Access Control Models for XML

Abdessamad Imine

Lorraine University & INRIA-LORIA Grand-Est

Nancy, France

Abdessamad.Imine@loria.fr
Outline

• Overview on XML
• Why XML Security?
• Querying Views-based XML Data
• Updating Views-based XML Data
Outline

- Overview on XML
- Why XML Security?
- Querying Views-based XML Data
- Updating Views-based XML Data
What is XML?

- eXtensible Markup Language [W3C 1998]

```xml
<files>
  <record>
    <name>Robert</name>
    <diagnosis>Pneumonia</diagnosis>
  </record>
  <record>
    <name>Franck</name>
    <diagnosis>Ulcer</diagnosis>
  </record>
</files>
```
What is XML?

• eXtensible Markup Language [W3C 1998]

```
<files>
  <record>
    <name>Robert</name>
    <diagnosis>
      Pneumonia
    </diagnosis>
  </record>
  ...
</files>
```
XML for Documents

- SGML
- HTML - hypertext markup language
- TEI - Text markup, language technology
- DocBook - documents -> html, pdf, ...
- SMIL - Multimedia
- SVG - Vector graphics
- MathML - Mathematical formulas
XML for Semi-Structured Data

- MusicXML
- NewsML
- iTunes
- DBLP [http://dblp.uni-trier.de](http://dblp.uni-trier.de)
- CIA World Factbook
- XBEL - bookmark files (in your browser)
- KML - geographical annotation (Google Maps)
- XACML - XML Access Control Markup Language
XML as Description Language

- Java servlet config (web.xml)
- Apache Tomcat, Google App Engine, ...
- Web Services - WSDL, SOAP, XML-RPC
- XUL - XML User Interface Language (Mozilla/Firefox)
- BPEL - Business process execution language
- Other Web standards:
  - XSLT, XML Schema, XQueryX
  - RDF/XML
  - OWL - Web Ontology Language
  - MMI - Multimodal interaction (phone + car + PC)
XML Tools

- Standalone:
  - xsltproc, mxquery, calabash (XProc)
- Most Programming Languages have XML parsers
  - SAX (streaming), DOM (in-memory) interfaces
  - libxml2, expat, libxslt (C)
  - Xerces, Xalan (Java)
- XPath (path expressions) used in many languages
  - JavaScript/JQuery
  - XSLT, XQuery
Native XML Databases

- Offer native support for XML data & query languages
  - Galax
  - MarkLogic
  - eXist
  - BaseX
  - among others...

- Suitable for new or lightweight applications
  - but some lack features like transactions, views, updates
XML in the Industry

- Most commercial RDBMSs now provide some XML support
  - Oracle 11g - XML DB
  - IBM DB2 pureXML
  - Microsoft SQL Server - XML support since 2005
- Language Integrated Query (LINQ) targets SQL & XML in .NET programs
- Data publishing, exchange, integration problems are very important
  - big 3 have products for all of these
- SQL/XML standard for defining XML views of relational data
XML Terminology

Tags and Text

- XML consists of tags and text
  
  ```xml
  <course cno = "Eng 055">
    <title>Spelling</title>
  </course>
  ```

- tags come in pairs: markups
  
  start tag: `<course>`
  
  end tag: `</course>`

- tags must be properly nested
  
  `<course> <title> ... </title> </course>` -- good
  
  `<course> <title> ... </course> <title>` -- ???
XML Terminology

Tags and Text

- XML consists of tags and text
  
  `<course  cno = “Eng 055”>
  `<title> Spelling </title>
  `</course>`

- tags come in pairs: markups
  
  start tag: `<course>`
  end tag: `</course>`

- tags must be properly nested
  
  `<course> <title> … </title> </course> -- good`
  `<course> <title> … </title> </course> </title> -- bad`
XML Terminology (cont.)

XML Elements

- Element: the segment between a start and its corresponding end tag

- Subelement: the relation between an element and its component elements.

```xml
<person>
  <name>Ali Baba</name>
  <tel>(33) 354595853</tel>
  <email>Ali.Baba@nights.com</email>
  <email>ababa@tales.org</email>
</person>
```
XML Attributes

A start tag may contain attributes describing certain “properties” of the element:

<picture>
  <height dim="cm">2400</height>
  <width dim="in">96</width>
  <data encoding="gif">M05+C$ …</data>
</picture>

References:

<person id="011" country="UK">
  <name>Stan Laurel</name>
</person>

<person country="USA" id="012">
  <name>Oliver Hardy</name>
</person>
XML Terminology (cont.)

Example: A relational database for school

Students:

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>sex</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Joe</td>
<td>male</td>
</tr>
<tr>
<td>002</td>
<td>Mary</td>
<td>female</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Course:

<table>
<thead>
<tr>
<th>cno</th>
<th>title</th>
<th>credit</th>
</tr>
</thead>
<tbody>
<tr>
<td>331</td>
<td>DB</td>
<td>3.0</td>
</tr>
<tr>
<td>350</td>
<td>Web</td>
<td>3.0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Enroll:

<table>
<thead>
<tr>
<th>id</th>
<th>cno</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>331</td>
</tr>
<tr>
<td>001</td>
<td>350</td>
</tr>
<tr>
<td>002</td>
<td>331</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
XML Terminology (cont.)

Example: A relational database for school

```xml
<school>
  <student id="001">
    <name> Joe </name>
    <sex> male </sex>
  </student>
  ...
  <course cno="331">
    <title> DB </title>
    <credit> 3.0 </credit>
  </course>
  ...
  <course>
  ...
  </course>
  <enroll>
    <id> 001 </id>
    <cno> 331 </cno>
  </enroll>
  ...
</school>
```
An XML document may come with an optional DTD - “schema”

```xml
<!DOCTYPE db [
  <!ELEMENT db (book*)>
  <!ELEMENT book (title, authors*, section*, ref*)>
  <!ATTLIST book isbn ID #required>
  <!ELEMENT section (text | section)*)>
  <!ELEMENT ref EMPTY>
  <!ATTLIST ref to IDREFS #implied>
  <!ELEMENT title #PCDATA>
  <!ELEMENT author #PCDATA>
  <!ELEMENT text #PCDATA>
]
```
Document Type Definition (DTD)

for each element type E, a declaration of the form:

```xml
<!ELEMENT   E   P>   E → P
```

where P is a regular expression, i.e.,

```plaintext
P ::= EMPTY | ANY | #PCDATA | E' | P1, P2 | P1 | P2 | P? | P+ | P*
```

- E’: element type
- P1, P2: concatenation
- P1 | P2: disjunction
- P?: optional
- P+: one or more occurrences
- P*: the Kleene closure
Document Type Definition (DTD)

✓ Extended context free grammar: `<!ELEMENT E P>`
  Why is it called extended?
  E.g., `book → title, authors*, section*, ref*`
✓ single root: `<!DOCTYPE db [ ... ]>`
✓ subelements are ordered.

