `accessDenied` elements from the previous view schema. However, since access-denied elements may have access-granted elements as descendants, we cannot remove `accessDenied` elements always. Only when the access-control policy satisfies *denial downward consistency* (i.e., it does not deny the access to elements without denying the access to their descendants as well), we can safely remove `accessDenied` elements from view schemas.

7. EXPERIMENTS

We have implemented our static analysis algorithm in Java (see Appendix). In this section, we present two experiments based on this implementation. In the first experiment, we evaluate how much the cost of query evaluations will be reduced by our static analysis and query optimization. In addition, we observe that view schemas eliminate superfluous information. In the second experiment, we measure the scalability of our static analysis for very large policies and schemas.

7.1 Effectiveness of Static Analysis

First, using a well-known collection of queries, we show which queries are made more efficient. Second, using an example document and the same collection of queries, we measure the number of nodes exempted from access or run-time checking.

7.1.1 *Settings.* We use a DTD and sample queries developed by the XMark project,[9] which is a well-known benchmark framework for XQuery based on an auction scenario. The DTD has 77 element types. An XML document valid against the DTD represents a list of auction items, participants' information, etc. There are 20 sample queries. For example, the following is Query #4.

```
for $b in document("auction.xml")//open_auction
where $b/bidder/personref[@person="person18829"]
  before $b/bidder/personref[@person="person10487"]
return <history>{$b/reserve/text()}</history>
```

We wrote a sample access-control policy for six roles, each of which was associated with 1 through 8 access-control rules. The policy is shown in Figure 4 and its semantics are summarized in Table I. Take, for example, the rules associated with the role *Seller*. The first rule says that a *Seller* is allowed to read the document root (`/`). Furthermore, this grant permission (+R) propagates downward, i.e., from the document root (`/`) to all other nodes. However, there are other rules with denial permission. Recall that the $\$userid$ variable represents the id of the user requesting the access. Therefore, a seller can read the contents of his own `//person/creditcard` and `//person/profile`, but not the credit cards and profiles of other users.

Note that this policy is an example of value-based access control. However, since none of the XPath predicates in it appear in the XMark queries, our static analysis relies on over- and underestimation, as described in Section 4.

7.1.2 *Queries Made Efficient.* For each query/role pair, we check whether or not our static analysis removes run-time checking. We performed the experiment for two cases: with the DTD and without the DTD. We statically analyzed all of the XPath expressions for each query.

Tables IIa and IIb show the results of our static analysis with and without the DTD, respectively. Each entry in the table indicates the result by either "G,"

[9]The XMark project page is available at `http://monetdb.cwi.nl/xml/`.

Role: *Maintainer*
+R, /

Role: *Member Mgmt.*
+r, /
+r, /site
+R, //people
+R, //open_auctions
+R, //closed_auctions

Role: *Item Mgmt.*
+r, /
+r, /site
+R, //regions
+R, //categories
+R, //catgraph

Role: *Seller*
+R, /
-R, //person[@id
!= $*userid*]/creditcard
-R, //person[@id
!= $*userid*]/watches
-R, //person[@id
!= $*userid*]/profile
-R, //bidder/personref
-R, //closed_auction[
seller/@person !=
$*userid*]/buyer
-R, //open_auction//privacy
-R, //closed_auction//happiness

Role: *Buyer*
+R, /
-R, //person[@id
!= $*userid*]/creditcard
-R, //person[@id
!= $*userid*]/watches
-R, //person[@id
!= $*userid*]/profile
-R, //bidder/personref
-R, //buyer[@person != $*userid*]
-R, //open_auctions//privacy
-R, //closed_auctions//happiness

Role: *Visitor*
+R, /
-R, //people
-R, //bidder/personref
-R, //seller
-R, //buyer
-R, //open_auctions//privacy
-R, //closed_auctions//happiness

Fig. 4. The definition of the sample policy.

"D," or "−."

- "G" indicates that all XPath expressions in the query are always granted.
- "D" indicates that at least one of the XPath expressions in the query is always denied and that all of the other XPath expressions are always granted.
- "−" indicates that at least one XPath expression in the query is statically indeterminate.

A query marked by "G" contains no XPath expressions requiring run-time checking. If a query is marked by "D," it contains XPath expressions that are always denied. However, in this case, we rewrite such expressions as null lists in advance. Finally, if queries are marked by "−," run-time checking is necessary, since its result is not predictable.

