Balisage: **The Markup Conference**

Secure Publishing using Schema-level Role-based Access Control Policies for Fragments of XML Documents

Tomasz Müldner

Professor
Jodrey School of Computer Science, Acadia University
<tomasz.muldner@acadiau.ca>

Robin McNeill

Graduate Student
Jodrey School of Computer Science, Acadia University
<063637m@acadiau.ca>

Jan Krzysztof Miziołek

Director
Computing Services Centre for Studies on the Classical Tradition in Poland and East-Central Europe, University of Warsaw, Warsaw, Poland
<jkm@ibi.uw.edu.pl>

Abstract

Popularity of social networks is growing rapidly and secure publishing is an important implementation tool for these networks. At the same time, recent implementations of access control policies (ACPs) for sharing fragments of XML documents have moved from distributing to users numerous sanitized sub-documents to disseminating a single document multi-encrypted with multiple cryptographic keys, in such a way that the stated ACPs are enforced. Any application that uses this implementation of ACPs will incur a high cost of generating keys separately for each document. However, most such applications, such as secure publishing, use similar documents, i.e. documents based on a selected schema. This paper describes RBAC defined at the schema level, (SRBAC), and generation of the minimum number of keys at the schema level. The main advantage of our approach is that for any application that uses a fixed number of schemas, keys can be generated (or even pre-generated) only once, and then reused in all documents valid for the given schema. While in general, key generation at the schema level has to be pessimistic, our approach tries to minimize the number of generated keys. Incoming XML documents are efficiently encrypted using single-pass SAX parsing in such a way that the original structure of these documents is completely hidden. We also describe distributing to each user only keys needed for decrypting accessible nodes, and for applying the minimal number of encryption operations to an XML document required to satisfy the protection requirements of the policy.

Table of Contents

Introduction

Recent years have seen the *eXtensible Markup Language (XML)* expand beyond its origins as a standard for web document authoring to become widely embraced as a data encoding format. For example, the most recent releases of commercial database systems include utilities for exporting relational data sets into XML format to facilitate data exchange. This trend has naturally motivated researchers to investigate fine-grained (i.e., element-level) access control models for XML data sets. While early approaches employed server-side techniques such as materialized security views and query re-writing, there has been some concern as to whether such approaches can scale up to handle applications involving thousands of users and/or complex access control policies. More recently, attention has been focused on *secure publishing* solutions, in which an XML document is made accessible to all users (e.g., by hosting it on an HTTP server) through possibly insecure channels. The desired *access control policy (ACP)* is enforced by encrypting regions of the document using cryptographic keys. Each user is assigned, through a secure channel, a set of keys (a keyring) corresponding to the permissions they have been granted under the access control policy, which allows them to decrypt exactly a document fragment consisting of element nodes they have been granted access to. There are two crucial issues to note under this scenario: (1) the access control policy is only capable of enforcing read operations, as there is nothing preventing an authorized user from altering a data field once they have decrypted it and republishing a new version, and (2) the document owner (author) relinquishes control over the document once it has been published: consequently, any changes to the access control policy, or to the contents of the document can only be imposed by publishing a new version of the document.

While secure publishing approaches can overcome many of the performance limitations of earlier approaches, some care has to be taken in their implementation to ensure scalability. A naive attempt to enforce an ACP may use *super-encryption*, where elements residing in the intersection of two or more document paths in an ACP are encrypted with multiple keys, which greatly increases the number of expensive encryption and decryption operations. We use a more efficient *multi-encryption* strategy, in which every node is encrypted with a *single* key. Additionally, a naive implementation may end up assigning many keys, effectively trading the problem of view explosion for key proliferation. Key generation is expensive, and requiring each user to handle multiple keys complicates key management. Hence, one seeks to minimize the number of keys required to enforce a particular access control policy. Since nowadays many documents conform to one or more schemas, it would be more efficient to generate keys once, at the schema level, and then re-use them for multi-encrypting documents valid in those schemas.

