XML Access Control Using Static Analysis

Makoto Murata
IBM Tokyo Research Lab/IUJ Research Institute
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan
mmurata@trl.ibm.com

Akihiko Tozawa
IBM Tokyo Research Lab
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan
atozawa@trl.ibm.com

Michiharu Kudo
IBM Tokyo Research Lab
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan
kudo@jp.ibm.com

ABSTRACT

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 engines for XML query languages. 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 permitted by the schema but hidden by the access control policy. Static analysis can be performed without evaluating any query expression against an actual database. Run-time checking is required only when static analysis is unable to determine whether to grant or deny access requests. A nice side-effect of static analysis is query optimization: access-denied expressions in queries can be evaluated to empty lists at compile time. We have built a prototype of static analysis for XQuery, and shown the effectiveness and scalability through experiments.

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

Keywords

XML, XQuery, XPath, schema, automaton, access control, query optimization, static analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CCS'03, October 27–31, 2003, Washington, DC, USA.
Copyright 2003 ACM 1-58113-738-9/03/0010 ...$5.00.

1. INTRODUCTION

XML [5] has become an active area in database research. XPath [6] and XQuery [4] 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 rather than document-level or collection-level access control. We believe that access control plays an important role in XML database systems, as it does in relational database systems. Some early experiences [21, 10, 3] with access control for XML documents have been reported already.

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 an access to an element or an attribute is granted or denied by such 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. [21, 10]) for XML access control achieve expressiveness by using XPath [6] 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 an XML database is accessed at run time, a query engine is required to determine 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.

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 databases. Thus, static analysis can be performed at *compile time* (when a query expression is created rather than each time it is eval-

uated). Run-time checking is required only when static analysis is unable to grant or deny access requests without examining the actual databases. In addition, static analysis facilitates query optimization, since access-denied XPath expressions in queries can be rewritten as empty lists at compile time. In Section 5, we will demonstrate effectiveness of static analysis using examples.

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 automata from queries, access control policies, and (optionally) schemas: (1) automata created from queries, called *query automata*, 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 automata while applying the following rules: (1) accesses by queries are *always-granted* if the intersection of query automata and schema automata is subsumed by the access control automata; (2) they are *always-denied* if the intersection of query automata, schema automata, and access control automata is empty; and (3) they are *statically indeterminate*, otherwise.

1.1 Related Works

Fine-grained access control for XML documents has been studied by many researchers [2, 21, 3, 10, 15]. 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 assume that XML documents are in the main memory and can be examined repeatedly.

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 queries 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 provides network structures and class hierarchies. Access control frameworks for the OODBMS have appeared in the literature [1, 28]. Such frameworks typically rely on run-time analysis 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 use regular path expressions (XPath) for locating elements or attributes. We can thus use automata for uniformly handling schemas, queries, and access control policies.

Implementation techniques for XQuery and XPath are being actively studied. However, when compared to SQL engines, XQuery engines are far from mature. To the best of the authors' knowledge, none of the existing XQuery engines provide competitive optimization as well as access control.

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., [20, 12]), query optimization using schemas (e.g., [14]), query optimization using views (e.g., [27, 11, 23, 25, 30]), consistency between integrity constraints and schemas (e.g., [13]). 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 [11, 23, 25, 30] 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 under-estimation of access control automata.

1.2 Outline

The rest of this paper is organized as follows. After reviewing 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 further demonstrate the effectiveness and scalability of static analysis in Section 5. We discuss future extensions for handling value-based access control and the advanced features of XPath and XQuery, and conclude in Section 6.

2. PRELIMINARIES

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

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 are inserted in 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 [29], and RELAX NG [7] from OASIS (and now ISO/IEC) are notable examples of schema languages.

We do not use particular schema languages in this paper, but rather use tree regular grammars [9] as a formal model of schemas. Murata et al. [24] have shown that tree regular grammars can model DTD, W3C XML Schema, and RELAX NG.

