CSchema: A Downgrading Policy Language for XML Access Control

Dong-Xi Liu (刘东喜)

Department of Mathematical Informatics, University of Tokyo, Japan
E-mail: liu@mist.i.u-tokyo.ac.jp

Received October 15, 2005; revised July 2, 2006.

Abstract The problem of regulating access to XML documents has attracted much attention from both academic and industry communities. In existing approaches, the XML elements specified by access policies are either accessible or inaccessible according to their sensitivity. However, in some cases, the original XML elements are sensitive and inaccessible, but after being processed in some appropriate ways, the results become insensitive and thus accessible. This paper proposes a policy language to accommodate such cases, which can express the downgrading operations on sensitive data in XML documents through explicit calculations on them. The proposed policy language is called calculation-embedded schema (CSchema), which extends the ordinary schema languages with protection type for protecting sensitive data and specifying downgrading operations. CSchema language has a type system to guarantee the type correctness of the embedded calculation expressions and moreover this type system also generates a security view after type checking a CSchema policy. Access policies specified by CSchema are enforced by a validation procedure, which produces the released documents containing only the accessible data by validating the protected documents against CSchema policies. These released documents are then ready to be accessed by, for instance, XML query engines. By incorporating this validation procedure, other XML processing technologies can use CSchema as the access control module.

Keywords access control, programming language, security policy, type system, XML

1 Introduction

XML is the standard data format for exchanging information on the Internet. In many cases, it is desirable to allow different users to access different parts in one document according to their privileges or roles. This is the problem of XML access control, which has attracted many research efforts to solve it. For example, some approaches[1–6] and standards[7,8] have been proposed.

Access control mechanism is always an integral part for information systems, in which access policy is used to specify the accessibility of the sensitive resources to different users. For example, the system security policy in UNIX systems can grant or deny access to files to users. The existing approaches to XML access control also follow this style. Their policy languages specify which elements in XML documents can be accessed and which cannot probably under some conditions[9].

This style is suitable for protecting physical objects, such as files or directories, but not necessarily suitable for protecting data stored in XML documents. For XML data protection, the released data sometimes needs to be computed from the original sensitive data, not directly selected from them. That is, the original sensitive data is not accessible, but the data computed from them is, as shown by the following motivating example. From the perspective of information flow security[10], this case asks downgrading of the sensitive data from high security level (inaccessible) to low security level (accessible).

1.1 Motivating Example

Consider the XML file in Fig.1(a), which stores the staff information for a company.

```
<staffs>
  <staff>
    <name>Tom</name>
    <salary>30</salary>
  </staff>
  <staff>
    <name>Peter</name>
    <salary>20</salary>
  </staff>
</staffs>
```

Fig.1. Example XML file. (a) Original data. (b) Data for the assistant.

The users of this file include all staffs in the department and a financial assistant. For some reason, the company needs to impose the following access policy:

* Tom and Peter can access the whole document except for each other's salary;
* the assistant is authorized to see a document in Fig.1(b), which is computed from the original data.

The document for the assistant includes two computed elements: the element number for the number of staffs and the element totalsalary for the sum of Tom and Peter's salary. In other words, the assistant is not allowed to know who is in the department and his/her salary.

The current approaches cannot specify such kind of policies because they are only able to hide inaccessible elements or release accessible elements existing in the
protected document, but cannot construct new accessible elements.

1.2 CSchema Overview

In this paper, programming language technologies, mainly types and operational semantics, are used to provide a novel way of enforcing flexible XML access control. The proposed policy language is called calculation-embedded schema (CSchema), which extends the ordinary schema languages, such as XML Schema or DTD, with protection type. This type consists of three components: the type of sensitive data in the original document, the type of the data accessible to users, and an expression to compute the accessible data. This expression is the embedded calculation in schema and describes the downloading operation. The intuitive meaning of protection type is that the sensitive data will be replaced by the data computed by the embedded expression. Since CSchema policies include embedded calculations, we provide a type system to guarantee their type correctness, and after type checking, this type system also generates a security view\cite{1} with respect to the checked CSchema policy.

Writing a CSchema policy is just similar to writing an ordinary schema for XML documents, except that when meeting with a type that corresponds to sensitive data, the type should be changed into a protection type. Some general XML processing languages, like Java or CDuce\cite{11}, can transform the original document to the released document shown in Fig.1, but writing policy with them is inconvenient since they cannot let policy writers focus on the sensitive data, explained more by the example in Section 6.