Contributions: In this paper, we address the performance issues alluded to earlier. Our main contribution is a generalization of key generation to the *schema level*, allowing an access control policy to be defined over an entire class of XML documents specified by an XML Schema definition. Specifically, we define SRBAC, Schema-based ACP, based on our specification of grammar paths. This is a departure from previous research on grammar-level ACPs, which have mainly focused on DTDs and require separate key generation for each document valid for the grammar. Our second contribution is a specification of processing the schema S, which generates the keyring K_S, consisting of the minimum number of keys needed. In general, key generation at the schema level has to be pessimistic because it must generate enough keys to correctly encrypt all possible documents valid in the schema. However, because these keys are generated only once, before any document is to be encrypted, the algorithms to determine the minimum number of keys necessary may afford less efficient processing. (This phase roughly resembles program optimization performed by the compiler before the program is executed). Our third contribution is a specification of a *role-available keyring* $K_S(R)$ which, for a given role R in the SRBAC policy, consists of keys that will be made available to the user in role R so that this user can access precisely the fragment of any document D, valid in S, allowed by this policy. Our fourth contribution is an *efficient multi-encryption* of XML documents in a *single pass* (using a SAX parser) in such a way that the decryption of multi-encrypted documents results in a well-formed XML document. The encrypted document hides all of the structure of the original document. While the decrypted documents are no longer valid in the original schema, one may create *sanitized* schema (for which such document will be valid), and use them to efficiently apply XML tools, such as XQuery for XML databases. Our current research concentrates on static documents, but using SAX parsing may allow considering streamed XML documents (which will be a focus of future research). Our fifth contribution is a comparison of the efficiency of our approach and super-encryption. Our last contribution is that our key generation technique applies to *any* access control model, not only to the role-based model considered in this paper. Note that while encryption is expensive, comparatively speaking, validation is not that expensive and we can't know if a document is valid until it has been completely traversed, so we may perform several costly and wasteful encryptions before discovering that the document is invalid. In such a case, combining validation and encryption only serves to slow down the validation, while performing two separate passes likely wouldn't take much longer and would avoid wasted encryptions. Therefore, we first validate the document and then encrypt it.

There are various areas where the technique described in this paper can be applied. Besides secure publishing mentioned earlier, it can also be used for areas such as providing secure data to medical organizations or social networks.

This paper is organized as follows. In Section 2, we describe related work. Section 3 describes RBAC at the document level, (DRBAC), and RBAC at the schema level, (SRBAC). In Section 4, we present our main results concerning SRBAC and key generation at the schema level, and briefly describe the implementation. Finally, in Section 5, we provide results of testing, and in Section 6 we describe conclusions and our future work.

Related Work

The increasing popularity of XML as a data encoding format has forced attention on the problem of controlling access to XML documents [De Capitani2003]. The majority of approaches to date (e.g. [Bertino2004], [Müldner2006], [Miklau2003] and [Kudo2000]) have focused on specification of access control policies at the document level, while a smaller number of proposals have additionally considered expression of policies over an entire class of documents at the schema level. [Bertino2002] and [Damiani2005] , [Damiani2002] specify systems in which discretionary access control policies can be expressed over an individual XML document, or over a document type definition (DTD) defining a class of XML documents. [Kuper2005] defines a secure querying method for annotating a DTD with security policies, from which a security view is generated. This view consists of a sanitized DTD Dv, which contains only the accessible portions of the original DTD D, and a function f, which specifies how to extract accessible data from instance XML documents conforming to D, in such a way that each query result generates a document conforming to Dv. By hiding f from clients, the schema and data in the original document remain hidden, while clients are able to issue queries on accessible regions using Dv.