The following two definitions are different. Why?

`<!ELEMENT section (text | section)>`
`<!ELEMENT section (text* | section* )>`
✓ recursive definition, e.g., section, binary tree:
  `<!ELEMENT node (leaf | (node, node))>`
  `<!ELEMENT leaf (#PCDATA)>`
Recursive DTDs

<!ELEMENT  person (name, father, mother)>
<!ELEMENT  father  (person)>
<!ELEMENT  mother (person)>

What is the problem with this? How to fix it?
Recursive DTDs

```xml
<!ELEMENT person (name, father, mother)>  
<!ELEMENT father (person)>  
<!ELEMENT mother (person)>  
```

What is the problem with this? How to fix it?
- optional (e.g., father?, mother?)
- Attributes

Ordering

How to declare element E to be an unordered pair (a, b)?
Recursive DTDs

```xml
<!ELEMENT  person (name, father, mother)>
<!ELEMENT  father  (person)>
<!ELEMENT  mother (person)>
```

What is the problem with this? How to fix it?
- optional (e.g., father?, mother?)
- Attributes

Ordering
How to declare E to be an unordered pair (a, b)?

```xml
<!ELEMENT  E  ((a, b) | (b, a)) >
```
Attribute Declaration

<!ATTLIST element_name
    attribute-name attribute-type default-declaration>

Example: “keys” and “foreign keys”

<!ATTLIST book
    isbn   ID   #required>

<!ATTLIST ref
    to    IDREFS  #implied>

Note: it is OK for several element types to define an attribute of the same name, e.g.,

<!ATTLIST person  name ID  #required>

<!ATTLIST pet    name ID  #required>
Document Type Definition (DTD)

Attribute Declaration

```xml
<!ATTLIST person
    id ID #required
    father IDREF #implied
    mother IDREF #implied
    children IDREFS #implied>
```

e.g.,

```xml
<person id="898" father="332" mother="336"
    children="982 984 986">
    ....
</person>
```
Valid XML Documents

A valid XML document must have a DTD.

✓ The document is well-formed
  - Tags have to nest properly
  - Attributes have to be unique

✓ It conforms to the DTD:
  - Elements conform to the grammars of their type definitions (nested only in the way described by the DTD)
  - Elements have all and only the attributes specified by the DTD
  - ID/IDREF attributes satisfy their constraints:
    - ID must be distinct
    - IDREF/IDREFS values must be existing ID values
XPath

W3C standard: [www.w3.org/TR/xpath](http://www.w3.org/TR/xpath)

- Navigating an XML tree and finding parts of the tree (node selection and value extraction)

  Given an XML tree T and a context node n, an XPath query Q returns
  - the set of nodes reachable via Q from the node n in T - if Q is a unary query
  - truth value indicating whether Q is true at n in T - if Q is a boolean query.

- Implementations: XALAN, SAXON, Berkeley DB XML, Monet XML - freeware, which you can play with

- A major element of XSLT, XQuery and XML Schema

- Version: XPath 3.0
XPath

XPath query $Q$:
- Tree traversal: downward, upward, sideways
- Relational/Boolean expressions: qualifiers (predicates)
- Functions: aggregation (e.g., count), string functions

```xml
/files/record/name[text()="Ali Baba"]
/files/record[name="Toto"]/diagnosis | /files/record[name="Pascal"]/diagnosis
```

```
<files>
  <record>
    <name>Robert</name>
    <diagnosis>
      Pneumonia
    </diagnosis>
  </record>
  <record>...
  </record>
</files>
```
XPath

Downward Traversal

Syntax:

$Q ::= l | @l | Q/Q | Q | Q | //Q | /Q | Q[q]$

$q ::= Q | Q \text{ op } c | q \text{ and } q | q \text{ or } q | \text{not}(q)$

- $l$: either a tag (label) or *: wildcard that matches any label
- $@l$: attribute
- $\slash$, $\mid$: concatenation (child), union
- $//$: descendants or self, “recursion”
- $[q]$: qualifier (filter, predicate)
  - $\text{op}$: $=$, $!=$, $<=$, $<$, $>$, $>=$, $>$
  - $c$: constant
  - $\text{and}$, $\text{or}$, $\text{not}()$: conjunction, disjunction, negation
XPath

Context node: starting point
XPath

Child

/\(a\) is equivalent to child::\(a\)
XPath

Descendant

/descendant::*
XPath

Descendant-or-self

//a is equivalent to descendant-or-self::*/child::a
XPath

Upward Traversal

Syntax:

\[ Q ::= \ldots | \ldots Q | \text{ancestor} :: Q | \text{ancestor-or-self} :: Q \]

✓ \ldots : parent
✓ ancestor, ancestor-or-self: recursion

Abreviations:

. is equivalent to self::*

.. is equivalent to parent::*
XPath

Ancestor
XPath

Ancestor-or-self
XPath

Sideways

Syntax:

Q ::= ... | following::Q | preceding::Q |
   | following-sibling ::Q | preceding-sibling::Q |
   | [p] (p is integer)

✓ **following-sibling**: the right siblings
✓ **preceding-sibling**: the left siblings
✓ **position function** (starting from 1): e.g., //author[position( ) < 2]
XPath

