社団法人　情報処理学会　研究報告
IPSJ SIG Technical Report

2003－ＣＳＥＣ－22　(12)
2003／7／17

静的解析を用いたＸＭＬアクセス制御

村田　真†　戸沢　晶彦†　工藤　道治†　羽田　知史†

† 日本アイビーエム(株) 東京基礎研究所　〒242-8502 大和市下鶴間 1623-14

あらまし　XMLデータベースに対するアクセス制御ポリシーでは、アクセスの対象を指定するためにXPathなどのパス表現を用いることが多い。しかし、そのようなアクセス制御ポリシーは、XQueryのエンジンにとっては負担となる。この負担を軽減するために、我々はXMLアクセス制御のための静的解析を提案する。アクセス制御ポリシー、検索式、および、スキーマが与えられると、静的解析は、この検索式がスキーマでは許可されるがアクセス制御ポリシーによっては隠蔽される要素や属性にアクセスしないと保証されているかどうか決定する。静的解析は、実際のデータベースに対していかなる検索式をも評価することなく実行できるので、実行時の検査は、静的解析は決定できなかったときにだけ必要となる。我々は、XQueryのための静的解析のプロトタイプを実装し、実験によってその有効性を示した。

キーワード　XMLデータベース、XQuery、XPath、アクセス制御、オートマトン

XML Access Control Using Static Analysis

Makoto MURATA†, Akihiko TOZAWA†, Michiharu KUDO†, and Satoshi HADA†

† IBM Japan, Co.,Ltd. Tokyo Research Laboratory　1623–14, Shimotsuruma, Yamato, Kanagawa, 242–8502 Japan

Abstract　Access control policies for XML database typically use regular path expressions such as XPath for specifying the objects to be accessed. However such access control policies are burdens to XQuery engines. 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. We have built a prototype of static analysis for XQuery, and shown the effectiveness through experiments.

Key words　XML database, XQuery, XPath, access control, and automaton

1. Introduction

XML has become an active area in database research. XPath and XQuery 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 access control for XML database systems. It should ideally provide expressiveness as well as efficiency. That is, (1) it should be easy to write element- and attribute-level access control policies, and (2) it should be possible to efficiently determine whether or not an access to an element or an attribute is granted by such fine-grained access control policies.

Some early experiences [1], [5], [8] with access control for XML documents have been reported already. Their policy languages achieve expressiveness by using XPath 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, we only have to specify a simple XPath expression `//article//name` as part of an access control policy. XPath-based access control policies are additional burdens for XML query engines, however. 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 will easily lead to unacceptable performance.

In this paper, we introduce static analysis as a new ap-

proach for XML access control. Static analysis examines access control policies and query expressions as well as schemas, if present. Unlike 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 evaluated). 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.

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. In the second phase, we compare these automata to make a static access decision. We have three possible decisions: (1) accesses by queries are *always-granted*; (2) they are *always-denied*; and (3) they are *statically indeterminate*. Run-time access control is no longer required when the decision is either always-granted or always-denied, but it is still needed when the decision is statically indeterminate.

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 strings. These elements collectively form a tree. The content of each element is a sequence of elements or strings. 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, and RELAX NG 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 [4] as a formal model of schemas. Murata et al. [9] have shown that tree regular grammars can model DTD, W3C XML Schema, and RELAX NG.

```
<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>
```

図 1 An XML document example

A *schema* is a 5-tuple $G = (N, \Sigma^E, \Sigma^A, S, P)$, where N is a finite set of *non-terminals*, Σ^E is a finite set of *element names*, Σ^A is a finite set of *attribute names*, S is a subset of $\Sigma^E \times N$, and P is a set of production rules $X \to r\ \mathcal{A}$ ($X \in N$, 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. For the sake of simplicity, we do not handle text as values of elements or attributes in this paper.

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

N_1 = {Record, Diag, Chem, Com, Patho, Presc},
Σ_1^E = {record, diagnosis, chemotherapy, comment, pathology, prescription},
Σ_1^A = {@type},
S_1 = {record[Record]},
P_1 = {Record → (diagnosis[Diag]*, chemotherapy[Chem]*, comment[Com]*, record[Record]*) ∅,
Diag → (pathology[Patho], comment[Com]*) ∅,
Chem → (prescription[Presc]*, comment[Com]*) ∅,
Com → ϵ ∅, Patho → ϵ {@type}, Presc → ϵ ∅}.

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)>
```

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. Most schemas (e.g., XHTML and DocBook) for narrative documents are recursive. Our static analysis can 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 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 is a sample query that 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. We use the term *node-level* access control to represent both the *element-level* and the *attribute-level* access control.

3.1 Syntax of Access Control Policy

The node-level access control policy for XML documents applies similar syntax used in the conventional access control models (e.g. [3], [6], [10]). 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 computer 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:

```
(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.