For example, we can see from Table IIa that the mark for Query #4 for role *IM* is "D." This means that when a user in the role *IM* makes Query #4, we can eliminate run-time checking by rewriting the query.

Table I. The Sample Access-Control Policy

	Role Name	Rule Semantics
1	*M* (*Maintainer*)	Access to all information is granted.
2	*MM* (*Member Mgmt.*)	Access to member info. (`//people`) or auction info. (`//open_auctions` and `//open_auctions`) is granted.
3	*IM* (*Item Mgmt.*)	Access to item info. (`//regions`, `//categories`, and `//catgraph`) is granted.
4	*S* (*Seller*)	A seller cannot see bidder info., privacy info., or personal info. (credit card info. and profiles). A seller can see who bought his item.
5	*B* (*Buyer*)	A buyer cannot see bidder info., privacy info., or personal info. A buyer can see his own purchases.
6	*V* (*Visitor*)	A visitor cannot see privacy info., or personal info, and cannot see who sells, bids or buys an item.

Table II. Results of Static Analysis of XMark Queries

	(a) With the DTD																			
Query#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
M	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
MM	G	G	G	G	G	D	—	G	D	G	G	G	D	D	G	G	G	G	D	G
IM	D	D	D	D	D	G	—	D	D	D	D	D	G	G	D	D	D	D	G	D
S	G	G	G	D	G	G	G	—	—	—	—	—	G	G	G	G	G	G	G	—
B	G	G	G	—	G	G	G	—	—	—	—	—	G	G	G	G	G	G	G	—
V	D	G	G	D	G	G	D	D	D	D	D	D	G	G	G	D	D	G	G	D
	(b) Without the DTD																			
Query#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
M	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G	G
MM	G	G	G	G	G	—	—	G	D	G	G	G	—	—	G	G	G	G	—	G
IM	—	—	—	—	—	G	—	—	—	—	—	—	G	—	—	—	—	—	G	—
S	G	G	G	D	G	—	—	—	—	—	—	—	—	—	G	—	—	G	—	—
B	G	G	G	—	G	—	—	—	—	—	—	—	—	—	G	—	—	G	—	—
V	D	G	G	D	G	—	—	D	D	D	D	D	—	—	G	—	D	G	—	D

Tables IIa and IIb show that 88 and 54% of the query/role pairs, respectively (i.e., "G" + "D"), do not require any run-time checking. Furthermore, for 27 and 9% of the query/role pairs (i.e., "D"), we can optimize the queries by rewriting.

From Table II b, we conclude that even when no DTDs are available, our static analysis can result in significant optimizations of the queries. From Table IIa, we conclude that the analysis can be further refined by exploiting the DTD information. Note that the sample policy contains XPaths with predicates, which cause over- and underestimation of the access-control automata. Even in such cases, our static analysis frequently makes run-time checking unnecessary.

7.1.3 *Nodes Exempted from Access or Run-Time Checking.* Here we consider how much the cost of query evaluation is reduced by our static analysis and query optimization. As the metric of reduced cost, we count the number of nodes exempted from access or run-time checking by our static analysis and query optimization.

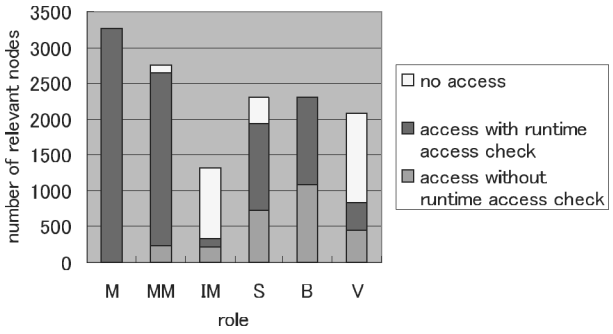


Fig. 5. Nodes exempted from access or run-time checking.

Table III. Size of View Schemas

Role	M	MM	IM	S	B	V
Number of element types in view schemas	77	46	32	74	74	51

As an example document, we use an auction document from the XMark project. This document has 250,000 nodes and was generated by specifying the factor parameter as 0.05 (see the XMark project page).

When a query is evaluated against this document, certain nodes in the document are accessed by the XPath expressions in the query. We classify these nodes into three groups, which are defined as follows:

- *no access.* These nodes are exempted from access. In other words, the original query accesses these nodes, but the rewritten query does not.
- *access without run-time checking.* These nodes are exempted from run-time checking of access-control policies, but they must still be read.
- *access with run-time checking.* These nodes are not exempted from run-time checking or accesses. In other words, the rewritten query accesses these nodes and these accesses require run-time checking.