Preceding-Sibling
XPath

Following
XPath

Preceding
XPath

Self
XPath

Positional Tests

//*[position()=2] (or just //*[2])
XPath

Positional Tests

//a/*[first()]
XPath

Positional Tests

//a/*[last()]
Outline

• Overview on XML
• Why XML Security?
• Querying Views-based XML Data
• Updating Views-based XML Data
XML Security

- XML data management
  - Business information: Confidential
  - Health-care data: the Patient Privacy Act

- Selective divulgence of XML data
  - A major concern for data providers and consumers
  - Preserving data confidentiality, privacy and intellectual property
Example

XML database containing medical records

Hospital

Psychiatry

Genetics

Record

Date

Doctor

Bill

Patient

Diagnosis

Name

Sex

Name

David

Mark

Record

Date

Doctor

Bill

Patient

Diagnosis

Name

Sex

Name

David

Mary

Record

Date

Doctor

Bill

Patient

Diagnosis

Name

Sex

Name

Angela

Mary

The Administrator could see the whole database
Example

XML database containing medical records

Doctor David can only access the records of his patients
Example

XML database containing medical records

- Hospital
  - Psychiatry
    - Record
      - Date
      - Doctor: 'David'
      - Bill
      - Patient: 'Mark'
      - Diagnosis
      - Name: 'David'
    - Record
      - Date
      - Doctor: 'David'
      - Bill
      - Patient: 'Mary'
      - Diagnosis
      - Name: 'Mary'
  - Genetics
    - Record
      - Date
      - Doctor: 'Angela'
      - Bill
      - Patient: 'Mary'
      - Diagnosis
      - Name: 'Angela'

Example

XML database containing medical records

Patient Mary can access his own medical records
XML Access Control

- Access control
  - multiple groups simultaneously query the same XML document
  - each user group has a different access-control policy

- Enforcement of access-control policies:
XML Access Control

For each user group of an XML document $T$,

- specify a access-control policy $S$,
- enforce $S$: for any query $Q$ posted by the group over the document $T$, $Q(T)$ consists of only data accessible w.r.t $S$.

Problems with access control for XML:

- How to specify access policies at various levels of granularity?
- How to efficiently enforce those access policies?
Models for XML Security

Several models have been proposed for XML: XACML, XACL, ...

✓ Specifying and enforcing access-control at a physical level
  - annotate data nodes in an XML document with accessibility, and check accessibility at runtime (with optimizations for tree-pattern queries and tree/DAG DTDs)

✓ Problems:
  - costly (time, space): multiple accessibility annotations
  - error-prone: integrity maintenance becomes a problem when the underlying data or access policy is updated
Models for XML Security

Several models have been proposed for XML: XACML, XACL, ...

- **Using at a Security Views**: multiple user groups
  - who wish to query the same XML document
  - different access policies may be imposed, specifying **the portions of the document** the users are **granted** or **denied** access to.

- **Two types of security views are used**
  - Virtual views
  - Materialized views
XML Views

- **Materialized views:** store data in the views
  - **Query support:** straightforward and efficient
  - **Consistency:** the views should be updated in response to changes to the underlying database

- **Virtual views:** do not store data
  - **Query support:** view queries should be translated to equivalent ones over the underlying data
  - **Updates:** not an issue
Virtual vs. Materialized

XML views are important for data exchange, Web services, access control (security), Web interface for scientific databases, ...

✓ **Materialized views**: publishing
  - sometimes necessary, e.g., XML publishing
  - when response time is critical, e.g., active system
  - “static”: the underlying database is not frequently updated

✓ **Virtual views**: shredding
  - “dynamic”: when the underlying data source constantly changes and/or evolves
  - Web interface: when the underlying database and the views are large
  - Access control: multiples views of the same databases are supported simultaneously for different user groups
Access Control Specification

Definition of rules for restricting access in XML data using various levels of granularity (entire subtrees or specific elements).

Each rule is a tuple of:

- **Requestor**
  - The user of set of users concerned by the authorization

- **Resource**
  - The data that the requestor is (or not) granted to access

- **Action**
  - The action (read, write, etc) is (or not) allowed on the resource

- **Effect**
  - It grants (sign '+' or denies (sign '-' access to the resource

- **Propagation**
  - It defines the scope of the rule
Language for Access Control

XPath language is used to specify the XML nodes concerned by an access rule.

Each rule’s resource is defined as a XPath expression:

- Accessible /Inaccessible nodes
- Conditional accessible nodes

- XPath is a navigation language that returns a subset of nodes
  - It is used by XML-related technologies (XQuery, XSLT, etc)

- Different XPath fragments are used
  - Navigational axis (e.g. child, descendent, attributes, etc)
  - Comparison operators (e.g. testing only equality)
  - Expressions are absolute or relative
Scope for Access Control Rule

Due to the hierarchical nature of XML: how to apply the access rule?

The access rule is **local** if the scope can be:
- The node only
- The node and its attributes
- The node and its text value

The access rule is **recursive** if the scope can be:
- The node, its attributes, all its descendants and their attributes
- Entire sub-trees
- *inheritance*: some nodes inherit the accessibility of their ancestors
Default Semantics

Given an access control policy, there is a question:

What happens to the node if there exists no access control rule that neither grants nor denies access to it?

The default semantics of the access control policy gives an implicit rule in this case. There are two semantics:

- **Deny**
  - The node is non-accessible
- **Grant**
  - The node is accessible
Conflict Resolution

A conflict occurs when a node is granted access (by a positive rule) and denied access (by a negative rule) at the same time.

There are different approaches to perform conflict resolution:

- **Priorities**
  - Each rule is assigned a priority and the rule with highest priority is considered

- **Deny overwrites**
  - Negative rule takes precedence over positive rule

- **Grant overwrites**
  - Positive rule takes precedence over negative rule
Example

XML database containing medical records

Rule: (Toto, //Name, Read, +, local)
Default semantics: Deny
Example

XML database containing medical records

Rule: (Toto, //Name, Read, +, local)
Default semantics: Deny
Example

XML database containing medical records

Rule: (Toto, //Record[./Patient/Name='Mark'], Read, +, recursive)
Default semantics: Deny
Example

XML database containing medical records

Rule: *(Toto, //Record[Patient/Name='Mark'], Read, +, recursive)*
Default semantics: *Deny*
Date, Doctor, Bill, Diagnosis, ... inherit the accessibility of Record
Example

XML database containing medical records

Rule 1: (Toto, //Patient/Name, Read, -, local)
Rule 2: (Toto, //Record[./Doctor/Name='David'], Read, +, recursive)
Default semantics: Deny
Example
XML database containing medical records

Rule1: (Toto, //Patient/Name, Read, -, local)
Rule2: (Toto, //Record[./Doctor/Name='David'], Read, +, recursive)
Default semantics: Deny
Conflict resolution policy: Deny
Outline

• Overview on XML
• Why XML Security?
• **Querying Views-based XML Data**
• Updating Views-based XML Data
Access control for XML Data proposed by Fundulaki et al. [Iri 2004].