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`(注1)`]/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 grant or denial decision in response to any access request.

In this paper, we consider another requirement called *denial downward consistency*, which is a 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 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, in some application, to process an element or an attribute may need to have access to the xml:base attribute [2] specified in an ancestor element. 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 based on the following three principles:

(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 principles. First, the algorithm gathers every grant rule with `+r` and marks "+" on the target nodes referred to by the XPath expression. 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 it is already marked. 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` element and subordinate text nodes with - and it 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.

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 four rules: `(`*Sbj*`,+r,/a)`, `(`*Sbj*`,+r,/a//*)`, `(`*Sbj*`,+r,/a//@*)`, and `(`*Sbj*`,+r,/a//text())`. Thus, `+R` and `-R` are syntax sugar of our underlying policy model for our static analysis. On the other hand, such syntax sugar will benefit GUI-based policy authoring tool because it represents policy writers' intention in more succinct way. Thus we use `+R` and `-R` in the following sections to make the policy specification more succinct.

(注1): We use a variable $userid to refer to the identity of the requesting user in the access control policies.

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

図 2 The XML document that *Intern* can see

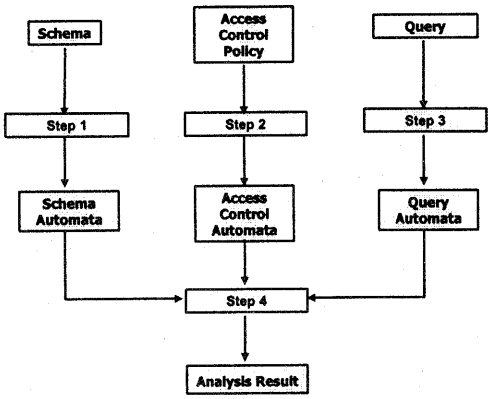

図 3 Framework of the analysis

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(注2).

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* per each node. Obviously, a naive implementation of this scenario leads 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.

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 [7]. 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 makes it easy to capture XPath expressions with automata. As long as an XPath expression contains no predicate, it is easy to construct an automaton from it. 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). 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). 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 Static Analysis Algorithm

a) Step 1: Creating schema automata

Since we are interested in permissible paths rather than

(注2): Another semantic model is to raise an access violation whenever the query processor encounters the "access denied" node.

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

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

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

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

$$\delta(x, \mathbf{a}) = \{q^{\text{fin}} \mid \mathbf{a} \in \mathcal{A} \text{ for some } x \to r\,\mathcal{A} \text{ in } P\},$$

where $\mathbf{e}$ is an element name in Σ^E and $\mathbf{a}$ is an attribute name in Σ^A.

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

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

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`, and
`/record/chemotherapy/comment`

Recall that our last schema in Section 2. is recursive. The schema automaton for that schema allows an infinite number of paths, since `record` can be repeated freely for each of the above paths.

b) 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.

Let $r_1, ..., r_l$ be the XPath expressions occurring in the grant rules with propagation (`+R`), $r_{l+1}, ..., r_m$ be the XPath expressions for the grant rules without propagation (`+r`), and let $r_{m+1}, ..., r_n$ be the XPath expressions occurring in the denial rules (`-R`). We first assume that none of $r_1, ..., r_n$ contain predicates. Then, the under-estimation access control automaton and over-estimation access control automaton can exactly capture the set of exposed paths and these automata, denoted M^Γ, are identical.

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 $r_{m+1}, ..., r_n$. Formally,

$$L(M^\Gamma) = (L(M[r_1]) \cdot \Sigma^* \cup \cdots \cup L(M[r_l]) \cdot \Sigma^* \cup L(M[r_{l+1}]) \cup \cdots \cup L(M[r_m])) \setminus (L(M[r_{m+1}]) \cdot \Sigma^* \cup \cdots \cup L(M[r_n]) \cdot \Sigma^*)$$

where $\Sigma = \Sigma^E \cup \Sigma^A$ and "$\cdot$" denotes the concatenation of two regular sets. We can construct M^Γ by applying boolean operations to $M[r_1], ..., M[r_n]$. Note that propagation of grant and denal rules can be handled by appending Σ^* which accepts any suffix to each path.