The bar chart (Figure 5) shows, for each role, the number of nodes in the three categories. These are the averages for Queries #1 through #20.

We observe that the cost of query evaluation for roles *M*, *MM*, *IM*, and *V* is reduced significantly, because the third portion is very small. In particular, in the case of *Maintainer* (*M*) we do not require run-time checking at all, because a maintainer has access to all nodes. On the other hand, for *IM* and *V*, we have a large number of skipped nodes that do not even need to be examined during the query evaluation.

Finally, we study whether view schemas eliminate superfluous information contained by the original schema. Since we have *not* implemented the algorithm (shown in Section 6) for constructing view schemas, we created view schemas by hand.

The number of element types in the original schema (DTD) is 77. Table III summarizes the size of view schemas for each role. When the role is *MM*, *IM*, or *V*, view schemas are significantly smaller than the original schema. Otherwise,

```
Role: ROLE10
  +R,/
  -R,/spec/body/div1/constraintnote/definitions
  -R,/spec/header/latestloc/xtermref
  -R,/spec/body/div1/definitions/exception
  -R,//copyright
  -R,/spec/header/prevlocs
  -R,/spec/header/revisiondesc
  -R,/spec/body/div1/vcnote/definitions/reference
  -R,/spec/header/authlist/author
  -R,/spec/back/div1/ulist/item/vcnote/glist
              /gitem/def/table/tbody/tr/td/wfcnote
  -R,//inform-div1/wfcnote
```

Fig. 6. Random policy example.

view schemas are almost the same as the original schema. From this result, we conclude that view schemas are, at least occasionally, very useful in eliminating superfluous information.

7.2 Scalability of Static Analysis

In the scalability test, we measure the running-time of our analysis itself. We use real-world DTDs and random policies with large sets of rules.

In this test, we distinguish two phases of the analysis, and examine each phase independently. The first phase is an initialization phase (Steps 1 and 2), where we first compute a schema automaton M^G, then compute an access-control automaton M^Γ for each role in the policy. The second phase is the analysis phase (Steps 3 and 4), where we statically analyze the XPath expressions in each query to determine whether they are always denied or always granted. When there are many queries, we cache M^G and M^Γ, which we computed in the initialization phase, and, later in the analysis phase, we repeatedly use them.

7.2.1 *Settings.* Three large DTDs were used, the xmlspec-v21.dtd from W3C XML Working Group [Bray et al. 2004], which has 157 element types, the version 1.2 of cXML.dtd (commerce XML),[10] which has 378 element types, and the version 4.2 of docbookx.dtd by OASIS DocBook technical committee, [11] which has 393 element types.

We use access-control policies with different sizes, with 1 through 500 rules per role. For each of the given DTDs, 10 access-control policies are randomly generated by using element names or attribute names defined in the DTD. As an example, Figure 6 shows an access-control policy generated from the xmlspec-v21.dtd.

For each of the DTDs, we statically analyze a query with 12 XPath expressions. Each query is derived from the Query #10 of XMark, in which each XPath expression has one // and several /. We chose element names appearing in these XPath expressions according to the corresponding DTDs.

[10] `http://www.cxml.org`.

[11] `http://www.oasis-open.org/committees/docbook`.

Table IV. Step 1

DTD	Elapsed Time [ms]	# of States of M^G
xmlspec-v21.dtd	180	214
cxml.dtd	204	623
docbookx.dtd	332	501

7.2.2 *Results.* Our test environment was a 2.4 GHz Pentium 4 machine with 512 Mbytes memory and the J2RE 1.4.0 IBM build for Linux. To maximize the performance of the JIT compiler, we performed several runs before the measured runs.

The initialization phase consists of Steps 1 and 2. Table IV shows the running times for creating schema automata M^G (Step 1), which is not affected by the access-control policies. This table also shows the number of states of M^G. Figure 7(a) shows the running times for creating access-control automata (Step 2). Each point indicates the time required to compute an access-control automaton for each role in the randomly generated policies.

Figure 7(b) shows the running times for the analysis phase (Steps 3 and 4), where each point indicates the average time required for analyzing each XPath expression in the query.