- **XPath fragment**
  
  \[
  \text{locapath} ::= \text{axis} '::' \text{ntst} ['\text{expr}'] | '/' \text{locapath} | \text{localpath} '/\text{localpath}' \\
  \text{expr } ::= \text{localpath} | \text{not expr} | \text{expr and expr} | \text{expr or expr} \\
  | \text{locapath op v}
  \]

  - \text{ntst} is a node label, * or function \text{text()}
  - \text{op} is comparison operator (e.g. \text{<=})
  - \text{v} is a value

- **Access Control Policy**
  
  - Defined by four sets of XPath filter expressions

  \[
  P_l, P_r: \text{positive local and recursive rules} \\
  N_l, N_r: \text{negative local and recursive rules}
  \]
XML without Schema

Example: XML database containing medical records

1. Grant access to all nodes: \( P_r = \{\ast\} \)

2. Only Name nodes are accessible: \( P_l = \{\text{Name}\} \)

3. All nodes are accessible except Diagnosis: \( P_r = \{\ast\}, N_l = \{\text{Diagnosis}\} \)

4. Grant access to the Record nodes and all its descendant nodes, except if they are below a Patient node whose Name node has the value 'Mark':

\[
\begin{align*}
P_r &= \{\text{Record}\} \\
N_r &= \{\text{Patient}[./\text{Name}='\text{Mark}']\}
\end{align*}
\]
XML without Schema

Enforcement of Access Control

• A XML document: D

• A query as a XPath expression: q

• An access control policy: ACP

• The query q is rewritten into q[expr] where expr is XPath expression, obtained from ACP in such a way the answer set of q must be filtered to obtain only the accessible node
XML without Schema

Access Control Policies with Only Local Rules

A node is accessible if there exists:
1. at least one positive rule that grants access to it, and
2. no negative rule that denies access to it

\[ q[expr] \]

✓ q targets element nodes

[expr] is \[ \bigvee_{p \in P_l} \text{self} :: p \bigwedge_{f \in N_l} \text{not self} :: f \]

✓ q targets attribute/text nodes

[expr] is \[ \bigvee_{p \in P_l} \text{parent} :: p \bigwedge_{f \in N_l} \text{not parent} :: f \]
XML without Schema

Access Control Policies with Only Recursive Rules

A node is **accessible** if:

1. there exists a positive rule that grants access to one of its ancestors, or the node itself, **and**

2. no negative rule that denies access to one of its ancestors or the node itself

\[ q[expr] \]

\[(1): \bigvee_{p \in P_r} \text{ancestor} - \text{or} - \text{self} :: p \]

\[(2): \bigwedge_{f \in N_r} \text{not ancestor} - \text{or} - \text{self} :: f \]
XML without Schema

Access Control Policies with Local and Recursive Rules

A node is accessible if:
1. there exists at least one positive recursive rule that grants access to it, or
2. there exists at least one positive local rule that grants access to it, and
3. there is no negative recursive rule, and
4. there is no negative local rule that denies access to it

$q[expr]$ 

1. $\left(\bigvee_{p \in P_r} \text{ancestor} - \text{or} - \text{self} :: p\right)$ or
2. $\bigvee_{p \in P_l} \text{self} :: p$ and
3. $\bigwedge_{f \in N_r} \text{not ancestor} - \text{or} - \text{self} :: f$ and
4. $\bigwedge_{f \in N_l} \text{not self} :: f$
XML without Schema

Problem: Security Breaches

Only nodes files and name are accessible: $P_1 = \{\text{files}\}, P_1 = \{\text{name}\}$

Query /files/record/name is rewritten in /files/record/name[self::name]
XML without Schema

Problem: Security Breaches

<files>
  <record>
    <name>Robert</name>
    <diagnosis>
      Pneumonia
    </diagnosis>
  </record>
  <record ...
</files>

Only nodes files and name are accessible: $P_i = \{\text{files}\}$, $P_i = \{\text{name}\}$

Query /files/record/name is rewritten in /files/record/name[self::name]
  ➔ Discloses the existence of hidden node
XML without Schema

Problem: Security Breaches

Only nodes files and name are accessible: \( P_I = \{\text{files}\}, P_I = \{\text{name}\} \)

Query \(/\text{files}/\text{record}/\text{name}\) is rewritten in \(/\text{files}/\text{record}/\text{name}[self::\text{name}]\)

\( \rightarrow \) Discloses the existence of hidden node

Solution: examining all nodes parsed in the query
XML without Schema

Problem: Rewriting may be impossible

Only nodes files and name are accessible: $P_1 = \{\text{files}\}, P_1 = \{\text{name}\}$

Query /files/name is rewritten in /files/name[self::name]

→ This query will be rejected
**XML without Schema**

**Problem: Rewriting may be impossible**

Only nodes files and name are accessible: \( P_1 = \{ \text{files} \}, P_1 = \{ \text{name} \} \)

Query /files/name is rewritten in /files/name[self::name]

⇒ This query will be rejected

**Solution:** Denial Downward Consistency Property

if a node is inaccessible then all its descendants are inaccessible
XML with Schema

Access control for XML Data proposed by Fan et al. [Fan 2004].

- **Security administrator**: specifies a access-control policy for each group by extending the document DTD with XPath qualifiers

- **Derivation module**: automatically derives a security-view definition from each policy: view DTD and mapping via XPath

- **Query translation module**: rewrite and optimize queries over views to equivalent queries over the underlying document
XML with Schema

Access control for XML Data proposed by Fan et al. [Fan 2004].