The access policy expressed in CSchema is enforced by an extended validation procedure. XML is an external format to represent data, and by validating an XML file against a schema, either an internal representation of the file is generated or the validation fails\cite{12}. In CSchema, when validating an external value against a protection type, instead of really performing validation as usual, the expression in this type will be evaluated and its result will be used as the internal value at the position of this external value, and thus this external value is hidden. If a document does not include sensitive data, then the CSchema policy for this document does not have any protection type, so the enforcement of this policy does not incur any overhead for access control purpose since it just performs the ordinary validation. The implementation of this enforcement method needs only a conservative extension to the ordinary validation procedure by supporting validation against protection type.

The contributions of this work are summarized as follows.

- Motivating the downgrading problem for XML access control, which moves the focus of downgrading policies from the traditional code level to data level.
- Designing a calculation-embedded schema language to express downgrading access policies for XML access control.
- Formalizing a type system to check the type correctness of CSchema policies and automatically generate security views.
- Enforcing CSchema policy by validation that can be implemented by a conservative extension to the ordinary validation procedure.

The remainder of this paper is organized as follows. Section 2 gives the syntax of CSchema and represents the motivating example by this syntax. In Section 3, the type system for CSchema is formalized and illustrated. Section 4 shows the validation procedure together with the dynamic semantics of embedded expressions and gives a property about CSchema. Section 5 discusses another potential application of CSchema and compares it with other approaches. Section 6 surveys the related work. Section 7 concludes this paper.

2 Syntax of CSchema

The syntax of CSchema is presented in Fig.2. It includes two parts: the syntax of types and the syntax of expressions. A CSchema policy is a type \( \tau \).

\[
\begin{align*}
 k &::= * | * \rightarrow * \\
 \tau &::= t (\text{type}) | \text{int} | \text{bool} | (\|) [\|] r \\
 &| \tau_1, \tau_2 [\tau_1 \rightarrow \tau_2] \rightarrow \tau \land \tau_1 \rightarrow \tau_2 \& e \mu. \tau \\
 e &::= d / p \mid \text{self} : / p f (e) \\
 d &::= x (\text{type}) | \text{isempty}(d) \mid x[| d_1 = e_1 \} n \} d_1 + d_2 \mid \text{true} \mid \text{false} \\
 &\mid \text{if } d \text{ then } d_1 \text{ else } d_2 \} (\|)[d] \} d/p \} d_1, d_2 \} \text{head}(d) \} \text{tail}(d) \} f (d) \\
 p &::= e \mid \text{child} : / p \\
 G &::= \Gamma, \text{fun} f (x : \tau_1) : \tau_2 = d
\end{align*}
\]

Fig.2. Syntax of CSchema.

2.1 Syntax of Types

The syntactic categories \( k \) and \( \tau \) in Fig.2 are relevant to the syntax of types. The types given here are built upon regular expression types\cite{13}. A type can be an atomic type including the type variable \( t \), the empty sequence \( () \), \text{string}, \text{int}, \text{bool}, or a composed type including the element type \( (\|) [\|] \), the sequence type \( \tau_1, \tau_2 \), the choice type \( \tau_1 | \tau_2 \), the type \( \tau^* \), the protection type \( \tau_1 \rightarrow \tau_2 \& e \) and the recursive type \( \mu. \tau. \) The occurrence modifier \( s \) means zero or more occurrences of the modified type. In this syntax, we omit other occurrence modifiers \? and \+. Actually, they can be defined using the existing constructs: \( \tau^? = () [\|] \) \tau \) for optional occurrence of \( \tau \), and \( \tau^+ = \tau, \tau^* \) for one or more occurrences of \( \tau \). The element type \( (\|) [\|] \) is said to be the parent of \( \tau \), and in turn \( \tau \) is its child. Later, \( b \) is often used to range over the set \{ \text{bool}, \text{int}, \text{string}, () \}.

The novelty in this type language is the protection type \( \tau_1 \rightarrow \tau_2 \& e \), where \( \tau_1 \), called original component, is the type of the original sensitive data; \( \tau_2 \), called view component, is the type of the data that replaces the