More recent work focuses on the use of XML Schema as the schema specification language, in place of DTDs. [Fundulaki2004] describes a system which allows role-based access control policies to be defined at either the schema or document level. At the schema level, access permissions are defined on schema objects (definitions of elements). [Ramaswamy2003] describes RBAC-based policy specifications at the schema level, augmented with policy constraints using Schematron. Note that in all of these works, access control is enforced entirely by a server component, with a customized view of a document, consisting of only the accessible portions of the requested document, being returned to each client in accordance with their access permissions on the original document. As a customized view must be generated in response to each client access request, in scenarios with complex documents and/or several users, view explosion becomes a problem.

In contrast, our strategy forgoes reliance on a centralized access control and instead publishes a single (partially-) encrypted copy of each XML document, and relies on public-key cryptography to enforce the desired access control permission. In this sense, it resembles the work of [Miklau2003] and [Müldner2006]. In addition to avoiding the view explosion problem (alluded to earlier) by publishing a single copy of each document, this type of approach is also more suited to deployment within decentralized architectures such as peer-to-peer, as it (1) facilitates the distribution of authentication and access control responsibilities between participating systems, rather than forcing reliance on a centralized component to enforce access control, and (2) is receptive to a variety of document distribution strategies, including broadcast, rather than being restricted to the client pull model. [Bertino2001] and [Baldano2002] describe how to generate keys for documents and for DTDs/Schema; however, since at the schema level some conditions cannot be evaluated, keys must be generated at both the schema and document levels.

In this paper, we present key generation at the schema level based on the concept of symbolic grammar paths. In [Müldner2008], we describe the design of parameterized role access control policies, which has recently been implemented.

Access Control Policies

This section starts with a definition of ACPs at the document level, and then introduces ACP at the schema level. It also gives several examples that will be used throughout the paper.

Documents and Document Paths

In our approach, document fragments, or views, are described using document paths based on XPath [XPath2008]. For a document D, by P_D we denote the set of paths in D. We allow disjunctions of document paths, using the | operator. To simplify the process of defining grammar path, we borrow the following two Unix/C conventions:

- parameterless macros may be declared using the #define command, and then called using $macro-name
- substitutions of the form

  $string_0\{string_1, string_2,..., string_n\}string$

  denote

  $string_0string_1string \mid string_0string_2string \mid ... \mid string_0string_nstring$

  where | denotes the alternative. Nested {} are also supported.

For example

```
        #define HP      /H/P
        $HP/{a, b, c/{d,e}}
```

stands for

```
      /H/P/a | /H/P/b | /H/P/c/d | /H/P/c/e
```

Example 3.1.

Consider a hospital, where there are three types of tests: basic, confidential and very confidential; and each patient has three attributes: name, Id, and perm (the last attribute determines whether or not patients with negative Ids will have access to very confidential tests). The path P:

```
        #define hp      /hospital/patient
        P: $hp{/@Id, /basic/text(), /basic, [@Id<"0" AND @perm="true"]/veryConfidential/text()}
```

is a document path for the following document D:

```
<?xml version="1.0" encoding="UTF-4"?>
<hospital>
        <patient  name="Kay" Id="-1" perm="true" >
                <basic>B1</basic>
                <confidential>C1</confidential>
                <veryConfidential>V1</veryConfidential>
        </patient>
        <patient  name="Smith" Id="-2" perm="false" >
                <basic>B2</basic>
                <confidential>C2</confidential>
                <veryConfidential>V2</veryConfidential>
        </patient>
        <patient  name="Zen" Id="200" perm="true" >
                <basic>B3</basic>
                <confidential>C3</confidential>
                <veryConfidential>V3</veryConfidential>
        </patient>
</hospital>
```

The above path P defines a view of D consisting of values of Id for all patients (both attribute and its value), the tagname <basic> and its text values, B1, B2 and B3; and the value of V1. □

Document-level Access Control Policies, DRBAC

In this section we define document-level RBAC, DRBAC and the related protection requirement. Here, we use a subset of XPath; a path expression p consists of one or more location steps; each location step operates on the child axis. A path is then of the form p | not p | p1 intersect p2 | p1 union p2 , and its evaluation produces an ordered sequence of nodes. A valid path is any path which does not evaluate to an empty sequence.