✓ Specification and enforcement: at the conceptual (schema) level
  - no need to update the underlying XML data
  - no need to materialize views or perform runtime check

✓ Schema availability: view schema is automatically derived
  - characterizing accessible data
  - exposing necessary schema information only
XML with Schema

Access control Specification

DTD $D$: element type definitions $A \rightarrow \alpha$

\[
\alpha ::= \text{PCDATA} \mid \varepsilon \mid A_1, \ldots, A_k \mid A_1 + \ldots + A_k \mid A^*
\]

Annotations are added in the DTD document to define the access control policy

\[
\text{Access policy} = \text{Document DTD} + \text{XPath qualifiers}
\]
XML with Schema

Access control Specification

✓ Specification \( S = (D, \text{access}(\ )) \): a mapping \( \text{access}(\ ) \) from the edges in the DTD document \( D \) \to \{ Y, N, [q] \}.

For each \( A \rightarrow \alpha \), for each \( B \) in \( \alpha \), define \( \text{Access}(A, B) \) as

- \( Y \): accessible (true)
- \( N \): inaccessible (false)
- \( [q] \): XPath qualifier, conditional: accessible iff \( [q] \) holds

XPath fragment:

\[
p ::= \epsilon \mid A \mid \ast \mid \mathbin{//} \mid p/p \mid p \cup p \mid p[q]
\]
\[
q ::= p \mid p = "c" \mid q_1 \land q_2 \mid q_1 \lor q_2 \mid \neg q
\]
XML with Schema

Example: an XML document of patients

Document DTD D

hospital → patient*
patient → SSN, name, record*
record → date, diagnosis, treatment
treatment → (trial + regular)
trial → trName, treatment*
regular → tname, bill

Access-control policies over docs of D:
✓ Doctors in the hospital are granted access to all the data in the docs
✓ Insurance company is allowed to access billing information only
XML with Schema

Example: an XML document of patients

\[
\text{access}(\text{hospital}, \text{patient}) = [\text{//diagnosis} = \text{"DIS"}] \quad \text{-- [q1]}
\]
\[
\text{access}(\text{patient}, \text{record}) = [\text{diagnosis} = \text{"DIS"}] \quad \text{-- [q2]}
\]
\[
\text{access}(\text{treatment}, \text{trial}) = \text{N}
\]
\[
\text{access}(\text{treatment}, \text{regular}) = \text{N}
\]
\[
\text{access}(\text{regular}, \text{tname}) = \text{Y}
\]

✓ **overriding:** if \( \text{access}(A, B) = \text{Y (N)}, \) then the B children of A override the accessibility of A

✓ **inheritance:** if \( \text{access}(A, B) \) is not explicitly defined, then the B children of A inherit the accessibility of A

✓ **content-based:** conditional accessibility via XPath qualifiers

Conditionally accessible
XML with Schema
Properties of the specification language

- XML tree of the document DTD: the accessibility of each data node is uniquely defined by an access specification
  - relative to the path from root
  - a qualifier at a node a constrains the entire subtree rooted at a,
    - e.g., [q2] constrains tname

- various levels of granularity: entire subtrees or specific elements

- schema level: the underlying XML data is not touched; efficient, easy to specify and maintain

Conditionally accessible
XML with Schema

Enforcing Access Control - Security Views

XML security view: $\sigma = (Dv, \text{xpath}(\ ))$ with respect to an access policy $S = (D, \text{access}(\ ))$,

- $Dv$: view DTD, exposed to the user and characterizing the accessible information (of document DTD $D$) w.r.t $S$

Schema availability: to facilitate query formulation

- $\text{xpath}(\ )$: mapping from instances of $D$ to instances of $Dv$ defined in terms of XPath queries and view DTD $Dv$
  - for each $A \rightarrow \alpha$ in $Dv$, for each $B$ in $\alpha$, $\text{xpath}(A, B) = p$
  - $p$: generates $B$ children of an $A$ element in a view
    - $p ::= \varepsilon \mid A \mid * \mid // \mid p/p \mid p \cup p \mid p[q]$
    - $q ::= p \mid p = "c" \mid q_1 \land q_2 \mid q_1 \lor q_2 \mid \neg q$
One needs an algorithm to compute a security-view definition:

- **Input:** an access policy $S = (D, \text{access}())$
- **Output:** a security-view definition $\sigma = (D_v, \text{xpath}())$
  - **sound:** accessible information only
  - **complete:** all the accessible data (structure preserving)
  - **DTD-conformant:** conforming to the view DTD
- **efficient:** $O(|S|^2)$ time (proposed in [Fan2004])
- **generic:** recursive/nondeterministic document DTDs
XML with Schema

Example: an XML document of patients

- Top-down traversal of the document DTD D
- short-cutting/renaming (via dummy) inaccessible element types
- normalizing the view DTD $D_v$ and reducing dummy types

```
xpath(hospital, patient) = hospital/patient[q1]
```

```
xpath(patient, record) = record[q2]
```

```
xpath(record, treatment) = treatment
```
XML with Schema

Example: an XML document of patients

- recursive and non-deterministic productions

\[
\text{xpath}(\text{treatment}, \text{dummy2}) = \text{regular} \\
\text{xpath}(\text{treatment}, \text{dummy1}) = \text{trail}
\]

- reducing dummy element types:
  \[
  (\text{dummy1/\text{treatment}})^* / \text{dummy2 / tname} \cup \text{dummy2/\text{tname}})
  \Rightarrow (\text{dummy1/\text{treatment}})^* / \text{dummy2 / tname} \Rightarrow \text{tname}^*
  \]

\[
\text{xpath}(\text{treatment, tname}) = //\text{tname}
\]
XML with Schema

Rewriting Algorithm

✓ Input:
  - $\sigma = (Dv, \text{xpath( )})$ (security view wrt $S = (D, \text{access( )})$), and
  - an XPath query $Qv$ over the view $(Dv)$

✓ Output: an equivalent XPath query $Qt$ over the document
  - for any XML document $T$ of $D$, $Qt(T) = Qv(\sigma(T))$