A *schema* is a 5-tuple $G = (N, \Sigma^E, \Sigma^A, S, P)$, where:

- N is a finite set of *non-terminals*,

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

Figure 1: An XML document example

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

Production rules collectively specify permissible element structures. We separate non-terminals 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 non-terminals and element names, W3C XML Schema and RELAX NG require this separation. Unlike the definition in [24], we allow production rules to have a set of permissible attribute names[1] .

For the sake of simplicity, we do not handle text as values of elements or attributes in this paper. In the case of DTDs, this restriction amounts to the confusion of `#PCDATA` and `EMPTY`.

Example 1 A schema for our motivating example is $G_1 = (N_1, \Sigma_1^E, \Sigma_1^A, S_1, P_1)$, where

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

An equivalent DTD is shown below.

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

[1]RELAX NG provides a more sophisticated mechanism for handling attributes [19].

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.

2.3 XPath

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

XPath provides a restricted variation of regular path expressions. XPath is widely recognized in the industry and used by XSLT [8] 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 6. 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 emerged recently. 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 in the rest of this paper.

FLWR (FOR-LET-WHERE-RETURN) expressions are of central importance to XQuery. A FLWR expression consists of a FOR, LET, WHERE, 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 FLWR expression create tuples. The WHERE clause imposes additional conditions on tuples. Those tuples not satisfying the WHERE clause are discarded. Then, for each of the remaining tuples, the RETURN clause is evaluated and a value or sequence of values is returned.

The following query lists the pathology-comment pairs for the 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

In general, the access control policy consists of a set of access control rules and 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[2]:

```
(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`

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.

[2]The syntax of the policy can be represented in a standardized way using XACML[16] but we use the above syntax for simplicity.