Definition 3.1.

For an XML document D and a finite set Ψ of roles, the document-level RBAC (DRBAC) is a mapping π_D:$\Psi \rightarrow P_D$ such that $\pi_D(\Psi)$ covers the set D; i.e., each element of D belongs to at least one document path that occurs in the policy. Often, the mapping π_D is tabulated and shown as a tuple [(R_1, P_1),(R_2, P_2),...,(R_n, P_n)] □

There are various *views* of the document D, and each view is defined by π_D(R). The designer of the policy for an XML document D may decide to leave some parts of D unencrypted (accessible to all users) or to make them inaccessible to all users (i.e., to encrypt them, but not to provide the key used for encryption of these nodes to any user). For the former case, the symbol ❙ is used, while for the latter case we use the symbol •. Therefore, the actual definition of the document-level RBAC is that it is pair (π_D, ┈), where ┈ is either ❙ or •. For simplicity (unless specified otherwise), in the sequel, we omit the second element of this pair, and assume that by default it is always • (i.e. by default, elements of D not covered by D are inaccessible to all users).

Definition 3.2.

The document-level protection requirement is said to be satisfied under the following conditions. For an XML document D, a finite set of roles Ψ, and the document-level RBAC π_D:$\Psi \rightarrow P_D$ a user in role R can access precisely the set π_D(R), and if the policy uses ❙ also those nodes in D which are not covered by any path. □

Example 3.2.

Consider the document D from [Example 3.1], and a set of roles Ψ = {Nurse, Physician, Resident, Smith}. We define four document paths:

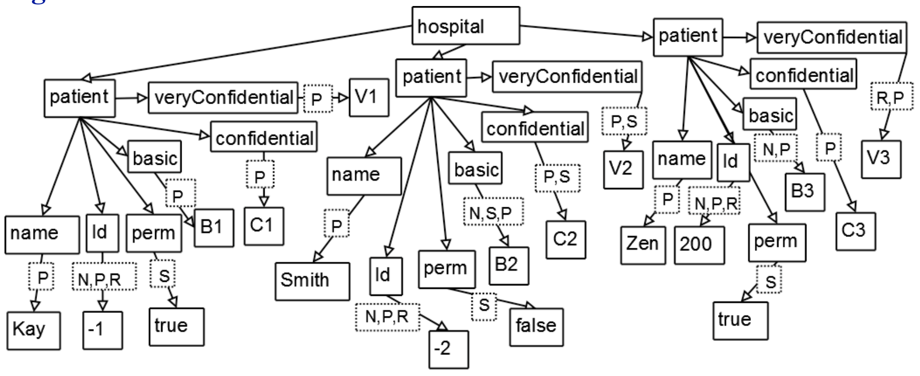
```
        #define hp      /hospital/patient
        #define Sname   $hp[@name="Smith"]
        ND: $hp{/@Id, [@Id<0]/basic/text()}
        PD: $hp/{@Id, @name, basic/text(), confidential/text(), veryConfidential/text()}
        RD: $hp{/@Id, [@Id>"100" AND @perm= "true"]/veryConfidential/text()}
        SD: $Sname/{@perm, basic/text(), confidential/text(), veryConfidential/text() }
```

We now define π_D as: [(Nurse, ND), (Physician, PD), (Resident, RD), (Smith, SD)], which gives:

- the user in role Nurse access to Ids of all patients, and to the textual content of the basic test for patients whose Id is negative;
- the user in role Physician access to Ids and names of all patients, and to the textual content of the basic, confidential and very confidential tests;
- the user in role Resident access to Ids of all patients, and to the textual content of the very confidential test for patients whose Id is greater than 100 and perm is set to true;
- the user in role Smith access to permissions, and the textual content of the basic, confidential and very confidential test for these patients whose name is Smith

Since by default we use the role •, other elements of D are available to nobody. In Figure 1, B1, C1, etc. stand for the text contents of the basic tag, confidential tag, etc.; elements accessible to the user in various roles are labeled with dotted boxes containing first letters of the names of these roles; nodes which are not available to any user are not labeled.