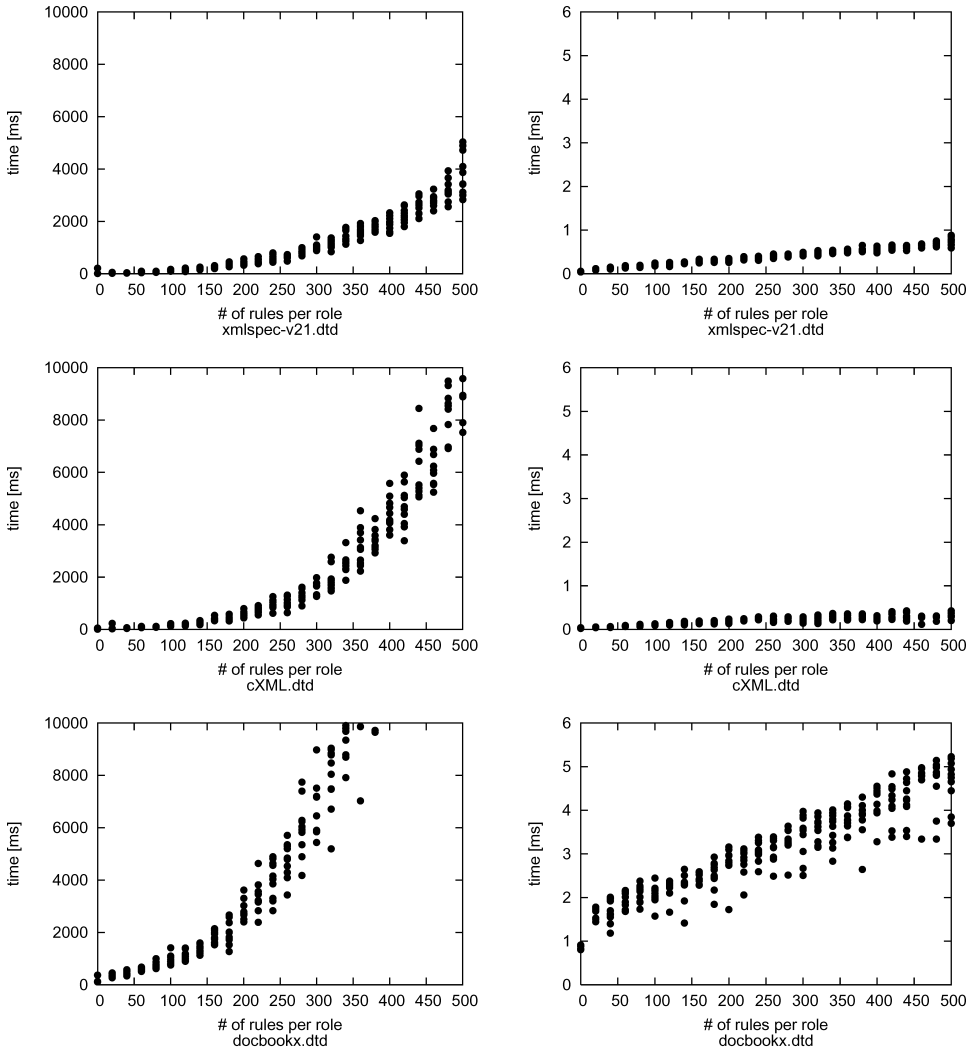
In both phases, the performance is much better for cXML.dtd than for xmlspec-v21.dtd or docbookx.dtd. This is because xmlspec-v21.dtd and docbookx.dtd contain many recursive definitions and are more complicated than cXML.dtd.

The initialization phase (computing M^G and M^Γ) takes more than 10 s for large policies and the running times increases nonlinearly. On the other hand, in the analysis phase, the running times increases almost linearly with the number of rules. In the real world, we perform the initialization phase just once per policy, while we perform the analysis phase once per XPath expression in queries. Therefore, we conclude that our static analysis scales with respect to the size of the schemas and the access-control policies.

8. CONCLUSIONS AND FUTURE WORK

In this paper, we have attempted to ease the burden of checking access-control policies for XML documents by distributing the burden between static analysis and run-time checks. The key idea for our static analysis is to use automata for representing and comparing queries, access-control policies, and schemas. We have built a prototype of our static analysis, demonstrated its effectiveness, and experimented with its performance. Our experiment (shown in Section 7) reveals that (1) static analysis frequently makes run-time checks unnecessary and also provides significant optimizations, and (2) our prototype scales nicely when the schemas, access-control policies, and queries are large.