```
<record>
  <diagnosis>
    <pathology type="Gastric Cancer">
      Well differentiated adeno carcinoma
    </pathology>
  </diagnosis>
  <chemotherapy>
    <prescription>
      5-FU 500mg and CDDP 10mg
    </prescription>
  </chemotherapy>
</record>
```

Figure 2: The XML document that *Intern* can see

3.1.1 Using XPath for XML Access Control

Many reports[21, 10, 3, 15] on the node-level access control for XML document use XPath to locate the target node in the XML document.

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 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 only to 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`[3]`] /diagnosis)`. This policy says that the access to a `diagnosis` element below the `record` element 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.

3.2 Semantics of Access Control Policy

In general, an access control policy should be designed to 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 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.

In this paper, we consider another requirement called *denial downward consistency*, which is a new requirement specific to XML access control. It requires that whenever a policy denies the access to an element, it must also deny the access to its subordinate elements and attributes. In other

[3]We use a variable $userid to refer to the identity of the requesting user in the access control policies.

words, whenever access to a node is allowed, access to all the ancestor elements must be allowed as well. We impose this requirement since we believe that elements or attributes isolated from their ancestor elements are meaningless.

For example, if an element or attribute specifies a relative URI, its interpretation depends on the attribute `xml:base` [22] specified at the ancestor elements. Another advantage of the denial downward consistency is that it makes implementation of runtime policy evaluation easier.

To satisfy the above requirements, the semantics of our access control policy is defined 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.

Note that the above algorithm does not always force a policy to satisfy the denial downward consistency. For example, if a rule with `+R` is specified on a certain node and a rule with `-R` is specified on an ancestor element of the explicitly access-granted node, the denial rule revokes the grant permission intended by the policy writer. Policy authoring tools (or analysis tools) can assist policy writers to detect such cases.

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.

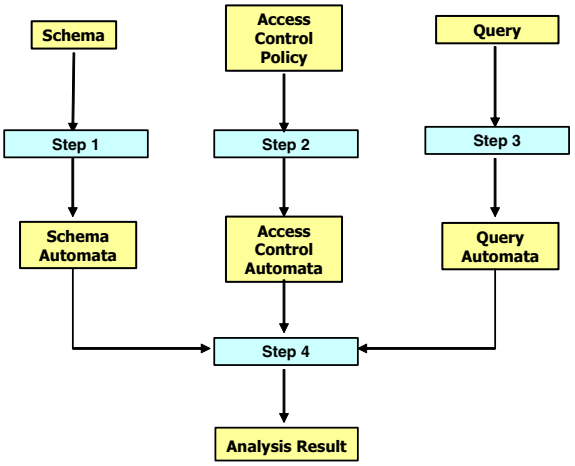


Figure 3: Framework of the analysis

3.3 Run-time Access Control

For the integration of access control and query processing, we assume that if there exist access-denied nodes in a target XML document, the query processor behaves as if they do not exist in the document. We believe that the node-level access control will greatly benefit by returning only authorized nodes without raising an error [4].

We explain how the semantics described above are enforced by the access control system at run-time. A 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 access control rules applicable to the requested node(s). Then, the controller computes the access decision(s) according to the rules and returns *grant* or *denial* for each node. Obviously, a naive implementation of this scenario can lead to poor performance by repeating evaluations of the rules per node.

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 query expressions.

Figure 3 depicts an overview of our static analysis. Static analysis has four steps as below:
Step 1: creating schema automata from schemas
Step 2: creating access control automata from access control policies
Step 3: creating query automata from XQuery queries
Step 4: comparison of schema automata, query automata, and access control automata

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

4.1 Automata and XPath expressions

In preparation, we introduce automata and show how we capture XPath expressions by automata.

[4]Another semantic model is to raise an access violation whenever the query processor encounters the "access denied" node.

A *non-deterministic 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 [18]. 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 "$\cdot\,\Omega^*\cdot$", 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 *over-estimation automaton* $\overline{M}[r]$ and an *under-estimation automaton* $\underline{M}[r]$. To construct $\overline{M}[r]$, we assume that predicates are always satisfied. That is, we first remove all predicates from r and then create an automaton $\overline{M}[r]$. Obviously, $\overline{M}[r]$ accepts all paths matching r and may accept other paths (over-estimation). For example, if r is `/record[...]`, then $L(\overline{M}[r]) = \{$`/record`$\}$.

Meanwhile, to construct $\underline{M}[r]$, we assume that predicates are never satisfied. That is, if a step in r contains one or more predicates, we first replace this step with an empty set, and then create an automaton $\underline{M}[r]$. Obviously, $\underline{M}[r]$ does not accept any paths if r contains predicates (under-estimation). For example, if r is `/record[...]`, then $L(\underline{M}[r])$ is an empty set.

As a special case, when r does not contain any predicates, $\overline{M}[r]$ is identical to $\underline{M}[r]$ and we simply write $M[r]$ for denoting both.

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 non-terminals (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

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

where $\mathbf{e}$ is an element name in Σ^E and $\mathtt{a}$ 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`$\}$,
$N = \{\text{Record}, \text{Diag}, \text{Chem}, \text{Com}, \text{Patho}, \text{Presc}\}$,
$\delta(q^{\text{ini}},$ `record`$) = \{\text{Record}\}$,
$\delta(\text{Record},$ `diagnosis`$) = \{\text{Diag}\}$,