Now, let us consider the case that predicates occur in $r_1, ..., r_n$. 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 $L(M[r_{m+1}]), ..., L(M[r_n])$ 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,

$$L(\underline{M^\Gamma}) = (L(\underline{M}[r_1]) \cdot \Sigma^* \cup \cdots \cup L(\underline{M}[r_l]) \cdot \Sigma^* \cup L(\underline{M}[r_{l+1}]) \cup \cdots \cup L(\underline{M}[r_m])) \setminus (L(\overline{M}[r_{m+1}]) \cdot \Sigma^* \cup \cdots \cup L(\overline{M}[r_n]) \cdot \Sigma^*)$$

and

$$L(\overline{M^\Gamma}) = (L(\overline{M}[r_1])\cdot\Sigma^* \cup \cdots \cup L(\overline{M}[r_l])\cdot\Sigma^* \cup L(\overline{M}[r_{l+1}]) \cup \cdots \cup L(\overline{M}[r_m])) \setminus (L(\underline{M}[r_{m+1}])\cdot\Sigma^* \cup \cdots \cup L(\underline{M}[r_n])\cdot\Sigma^*).$$

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.

c) 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.

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, $L(M^r) = L(\overline{M}[r])\cdot\Sigma^*$. 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 paths such as `/record/comment/record` and `/record/comment/record/comment/@type` in addition to `/record/comment`.

d) 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,

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

When schemas are unavailable, we assume that M^G allows all paths and examine if $L(M^r) \subseteqq L(M^\Gamma)$.

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.

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: We examine $L(M^r) \cap L(M^G) \subseteqq L(\underline{M^\Gamma})$ (when schemas are available) or $L(M^r) \subseteqq L(\underline{M^\Gamma})$ (when schemas are unavailable). Likewise, we use an over-estimation $\overline{M^\Gamma}$ when we determine whether or not a path expression is always-denied: We examine $L(M^r) \cap L(M^G) \cap L(\overline{M^\Gamma}) = \emptyset$ (when schemas are available) or $L(M^r) \cap L(\overline{M^\Gamma}) = \emptyset$ (when schemas are unavailable).

4.3 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. Experiment

We have implemented our static analysis algorithm in Java. In this section, we present an experiment based on this implementation to show how much the cost of query

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

表 1 Sample Analysis

evaluation can be reduced by our static analysis and query optimization.

We use the sample queries and the DTD developed by the XMark project, which is a well-known benchmark framework for XQuery (注3). The number of sample queries is 20 and the DTD defines 77 elements. We use a sample access control policy in which 9 roles are defined. Each role is associated with 1 through 15 access control rules. The sample policy is a value-based policy, i.e., XPath predicates appear in the rules. We omit the detail due to space limitations.

For each pair of query and role, we check whether or not our static analysis removes the run-time access check. Also, we repeat the experiment for two cases: one case with the DTD and the other case without the DTD.

Tables 1(a) and 1(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 and thus no run-time access check is required.
- "D" indicates that at least one of the XPath expressions in the query is always-denied, but no expression is statically indeterminable. In this case, we rewrite the query. Again, no run-time access check is required.
- "–" indicates that at least one XPath expression in the query is statically indeterminable and thus a run-time access check is still required. If the query contains an XPath expression that is always-denied, we rewrite it.

For example, Table 1(a) shows that the result for Query #4 and role *IM* is "D", which means that when a user filling a role *IM* makes Query #4, it can be rewritten statically and no run-time access check is required.

Tables 1(a) and 1(b) show that 65% and 40% of the query/role pairs, respectively, do not require run-time access checks. Furthermore, for 25% and 10% of the query/role pairs, we can optimize queries by rewriting meaningless XPath expressions as null lists. Table 1(b) shows that even when no DTD is available our static analysis works very well, and the analysis is further refined by exploiting DTD information. We conclude that our static analysis can frequently make run-time access checks unnecessary.

6. Conclusion

In this paper, we have proposed static analysis to ease the burden of checking access control policies for XML documents We have built a prototype of our static analysis and demonstrated its effectiveness. However, our static analysis has some limitations. In particular, we dealed with only a subset of XPath. We need to extend our static analysis technique so that it supports more features of XPath expressions.

文 献

[1] 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.

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

[3] S. Castano, M. Fugini, G. Martella, and P. Samarati. *Database Security*. Addison-Wesley, 1994.

[4] 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.

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

[6] D. E. Denning. *Cryptography and Data Security*. Addison-Wesley, 1983.

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

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

[9] M. Murata, D. Lee, and M. Mani. "Taxonomy of XML Schema Languages using Formal Language Theory". In *Extreme Markup Languages*, Aug. 2001. `http://www.cs.ucla.edu/~dongwon/paper/`.

[10] T. Y. C. Woo and S. S. Lam. A framework for distributed authorization. In *CCS-1*, pages 143–158. ACM, Nov. 1993.

(注3): The project home page can be found at `http://monetdb.cwi.nl/xml/`