We have also attempted to help the programmer by eliminating superfluous information from the schemas. This is done by creating view schemas from the input schemas and access-control policies. In other words, we derive view schemas from input schemas by enforcing access-control policies statically.



(a) Elapsed time for Step 2

(b) Elapsed time for Step 3 and 4

Fig. 7. Results of scalability tests.

However, our static analysis has some limitations. We summarize these limitations.

- *Simple types.* Simple types (e.g., integers) in schemas are ignored in constructing schema automata. This omission makes static analysis imprecise. For example, suppose that one of the access-control rules specifies `/record[@patientId = "foo"]` as the object and that the schema specifies integers as the simple type for the `patiendId` attribute. Obviously, this equality never holds. However, our static analysis assumes that it may hold.
- *Backward axes of XPath.* Our static analysis does not cover all axes (e.g., backward axes) of XPath. Although we can use tree automata (rather than

string automata) to capture all of the axes of XPath, tree automata are more complicated and make implementations significantly harder. However, as a special case, we can easily handle some of the backward axes by rewriting backward axes as forward ones [Olteanu et al. 2002].

- *The interaction of predicates and schemas.* XPath predicates may reference to descendant elements and attributes, which are controlled by schemas. However, our static analysis does not consider the interaction of predicates and schemas. For example, suppose that one of the access-control rules specifies `//foo[a and b]` as the object and that the schema specifies either `a` or `b`, but not both, as the child of `foo`. Since the predicate `[a and b]` requires the presence of *both* `a` and `b`, this rule is never used. However, our static analysis assumes that it *may* be used.
- *Recursive queries.* We have significantly simplified XQuery here, but the full set of XQuery allows recursive queries. Since we cannot extract a finite set of XPath expressions from a recursive query, we are forced to rely on run-time checking.

Our next step is to incorporate static analysis as part of an XML database system and seek a good balance between run-time checking and static analysis.

APPENDIX

Implementation

Here we present a few techniques for improving the performance of our implementation. These techniques contribute substantially to the performance and scalability reported in Section 7.2.

Our static analysis is built on top of an automata library. It provides the Boolean operations ($\cap$, $\cup$, $\setminus$) as well as the determinization and minimization operations.

From our experience, the performance of our static analysis largely depends on the minimization operation, since Step 2 performs this operation repeatedly. To improve its performance, we use a very efficient algorithm by Hopcroft [1971]. In addition, in constructing intersection ($\cap$) and difference ($\setminus$) automata, we avoid minimization when we can make automata small enough by removing redundant states (i.e., unreachable states and deadend states).

While computing $\overline{M^\Gamma}$ and $\underline{M^\Gamma}$ in Step 2 (see Section 4.3), we always determinize intermediate automata and finally determine $\overline{M^\Gamma}$ and $\underline{M^\Gamma}$. By doing so, we can efficiently perform $\setminus$ in Step 4 (see Section 4.5). If $\overline{M^\Gamma}$ and $\underline{M^\Gamma}$ were large nondeterministic automata, $\setminus$ (which requires determinization) would be prohibitively expensive.

Each formula in Step 4 (see Section 4.5) is of the form $L(E^r) \cap X = \emptyset$, where E^r is a query regular expression and X is a regular set (e.g., $L(M^G) \setminus L(M^\Gamma)$ or $L(M^G) \cap L(M^\Gamma)$). To examine if the formula holds, we could construct an automaton from E^r, compute the intersection automaton, and examine if the intersection automaton does not accept anything. In fact, our early implementation shown in Murata et al. [2003] was based on this approach. However,

since E^r (which is created from an XPath expression r) does not exploit the full power of the Kleene star operator ($*$), we can devise a significantly faster algorithm. It was inspired by the look-up table of Luo et al. [2004].

We assume that X is represented by a nondeterministic automaton M^X. For each u in $(\Sigma^E \cup \Sigma^A)^*$, let Q_u be the set of states that are reached by executing M^X against u. Let Q_{E^r} be the union of Q_u such that u matches E^r. The formula in question holds if, and only if, Q_{E^r} contains no final states of M^X.