$\delta(\text{Record},$ `chemotherapy`$) = \{\text{Chem}\}$,
$\delta(\text{Record},$ `comment`$) = \{\text{Com}\}$,
$\delta(\text{Record},$ `record`$) = \{\text{Record}\}$,
$\delta(\text{Diag},$ `pathology`$) = \{\text{Patho}\}$,
$\delta(\text{Diag},$ `comment`$) = \{\text{Com}\}$,
$\delta(\text{Chem},$ `prescription`$) = \{\text{Presc}\}$,
$\delta(\text{Chem},$ `comment`$) = \{\text{Com}\}$,
$\delta(\text{Patho},$ `@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 *under-estimation access control automaton* and an *over-estimation access control automata*. This pair captures the set of those paths to elements or attributes which 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 $r_1, ..., r_m$ be the XPath expressions occurring in the grant rules (`+r`), and let $r'_1, ..., r'_n$ be the XPath expressions occurring in the denial rules (`-r`).

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

$$\begin{aligned}
L(M^\Gamma) &= (L(M[r_1]) \cup \cdots \cup L(M[r_m])) \\
&\qquad \setminus \quad (L(M[r'_1]) \cup \cdots \cup L(M[r'_n]))
\end{aligned}$$

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

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,

$$\begin{aligned}
L(M^\Gamma) &= \{\mathtt{record}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}) \\
&\qquad \setminus \quad (\Sigma^E)^* \cdot \{\mathtt{comment}\} \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\})
\end{aligned}$$

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

$$\begin{aligned} L(\underline{M^\Gamma}) &= (L(\underline{M}[r_1]) \cup \cdots \cup L(\underline{M}[r_m])) \\ &\quad \setminus \ (L(\overline{M}[r'_1]) \cup \cdots \cup L(\overline{M}[r'_n])), \\ L(\overline{M^\Gamma}) &= (L(\overline{M}[r_1]) \cup \cdots \cup L(\overline{M}[r_m])) \\ &\quad \setminus \ (L(\underline{M}[r'_1]) \cup \cdots \cup L(\underline{M}[r'_n])). \end{aligned}$$

Again, we can construct $\underline{M^\Gamma}$ and $\overline{M^\Gamma}$ by applying boolean operations to 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,

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

4.4 Step 3: Creating query automata

Given a FLWR 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, and WHERE) clauses. XPath expressions in FOR-LET-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.

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-WHERE

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

RETURN

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

Next, we create a query automaton M^r for each r of the extracted XPath expressions. If r occurs in a FOR-LET-WHERE clause, then M^r is defined as $\overline{M}[r]$. Observe that we over-estimate r, since we would like to err on the safe side in our static analysis. When r occurs in a RETURN clause, M^r is defined as an automaton that accepts a path if and only if some of its sub-paths matches r. Formally,

$$L(M^r) = L(\overline{M}[r]) \cdot (\Sigma^E)^* \cdot (\Sigma^A \cup \{\epsilon\}).$$

This automaton can easily be constructed from $\overline{M}[r]$.

As an example, let r be `/record//comment`, which is the last XPath expression occurring in the RETURN clause. Then, M^r accepts `/record/comment`, `/record/comment/@type`, `/record/comment/record`, `/record/comment/diagnosis`, and so forth.

4.5 Step 4: Comparison of automata

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

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

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

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

$$L(M^r) \subseteq L(M^\Gamma).$$

The path expression r is *always-denied* if no path is accepted by all of the schema automaton, query automaton, and access control automaton; that is,

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

When schemas are unavailable, we examine if

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

The path expression r is *statically indeterminate* if it is neither always-granted or 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 under-estimation $\underline{M^\Gamma}$ when we want to determine whether or not a query is always-granted. That is, we examine if

$$L(M^r) \cap L(M^G) \subseteq L(\underline{M^\Gamma}).$$

When schemas are unavailable, we examine if

$$L(M^r) \subseteq L(\underline{M^\Gamma})$$

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

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

	role name	rule semantics
1	*M* (*Maintainer*)	Access to all information is granted.
2	*MM* (*Member Mgmt.*)	Access to all member info. is granted, but access to item info. is not.
3	*IM* (*Item Mgmt.*)	Access to all item info. is granted.
4	*S* (*Seller*)	A seller cannot see privacy info., and personal info. (credit card info. and profiles). A seller can see who bought his item. Otherwise, access to anonymous bidder info. and buyer info. is denied.
5	*B* (*Buyer*)	A buyer cannot see privacy info., and personal info. A buyer can see his own bids and purchases. Otherwise, access to anonymous bidder info. and buyer info is denied.
6	*V* (*Visitor*)	A visitor cannot see privacy info., and personal info. A visitor cannot see who sells, bids and buy an item.
7	*UB* (*Buyer: US only*)	The same access permission as Buyer except that access to foreign items is denied.
8	*US* (*Seller: US only*)	The same access permission as Seller except that access to foreign items is denied.
9	*UV* (*Visitor: US only*)	The same access permission as Visitor except that access to foreign items is denied.

Table 1: The sample access control policy

When schemas are unavailable, we examine if

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

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 to perform run-time checking of the access control policy for r, and may trigger further optimization 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. EXPERIMENTS

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