Dynamic programming:

✓ for any subquery $Qv'$ of $Qv$, any node $A$ in view-DTD graph $Dv$
  rewrite $Qv'$ at $A$ by incorporating $\text{xpath}(A, _) \Rightarrow Qt'$ ($A$)

✓ efficient: $O(|Qv| \times |\sigma|^2)$ time

✓ a practical class of XPath (with union, descendant, qualifiers) vs.
  tree-pattern queries studied in previous security models
XML with Schema

Example: an XML document of patients

Qv = // patient[name="Joe"] // tname over the view

Qt = /hospital/patient[name = "Joe" and //diagnosis = "DIS"]
   /record[diagnosis = "DIS"]
   /treatment // tname

Equivalent query over document

xpath(hospital, patient) [name = "Joe"] /
  xpath(patient, record) /
  xpath(record, treatment) /
  xpath(treatment, tname)
XML with Schema

Problems when using “dummy” nodes

Replacing inaccessible nodes with anonymous nodes

Original XML Document

- competition
  - Engineering School
    - candidate
      - exams
  - University
    - Department
      - candidate
        - exams

User view

- competition
  - Dummy 1
  - Dummy 2
    - exams
  - Dummy 3
  - Dummy 4
    - exams
  - Dummy 5
XML with Schema

Problems when using “dummy” nodes

User queries may contain “dummy” nodes

Original XML Document

competition

Engineering School

University

Department

candidate

candidate

exams

exams

User view

competition

Dummy 1

Dummy 2

Dummy 3

Dummy 4

Dummy 5

exams

exams

pre-processing

//candidate/exams

//dummy2/exams
XML with Schema

Problems when using “dummy” nodes

User queries may disclose some confidential information

Original XML Document

User view

---

competition

Engineering School

University

Department

candidate

candidate

exams

exams

dummy3, dummy4,...

dummy3, dummy4,...

/*={university, department,...}
XML with Schema

Problems when using “dummy” nodes

User queries do not contain “dummy” nodes ...

Original XML Document

User view

... Difficult to express some queries (e.g. exams under Dummy 2)
Problem: the XPath fragment is not closed un rewriting

XPath fragment:
\[ p ::= \varepsilon \mid A \mid * \mid // \mid p/p \mid p \cup p \mid p[q] \]
\[ q ::= p \mid p = "c" \mid q_1 \land q_2 \mid q_1 \lor q_2 \mid \neg q \]

Q\text{v} = //D over Dv
xpath(Root,A) = A
xpath(A,D) = D \cup B/D \cup C/D

Q\text{v} is rewritten into
\[ Q = /Root/xpath(Root,A)/(D \cup B/D) \]
over D1
XML with Schema

Problem: the XPath fragment is not closed un rewriting

XPath fragment:
\[ p ::= \varepsilon | A | * | // | p/p | p \cup p | p[q] \]
\[ q ::= p | p = "c" | q_1 \land q_2 | q_1 \lor q_2 | \neg q \]

\[ Qv = //D \text{ over } Dv \]
\[ \text{xpath} (\text{Root}, A) = A \]
\[ \text{xpath} (A, D) = D \cup B/D \cup C/D \cup C/C/D \]
\[ \ldots \cup C/C/C/C/D \ldots \]

Qv cannot be rewritten as \text{xpath}(A, D) leads to infinitely many paths
XML with Schema

Problem: the XPath fragment is not closed on rewriting

XPath fragment:
\[
p ::= \varepsilon \mid A \mid \ast \mid // \mid p/p \mid p \cup p \mid p[q]
\]
\[
q ::= p \mid p = "c" \mid q_1 \land q_2 \mid q_1 \lor q_2 \mid \neg q
\]

Qv = //D over Dv

xpath(Root,A) = A

xpath(A,D) = D \cup B/D \cup C/D \cup C/C/D

... \cup C/C/C/C/D ...

Qv cannot be rewritten as xpath(A,D)
leads to infinitely many paths

XPath does not contain the Kleene Star
XML with Schema

Solution 1: Using Regular XPath for rewriting [Fan 2007]

Capture DTD recursion and XPath recursion in a uniform framework
✓ Regular XPath:
  \[ Q ::= \varepsilon \mid A \mid Q/Q \mid Q \cup Q \mid Q^* \mid Q[q] \]
  \[ q ::= Q \mid Q = 'c' \mid q \land q \mid q \lor q \mid \text{not } q \]
✓ The child-axis, Kleene closure, union
✓ An XPath fragment: \( Q//Q \) instead of \( Q^* \)

Example:
/Root/A /C//C/D is translated into
/Root/A/(C) */D

[Diagram]

DTD D2

View Dv
XML with Schema

Solution 1: Using Regular XPath for rewriting [Fan 2007]

Drawback of Regular XPath Query

- the size of the rewritten query $Q_T$, if directly represented in Regular XPath, may be exponential in the size of input query $Q_V$.

- Regular XPath remains a theoretical achievement (it is not yet accepted as a standard)

- There are no translation and evaluation tools
XML with Schema

Solution 2: Extending the fragment for rewriting [Mah 2012]

Using two XPath fragments in a uniform framework

- XPath fragment $F$ for expressing queries:
  
  \[
  p ::= \varepsilon \mid A \mid * \mid \mathbb{X} \mid p/p \mid p \cup p \mid p[q] \\
  q ::= p \mid p = \text{"c"} \mid q_1 \land q_2 \mid q_1 \lor q_2 \mid \neg q
  \]

- Extended XPath fragment for rewriting queries:
  
  $F + ..\mathbb{X}Q \mid ancestor :: Q \mid ancestor-or-self :: Q \mid p[n]$  
  - $..\mathbb{X}$: parent
  - ancestor, ancestor-or-self: ascendant axis
  - $p[n]$: Position function
XML with Schema

Solution 2: Extending the fragment for rewriting [Mah 2012]

\[ Q_v = //D \text{ over } D_v \text{ such that } \text{xpath}(\text{Root}, A) = A \]
\[ \text{xpath}(A, D) = D \cup B/D \cup C/D \cup C/C/D \ldots \cup C/C/C/C/D \ldots = D \cup B/D \cup D[\text{ancestor::C}[1]] \]

\[ Q_v \text{ is rewritten into } //D[\text{not ancestor::C}[1]] \]
Outline

- Overview on XML
- Why XML Security?
- Querying Views-based XML Data
- **Updating Views-based XML Data**
XML Updates