It remains to construct Q_{E^r}. In preparation, for each state of M^X, we compute the set of states reachable (directly or indiretly) from this state by Σ^E transitions and store this set in a look-up table. We begin with the start state set of M^X as the current state set and examine E^r from the beginning. Recall that E^r is a sequence of "$(\Sigma^E)^*$" or symbols in $\Sigma^E \cup \Sigma^A$. When "$(\Sigma^E)^*$" is encountered, we construct the next state set by examining the look-up table for every state in the current state set. When a symbol in $\Sigma^E \cup \Sigma^A$ is encountered, we construct the next state set by applying the transition function of M^X to each state in the current state set. We have Q_{E^r} by repeating this construction until the end of E^r is reached.

REFERENCES

ALTINEL, M. AND FRANKLIN, M. J. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*. Morgan Kaufmann, Cairo. 53–64.

ATKINSON, B. 2002. Schema centric XML canonicalization version 1.0. OASIS Committee Specification. `http://uddi.org/pubs/SchemaCentricCanonicalization.htm`.

BAEZA-YATES, R. A. AND NAVARRO, G. 2002. XQL and proximal nodes. *Journal of the American Society for Information Science and Technology 53*, 6, 504–514.

BARTON, C., CHARLES, P., GOYAL, D., RAGHAVACHARI, M., FONTOURA, M., AND JOSIFOVSKI, V. 2003. An algorithm for streaming XPath processing with forward and backward axes. In *Proceedings of the 19th International Conference on Data Engineering*. IEEE Computer Society. 455–466.

BERTINO, E. 1992. Data hiding and security in object-oriented databases. In *Proceedings of the 8th International Conference on Data Engineering*. IEEE Computer Society, Tempe. 338–347.

BERTINO, E., CASTANO, S., FERRARI, E., AND MESITI, M. 1999. Controlled access and dissemination of XML documents. In *The 2nd Workshop on Web Information and Data Management*. ACM, New York. 22–27.

BERTINO, E., CASTANO, S., FERRARI, E., AND MESITI, M. 2001. Author-X: a Java-based system for XML data protection. In *14th IFIP Workshop on Database Security*. IFIP Conference Proceedings, vol. 201. Kluwer, Academic Publ., Boston, MA. 15–26.

BOAG, S., CHAMBERLIN, D., FERNÁNDEZ, M. F., FLORESCU, D., ROBIE, J., AND SIMÉON, J. 2003. XQuery 1.0: An XML query language. W3C working draft 12 November 2003. `http://www.w3.org/TR/xquery/`.

BOYER, J. 2001. Canonical XML version 1.0. W3C Recommendation. `http://www.w3.org/TR/xml-c14n/`.

BRAY, T., PAOLI, J., SPERBERG-MCQUEEN, C. M., MALER, E., AND YERGEAU, F. 2004. Extensible Markup Language (XML) 1.0. W3C Recommendation. `http://www.w3.org/TR/REC-xml`.

CHAN, C. Y., FELBER, P., GAROFALAKIS, M. N., AND RASTOGI, R. 2002. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the 18th International Conference on Data Engineering*. 225–234.

CLARK, J. 1999. XML Transformations (XSLT) version 1.0. W3C Recommendation. `http://www.w3.org/TR/xslt`.

CLARK, J. AND DEROSE, S. 1999. XML Path Language (XPath) version 1.0. W3C Recommendation. `http://www.w3.org/TR/xpath`.

CLARK, J. AND MURATA, M. 2001. RELAX NG specification. OASIS Committee Specification. http://www.oasis-open.org/committees/relax-ng/spec-20011203.html.

COMON, H., DAUCHET, M., GILLERON, R., JACQUEMARD, F., LUGIEZ, D., TISON, S., AND TOMMASI, M. 1997. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata. release October, 1st 2002.

DAMIANI, E., DI VIMERCATI, S. D. C., PARABOSCHI, S., AND SAMARATI, P. 2000. Securing XML documents. In *Proceedings of the 7th International Conference on Extending Database Technology (EDBT)*. Lecture Notes in Computer Science, vol. 1777. Springer, Konstanz. 121–135.

DEUTSCH, A. AND TANNEN, V. 2001. Containment and integrity constraints for XPath fragments. In *Proceedings of 8th International Workshop on Knowledge Representation Meets Databases*.

DRAPER, D., FRANKHAUSER, P., FERNÁNDEZ, M., MALHOTRA, A., ROSE, K., RYS, M., SIMEON, J., AND WADLER, P. 2004. XQuery 1.0 and XPath 2.0 formal semantics. W3C working draft 20 February 2004.