```
Role:   Maintainer
+R, /

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

Role:   Visitor
+R, /
-R, //person
-R, //bidder/personref
-R, //seller
-R, //buyer
-R, //privacy
```

Figure 4: The definition of sample policy

5.1 Effectiveness of Static Analysis

We wish to show the percentage of queries that are made more efficient, for average, real-world cases. For each query, we also benchmark the performance increase. 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 runtime access checks.

Settings. We use the sample queries and the DTD developed by the XMark project[5], which is a well-known benchmark framework for XQuery based on an auction scenario. An auction document consists of a list of auction items, participants information., etc. The benchmark has 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>
```

There are 77 element types defined by the DTD. We wrote a sample access control policy in which 9 roles are defined. Each role is associated with 1 through 15 access control rules. Their semantics are summarized in Table 1. We list a part of the policy definitions in Figure 4. 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 buyer can read the contents of his own //person/creditcard and //person/profile, but not the credit cards and profiles of other users.

Note that the sample policy is a value-based policy, i.e., XPath predicates appear in the rules. As described in Sec-

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

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
US	G	-	-	-	-	-	-	-	-	-	-	-	D	-	-	-	G	-	-	-
UB	G	-	-	-	-	-	-	-	-	-	-	-	D	-	-	-	G	-	-	-
UV	D	-	-	-	-	-	-	-	-	D	-	-	D	-	-	-	D	-	-	D

(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	-	-	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
US	G	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
UB	G	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
UV	D	-	-	-	-	-	-	-	-	D	-	-	-	-	-	D	-	-	D	-

(b) Without the DTD

Table 2: Results of Static Analysis of XMark Queries

tion 4, we over- and under-estimate the access control automata in order to perform the static analysis.

Queries Made Efficient. For each query/role pair, we check whether or not our static analysis removes the runtime access check. We perform the experiment for two cases: one case with the DTD and the other case without the DTD. We statically analyze all the XPath expressions for each query. Recall that if an XPath expression in the query is always-denied, we rewrite the query.

Tables 2(a) and 2(b) show the results of our static analysis with and without the DTD, respectively. Each entry in the table indicates the result by either "G", "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, while all other expressions are always-granted.
- "−" indicates that at least one XPath expression in the query is statically indeterminable.

A query marked by "G" contains no XPath expressions requiring the runtime access check. If a query is marked by "D", it contains XPath expressions that always fail their runtime access checks. However, in this case, we rewrite such expressions as null lists in advance. As a result of this rewrite, the runtime access check becomes unnecessary. Finally, if queries are marked by "−", the result of the runtime check is not predictable, and must be performed.

For example, we can read from Table 2(a) that the mark of Query #4 for role *IM* is "D". This means that when a user filling a role *IM* makes Query #4, we first rewrite the query so that it can be evaluated without the runtime access check.

Tables 2(a) and 2(b) show that 65% and 40% of the query/role pairs, respectively (i.e., "G" + "D"), do not require the runtime access check. Furthermore, for 25% and 10% of the query/role pairs (i.e., "D"), we can optimize queries by rewriting.

From Table 2(b), we conclude that even when no DTDs are available, our static analysis can result in significant optimization of the query. From from Table 2(a), we conclude that the analysis can be further refined by exploiting DTD information. Note that the sample policy contains XPaths with predicates, which cause over- and under-estimation of the access control automata. Even in such a case, our static analysis frequently makes runtime access checks unnecessary.

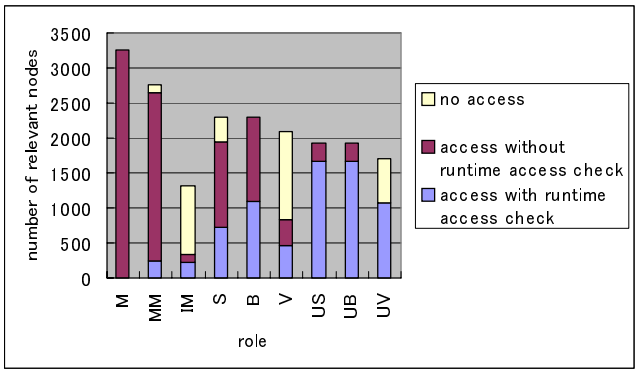


Figure 5: Nodes Exempted from Access or Runtime Access Check

Nodes Exempted from Access or Runtime Access Check. Here we consider how much 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 runtime access check by our static analysis and query optimization.