Input: an XML tree $T$ and XML update $\Delta T$

Output: updated XML tree $T' = T + \Delta T$
Atomic Updates

Basic changes that can be applied to tree

u ::= insertInto(n,t)
     | insertAsFirstInto(n,t)
     | insertAsLastInto(n,t)
     | insertBefore(n,t)
     | insertAfter(n,t)
     | delete(n)
     | replace(n,t)
     | replaceValue(n,s)
     | rename(n,a)
Atomic Updates

Insertion

- InsertInto (c, <x><y/></x>)

Before

After
Atomic Updates

Insertion

- InsertAsFirstInto \((c, <x><y/></x>)\)
Atomic Updates

Insertion

- InsertAsLastInto (c, <x><y/></x>)
Atomic Updates

Insertion

- InsertBefore \((c, <x><y/></x>)\)

Before

After
Atomic Updates

Insertion

- InsertAfter (c, <x><y/></x>)

Before

After
Atomic Updates

Deletion

- Delete (c)

Before:

After:
Atomic Updates

Replace Text Value

- ReplaceValue (d, “toto”)

Before

After

toto
Atomic Updates

Replace Subtree

- Replace \((c, <x><y/><z/></x>)\)

Before

After
Atomic Updates

Rename

- Rename \((c, x)\)

Before

After
Access Control with Updates

Existing access control approaches

- Most of XML access control approaches deal only with *read access rights*
- Access control considering *update rights* has not received more attention
- The *XQuery Update Facility*: a recommendation of W3C providing facility to modify XML documents

Drawbacks

- Existing update access control languages are *unable* to specify some update policies in case of recursive DTDs
- *No practical tool exists* for securely querying and updating XML data over recursive DTDs
Access Control with Updates

Example of DTD: Biopolymer and Genealogical Data

(a) BIOML

(b) GedML
Access Control with Updates

Example of DTD: Hospital Data

(a) Patient Treatment
   Life-cycle Management

(b) Corresponding DTD
Basic Notions

DTD (Document Type Definition)

A DTD $D$ is a triple $(Ele, Rg, root)$ where:

- $Ele$ is a set of element types;
- $root$ is a distinguished element type, called the root type;
- $Rg$ is a function such that for any $A$ in $Ele$, $Rg(A)$ is a regular expression of the form:

$$\alpha ::= \text{str} \mid \epsilon \mid B \mid \alpha', \alpha \mid \alpha' \alpha \mid \alpha^* \mid \alpha^+ \mid \alpha?$$

- $A \rightarrow Rg(A)$ is the production of $A$;
- $B$ is a child type of $A$, and $A$ is a parent type of $B$;
- $D$ is recursive if there is an element type $A$ defined in terms of itself directly or indirectly.
Basic Notions

Xquery Update Operations

In the following, *source* is a set of XML nodes, and *target* is an XPath expression which returns a single node in case of *Insert* and *Replace* operations.

- **Insert source into target**: insert nodes in source as children of target's node.
- **Insert source as first/as last into target**: insert nodes in source as first (resp. as last) children of target's node.
- **Insert source before/after target**: insert nodes in source as preceding (resp. following) sibling nodes of target's node.
- **Replace target with source**: replace target's node with the nodes in source.
- **Replace value of target with string-value**: replace the text-content of target's node with the new value *string-value*.
- **Delete target**: delete nodes returned by target along with their descendant nodes.
- **Rename target with string-value**: rename the label of target's node with the new label *string-value*.
Access Control Policy

Goals

For each user group of an XML document $T$:  
- **Specifying** an update-access policy $S$
- **Enforcing** $S$ at update time: any update $op$ must be performed only at nodes that are updatable w.r.t. $S$.

Challenges

- How to specify update policies at various granularity levels?
- How to specify update policies over arbitrary DTDs?
- How to efficiently enforce those update policies?
Access Control Policy

Example: Doctor Update Policy

Update Policy:
Each doctor can update only data of treatments that she/he has done.
Access Control Policy

Example: Update rights of Dr Imine

User update:
Delete //treatment [type='surgery']/Tresult
ERROR: insufficient privilege
Existing Access Control Models

Model of Fundulaki et al. [Fun2007]

- An XPath-based rules language (\textbf{XACU}) is proposed to specify update policies.
- An \textbf{XACU} rule has the form: \textit{\texttt{(object, action, effect)}}.
- An \textbf{XACU} rule can be \textit{positive/negative, local/recursive}.
- \textit{Grant/Deny} overrides as conflict resolution policy.

Drawbacks

- The \textbf{XACU} language can be used only for non-recursive DTDs.
Existing Access Control Models

Model of Fundulaki et al. [Fun2007]

Update Policy:

Each doctor can update only data of treatments that she/he has done.
Existing Access Control Models

Model of Fundulaki et al. [Fun2007]

Update Policy:
Each doctor can update only data of treatments that she/he has done.

Some XACU rules:
- (intervention[doctor/dname='Imine']/treatment, delete, +)
- (intervention[doctor/dname≠'Imine']/treatment, delete, -)
Existing Access Control Models

Model of Fundulaki et al. [Fun2007]

Some XACU rules:

- (//intervention[doctor/dname='Imine']/treatment, delete, +)
- (//intervention[doctor/dname!='Imine']/treatment, delete, -)

Limitation:

Nodes treatment_3 and treatment_4 are in the scopes of both the two XACU rules.

Grant overrides: node treatment_3 becomes updatable for Imine.

Deny overrides: node treatment_4 becomes not updatable for Imine.
Existing Access Control Models

Model of Damiani et al. [Dam2008]

- Update policies are defined by annotating element types of the DTD by security attributes.

- E.g., attribute $@\text{insert}=\text{Y}$ on element type $\text{treatment}$ specifies that some nodes can be inserted as children of treatment nodes.

- Update policy is translated into security automaton.