FAN, W. AND LIBKIN, L. 2002. On XML integrity constraints in the presence of DTDs. *J. ACM 49*, 3, 368–406.

FAN, W., CHAN, C. Y., AND GAROFALAKIS, M. N. 2004. Secure XML querying with security views. In *Proceedings of the 23rd SIGMOD International Conference on Management of Data, to appear*. ACM, New York.

FERNÁNDEZ, M. F. AND SUCIU, D. 1998. Optimizing regular path expressions using graph schemas. In *Proceedings of the 14th International Conference on Data Engineering*. 14–23.

GABILLON, A. AND BRUNO, E. 2001. Regulating access to XML documents. In *Proceedings of the 15th IFIP WG 11.3 Working Conference on Database Security*. 299–314.

GODIK, S. AND MOSES, T., EDS. 2003. Extensible access-control markup language (XACML) version 1.0. OASIS Standard http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml.

GOTTLOB, G., KOCH, C., AND PICHLER, R. 2002. Efficient algorithms for processing XPath queries. In *Proceedings of the 28th International Conference on Very Large Data Bases*. 95–106.

GREEN, T. J., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2002. Processing XML streams with deterministic automata. In *Proceedings of the 9th International Conference on Database Theory*. Springer-Verlag, New York. 173–189.

HOPCROFT, J. E. 1971. An $n \log n$ algorithm for minimizing states in a finite automaton. *Theory of Machines and Computations*. 189–196.

HOPCROFT, J. E. AND ULLMAN, J. D. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

HOSOYA, H. AND MURATA, M. 2002. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*.

HOSOYA, H. AND PIERCE, B. C. 2003. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology 3*, 2, 117–148.

KUDO, M. AND HADA, S. 2000. XML document security based on provisional authorization. In *Proceedings of the 7th Conference on Computer and Communications Security*. ACM, New York. 87–96.

LI, Q. AND MOON, B. 2001. Indexing and querying XML data for regular path expressions. In *Proceedings of the 27th International Conference on Very Large Databases*. 361–370.

LUO, B., LEE, D., LEE, W.-C., AND LIU, P. 2004. QFilter: Practical and scalable XML access controls via NFA-based query filtering. Tech. rep., Penn State University. Februrary.

MIKLAU, G. AND SUCIU, D. 2004. Containment and equivalence for a fragment of XPath. *J. ACM 51*, 1, 2–45.

MURATA, M. 2001. Extended path expressions for XML. In *Proceedings of the 20th Symposium on Principles of database systems*. Santa Barbara, CA. 126–137.

MURATA, M., TOZAWA, A., KUDO, M., AND HADA, S. 2003. XML access-control using static analysis. In *Proceedings of the 10th ACM Conference on Computer and Communication Security*. ACM Press, New York. 73–84.

MURATA, M., LEE, D., MANI, M., AND KAWAGUCHI, K. 2005. Taxonomy of XML schema languages using formal language theory. *ACM Transactions on Internet Technology 5*, 4, 660–704.

NAISHIN QI, M. K. 2005. XML access-control with policy matching tree. In *10th European Symposium On Research In Computer Security*.

NEVEN, F. AND SCHWENTICK, T. 2003. XPath containment in the presence of disjunction, DTDs, and variables. In *The 9th International Conference on Database Theory*. 315–329.

OLTEANU, D., MEUSS, H., FURCHE, T., AND BRY, F. 2002. XPath: Looking forward. In *Proceedings of the EDBT Workshop on XML Data Management (XMLDM)*. Vol. 2490. Springer, New York. 109–127.

PAPAKONSTANTINOU, Y. AND VASSALOS, V. 1999. Query rewriting for semistructured data. In *Proceedings of the 1999 ACM SIGMOD international conference on Management of data*. ACM Press, New York. 455–466.

RABITTI, F., BERTINO, E., KIM, W., AND WOELK, D. 1991. A model of authorization for next-generation database systems. *ACM Trans. Database Syst. 16*, 1, 88–131.

THOMPSON, H. S., BEECH, D., MALONEY, M., AND MENDELSOHN, N. 2001. XML Schema part 1: Structures. W3C Recommendation. `http://www.w3.org/TR/xmlschema-1/`.

WOOD, P. T. 2003. Containment for XPath fragments under DTD constraints. In *The 9th International Conference on Database Theory*. 297–311.

Received June 2004; revised August 2005; accepted March 2006