As an example document, we use an auction document of the XMark project. This document has 250,000 nodes, and is generated by specifying factor =0.05 (see the XMark project page).

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

no access These nodes are exempt from access. In other words, the original query accesses these nodes but the rewritten query does not.

access without runtime access check These nodes are exempt from the runtime access control check, but they must still be read.

access with runtime access check These nodes are not exempt from runtime access check or access. In other words, the rewritten query accesses these nodes and this access requires runtime access check.

```
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
```

Figure 6: Random policy example

The bar chart (Figure 5) shows, for each role, the number of nodes in the three categories. It shows the average 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 runtime access checks at all, because a maintainer has an 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.

5.2 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, 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, where we statically analyze XPath expressions in each query, i.e., we determine whether they are always-denied or always-granted. When there are many queries, we cache M^G and M^Γ which are computed in the initialization phase, and later in the analysis phase we repeatedly use them.

Settings. Three large DTDs are used, that is, the xmlspec-v21.dtd from W3C XML Working Group [5], which has 157 element types, the HL7.dtd from Health Level Seven[6], which has 621 element types (as far as we know, this is the biggest DTD publicly available), and the docbookx.dtd by OASIS DocBook technical committee[7], which has 393 element types.

We use access control policies with different sizes, i.e., 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.

[6] `http://www.hl7.org`

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

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. The JIT-compiler is fully warmed up before each run.

Figure 7(a) shows the running-time of the initialization phase. Each point indicates the time required to compute an access control automaton for each role in the randomly generated policies. The running-time does not include the time required to compute schema automata M^G, which is 63.7ms, 57.5ms, and 249.3ms for xmlspec-v21.dtd, HL7.dtd, and docbookx.dtd, respectively. The sizes of states of M^G are 214, 623, and 501 for xmlspec-v21.dtd, HL7.dtd, and docbookx.dtd, respectively. The size of the access control automaton M^Γ is about 1,200 in the worst case.

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

In both phases, the performance is much better in the case of HL7.dtd than in the case of xmlspec-v21.dtd or docbookx.dtd. This is because xmlspec-v21.dtd and docbookx.dtd contain many recursive definitions and thus are more complicated than HL7.dtd.

The initialization phase (computing M^Γ) takes more than 10 seconds for large policies and the running-time increases non-linearly. On the other hand, in the analysis phase, the running-time 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 schemas and access control policies.

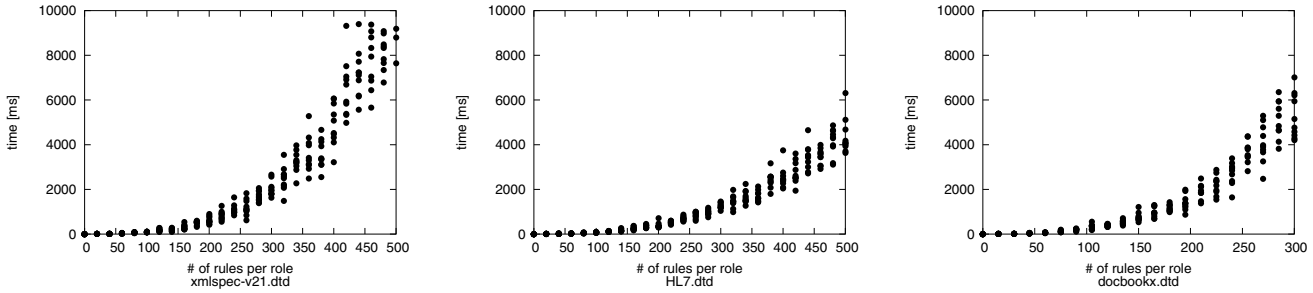
6. CONCLUSIONS AND FUTURE WORK

In this paper, we have attempted to ease the burden of checking access control policies for XML documents by distributing the burden to 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 5) reveals that (1) static analysis frequently makes run-time checks unnecessary and further provides significant optimizations, and (2) our prototype scales nicely when schemas, access control policies, and queries are large.