- Each update operation is rewritten into a safe one by parsing this automaton.

Drawbacks

- Query rewriting over automaton is guaranteed only when DTDs are non-recursive

- Update annotations are local which is insufficient to specify some update constraints.
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Security Administrator:

*Specifies* for each group of users an update policy by annotating the DTD with update constraints (i.e. XPath qualifiers).

Updates Rewriter Module:

*Translates* each update operation into a *safe* one in order to be performed only over nodes that can be updated w.r.t. the update policy.
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Update Specification: Update policy = DTD + XPath Qualifiers

An update specification $S=(D, Annot)$: $Annot$ is a mapping from element types of $D$ into: $Y$, $N$, $[Q]$.
For an element type $A$ in $D$, and an update of type $op$, define $Annot(A, op)$ as:

- $Y$: operation of type $op$ can be performed at nodes of type $A$.
- $N$: operation of type $op$ cannot be performed at nodes of type $A$.
- $[Q]$: operation of type $op$ can be performed at node of type $A$ iff $[Q]$ is valid.

Update types:

We define restricted update operations that can be performed only for some specific element types. E.g. $insertInto[B]$, $delete[B]$, $replaceNode[Bi, Bj]$.

Local and recursive rules:

Inheritance and overriding of update rights
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Example: Update Policy for Nurses

Update specification:

- \text{Annot} (\text{department}, \text{insertInto}[\text{patient}]) = [\text{name}='\text{Critical care}']

- \text{Annot} (\text{Sibling}, \text{insertInto}[\text{patient}]) = N
Existing Access Control Models

**Model of Mahfoud et al. [Mah2012]**

Example: Update Policy for Dr Imine

Each doctor can update only data of treatments that she/he has done.

Update specification:

- $\text{Annot}(\text{intervention}, \text{replaceValue}[Tresult]) = \{\text{dname}='\text{Imine}'\}$
- $\text{Annot}(\text{intervention}, \text{insertAfter}[\text{type}, Tresult]) = \{\text{dname}='\text{Imine}'\}$
- $\text{Annot}(\text{intervention}, \text{delete}[Tresult]) = \{\text{dname}='\text{Imine}'\}$
Rewriting principle:

Given an update specification $S=(D, \text{Annot})$ and an update operation $op$ over an instance $T$ of $D$. We rewrite $op$ into a safe one $op^+$ such that executing $op^+$ over $T$ has to modify only nodes that are updatable w.r.t. $S$.

Rewriting Problem:

Consider the XPath fragment $\mathcal{X}$ defined as follows:

\[
\begin{align*}
p & := \alpha :: \text{lab} \mid p[q] \mid p/p \mid p \cup p \\
q & := p \mid p/text()='c' \mid q \text{ and } q \mid q \text{ or } q \mid \text{not } (q) \\
\alpha & := \varepsilon \mid \downarrow \mid \downarrow^+ \mid \downarrow^*
\end{align*}
\]

For recursive DTDs, the fragment $\mathcal{X}$ is not closed under update operations rewriting.
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Example: Update Policy for Dr Imine

1. Annot(intervention, delete[Tresult]) = [dbname='Imine']

User update:

- Delete //Tresult cannot be rewritten in $\chi$
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Example: Update Policy for Dr Imine

\[ \text{Annot}(\text{intervention}, \text{delete}[\text{Tresult}]) = [\text{dname}='\text{Imine'}] \]

User update:

- Delete \( [\text{Tresult}] \) cannot be rewritten in \( \chi \)

A possible rewriting:

Delete \(/\text{intervention}[\text{doctor/dname}='\text{Imine'}]/\text{treatment/ (implies/diagnosis/treatment)*/Tresult} \)
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Example: Update Policy for Dr Imine

\[
\text{Annot(intervention, delete[Tresult])} = \ [\text{dname='Imine']}\]

User update:

- Deleted \(//Tresult\) cannot be rewritten in \(\chi\)

- A possible rewriting:

  Delete \(//\text{intervention[doctor/dname='Imine']}/\text{treatment/}(\text{implies/diagnosis/treatment})^*/\text{Tresult}\)

LIMIT. The kleene star (*) cannot be expressed in the standard XPath.
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Solution:

We extend fragment $\mathcal{X}$ as follows:

$$ p \ := \ \alpha :: \text{lab} \mid p[q] \mid p/p \mid p \cup p \mid p [n] $$
$$ q \ := \ p \mid p/text() = 'c' \mid q \text{ and } q \mid q \text{ or } q \mid \text{not} (q) $$
$$ \alpha \ := \ \varepsilon \mid \downarrow \mid \downarrow^+ \mid \downarrow^* \mid \uparrow \mid \uparrow^+ \mid \uparrow^* $$

We extend $\mathcal{X}$ into $\mathcal{X}_{[n]}^{\uparrow}$ by adding upward axes (parent, ancestor, and ancestor-or-self), and the position predicate (i.e., [n]).

For recursive DTDs, the fragment $\mathcal{X}_{[n]}^{\uparrow}$ is closed under update operations rewriting.
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Update Rewriting Algorithm

- **Input:**
  An update specification $S=(D, Annot)$ and an update operation $op$ defined in $X$.

- **Output:**
  A safe update $op^t$ defined in $X_{[n]}^\uparrow$ such that executing $op^t$ over any instance $T$ of $D$ has to modify only nodes that are updatable w.r.t. $S$.

- **Efficiency:**
  For any update specification $S=(D, Annot)$ and any update operation $op$, rewriting of $op$ can be done in $O(|Annot|)$ time.
Existing Access Control Models

Model of Mahfoud et al. [Mah2012]

Example: Update Policy for Dr Imine

- $\text{Annot}(\text{intervention}, \text{delete[}T\text{result}\text{]}) = [\text{dname}='\text{Imine}']$

User update:

- Delete //Tresult can be rewritten in $\chi[n]$

Delete //Tresult[ancestor::intervention[1] [doctor/dname='Imine']]$

Which has to delete nodes $T\text{result}_1$, $T\text{result}_2$ and $T\text{result}_3$. 
Some References


• Some slides of this course are inspired from lectures taught by Pr Wenfei Fan.