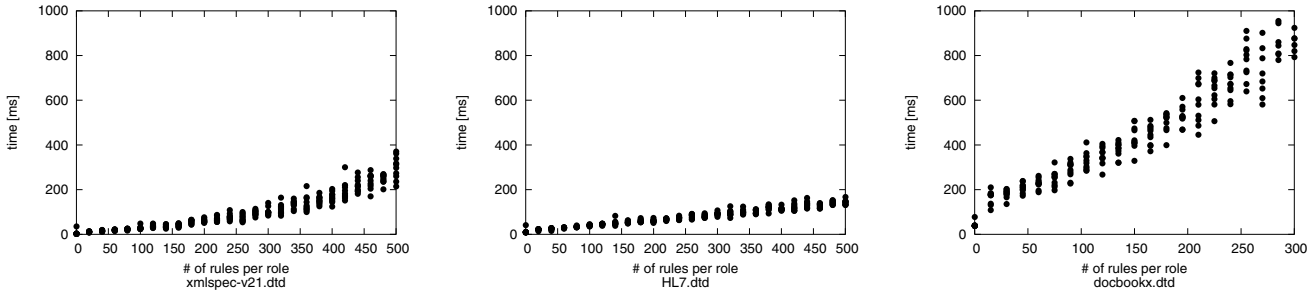
However, our static analysis has some limitations. We summarize these limitations and sketch future extensions for overcoming them.

Value-based access control: Value-based access control requires that the XPath expressions in the access control policies contain predicates. Query expressions may also use XPath predicates. We have approximated such access control policies and query expressions by creating under-estimation automata and over-estimation automata. These approximations make some queries statically indeterminate.

However, when an access control policy and query expression specify the same value, we can capture predicates by incorporating them into the underlying alphabet (e.g., by handling `record[@patientId = $userid]` as a "symbol"). Such automata help to statically perform value-based access control. For example, if a query expression and an access control policy use `record[@patientId = $userid]`, we can statically grant this access.



(a) Initializing M^{Γ}



(b) Analysis of each XPath expression

Figure 7: Results of Scalability Tests

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 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 [26].

Advanced features of XQuery: We have significantly simplified XQuery here, but XPath allows arbitrary nesting of FLWR expressions and even allows recursive queries. We cannot handle recursive queries and are forced to rely on run-time checking. However, we can handle nested FLWR expressions by extracting XPath expressions.

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.

7. ADDITIONAL AUTHORS

Additional author: Satoshi Hada (IBM Tokyo Research Lab, 1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken 242-8502, Japan, email: satoshih@jp.ibm.com)

8. REFERENCES

[1] E. Bertino. Data hiding and security in object-oriented databases. In *ICDE 92*, 1992.

[2] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Controlled access and dissemination of XML documents. In *WIDM'99*. ACM, Nov. 1999.

[3] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Author-X: a Java-based system for XML data protection. In *IFIP WG 11.3 Working Conference on Database Security*, 2000.

[4] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, and J. Simeon. XQuery 1.0: An XML query language. W3C working draft 16 august 2002. http://www.w3.org/TR/xquery/, August 2002.

[5] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible Markup Language (XML) 1.0. W3C Recommendation. http://www.w3.org/TR/REC-xml, February 1998.

[6] J. Clark and S. DeRose. XML Path Language (XPath) version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath, Nov 1999.

[7] J. Clark and M. Murata (Eds). "RELAX NG Specification". OASIS Committee Specification, Dec. 2001.

[8] J. Clark (Eds). "XML Transformations (XSLT) Version 1.0". W3C Recommendation, Nov. 1999. http://www.w3.org/TR/xslt.

[9] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available at http://www.grappa.univ-lille3.fr/tata, 1997.

[10] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *EDBT 2000*, LNCS 1777, Mar. 2000.

[11] A. Deutsch and V. Tannen. Containment of regular path expressions under integrity constraints. In *KRDB*, 2001.

[12] D. Draper, P. Fankhauser, M. Fernandez, A. Malhotra, K. Rose, M. Rys, J. Simeon, and P. Wadler. XQuery 1.0 and XPath 2.0 formal semantics. W3C working draft 16 august 2002, August 2002.

[13] W. Fan and L. Libkin. On XML integrity constraints in the presence of DTDs. *J. ACM*, 49(3), 2002.

[14] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, 1998.

[15] A. Gabillon and E. Bruno. Regulating access to XML documents. In *IFIP WG 11.3 Working Conference on*

Database Security, Jul. 2001.

[16] S. Godik and T. Moses (Eds). "eXtensible Access Control Markup Language (XACML) Version 1.0". OASIS Standard, Feb. 2003.

[17] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. *Theory of Machines and Computations*, 1971.

[18] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.

[19] H. Hosoya and M. Murata. Validation and boolean operations for attribute-element constraints. In *Programming Languages Technologies for XML (PLAN-X)*, October 2002.

[20] H. Hosoya and B. C. Pierce. "XDuce: A Typed XML Processing Language". In *WebDB*, 2000.

[21] M. Kudo and S. Hada. XML document security based on provisional authorization. In *CCS-7*. ACM, Nov 2000.

[22] J. Marsh (Eds). XML Base. W3C Recommendation, June 2001. http://www.w3.org/TR/2001/REC-xmlbase-20010627/.

[23] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In *PODS*, 2002.

[24] M. Murata, D. Lee, and M. Mani. "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extreme Markup Languages*, Aug. 2001. `http://www.idealliance.org/papers/extreme02/titles.html`.

[25] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.

[26] D. Olteanu, H. Meuss, T. Furche, and F. Bry. XPath: Looking forward. In *Proceedings of Workshop on XML Data Management (XMLDM)*, LNCS. Springer, 2002.

[27] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD 1999*, 1999.

[28] F. Rabitti, E. Bertino, W. Kim, and D. Woelk. A model of authorization for next-generation database systems. *TODS*, 16(1), 1991.

[29] W3C. "XML Schema". W3C Recommendation, May 2001.

[30] P. T. Wood. Containment for XPath fragments under DTD constraints. In *ICDT*, 2003.

APPENDIX

A. IMPLEMENTATION

Here we present a few techniques for improving the performance of our implementation.

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

From our experience, the performance of our static analysis largely depends on the minimization operation. To improve the performance of this operation, we use a very efficient algorithm by Hopcroft [17]. Furthermore, we avoid minimization when we can make automata small enough by removing redundant states (i.e., unreachable states and deadend states). We use this technique when we compute intersection ($\cap$) and difference ($\setminus$) automata.

On the other hand, we heavily use the determinization operation. While computing $\overline{M^\Gamma}$ and $\underline{M^\Gamma}$ in Step 2 (Section 4.3), we always determinize intermediate automata as well as $\overline{M^\Gamma}$ and $\underline{M^\Gamma}$. By doing so, we can efficiently test $\subseteq$ by applying $\setminus$ in Step 4 (Section 4.5). If $\overline{M^\Gamma}$ and $\underline{M^\Gamma}$ were large non-deterministic automata, $\setminus$ (which requires the determinization) would be prohibitively expensive.

In computing automata in Step 4 (Section 4.5), we first use M^r, which is typically compact, and then use M^G, which can be very large and complex. More precisely, in checking $L(\overline{M^\Gamma}) \cap L(M^G) \cap L(M^r) = \emptyset$, we first compute $L(\overline{M^\Gamma}) \cap L(M^r)$ and then compute its intersection with M^G; in checking $L(M^r) \cap L(M^G) \subseteq L(\underline{M^\Gamma})$, we first compute $L(M^r) \setminus L(\underline{M^\Gamma})$ and then check whether or not $(L(M^r) \setminus L(\underline{M^\Gamma})) \cap L(M^G)$ is empty. In our experience, $L(\overline{M^\Gamma}) \cap L(M^r)$ and $L(M^r) \setminus L(\underline{M^\Gamma})$ are reasonably compact, but $L(\overline{M^\Gamma}) \cap L(M^G)$ and $L(M^G) \setminus L(\underline{M^\Gamma})$ can be very complex.