Figure 1.



Labeled document tree for the hospital

Let us now use [Example 3.2] to compare super-encryption and multi-encryption. In the former case, nodes which are on the intersection of document paths would have to be encrypted and then decrypted with several keys. In the latter case, each node will be encrypted and decrypted with a single key. (In Section 6, we provide results of experiments comparing these two techniques.)

Schemas and Grammar Paths

In this paper, we consider schema-oriented XML documents which have to follow the grammar rules specified by an associated schema, rather than schema-less documents which merely have to be well-formed according to the XML language specification [XML2008]. A commonly used grammar description is XML Schema [XML-Schema2008], which we also use in this paper. Throughout this paper, by L_Σ we denote the language generated by the schema Σ. Specifically, if Σ is the schema of schemas, i.e. the grammar defining schemas, then L_Σ is the language of all schemas, and for $S \in L_\Sigma$, a document D is valid for S iff $D \in L_S$. We consider only schemas for which there are no anonymous types (we can do it without a loss of generality because a schema with anonymous types can always have these types replaced by named types.) In the current implementation, we consider only schema with no cycles. By the top-level typename of the schema S, we mean the typename of the root element of any document valid in S. With each schema, we will associate a schema graph. The following examples show two schema and the corresponding graphs.

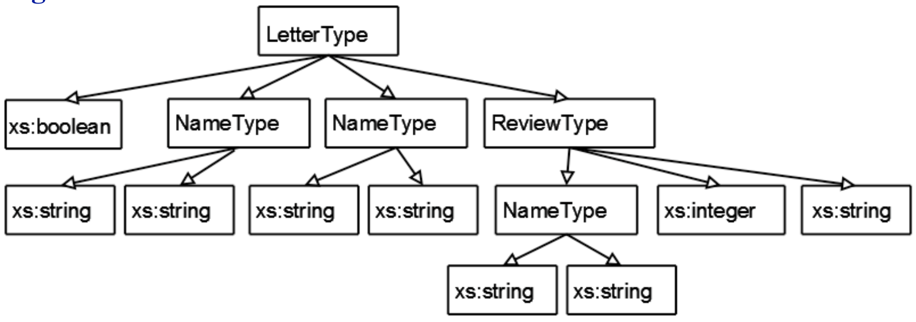
Example 3.3.

Consider a reference letter, which provides a specification as to whether or not the letter is confidential, names of a referee and an applicant, and zero, one or more reviews, where each review provides the name of a supervisor, a numeric score, and a comment. The schema T for such a reference letter is given below.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="letter" type="LetterType"/>
        <xs:complexType name="LetterType">
                <xs:sequence>
                        <xs:element name="referee" type="NameType"/>
                        <xs:element name="applicant" type="NameType"/>
                        <xs:element name="review" type="ReviewType"/>
                </xs:sequence>
                <xs:attribute name="confidential" type="xs:boolean"/>
        </xs:complexType>
        <xs:complexType name="ReviewType" maxOccurs="unbounded">
                <xs:sequence>
                        <xs:element name="supervisorName" type="NameType"/>
                </xs:sequence>
                <xs:attribute name="score" type="xs:integer"/>
                <xs:attribute name="comments" type="xs:string"/>
        </xs:complexType>
        <xs:complexType name="NameType">
                <xs:sequence>
                        <xs:element name="first" type="xs:string"/>
                        <xs:element name="last" type="xs:string"/>
                </xs:sequence>
        </xs:complexType>
</xs:schema>
```

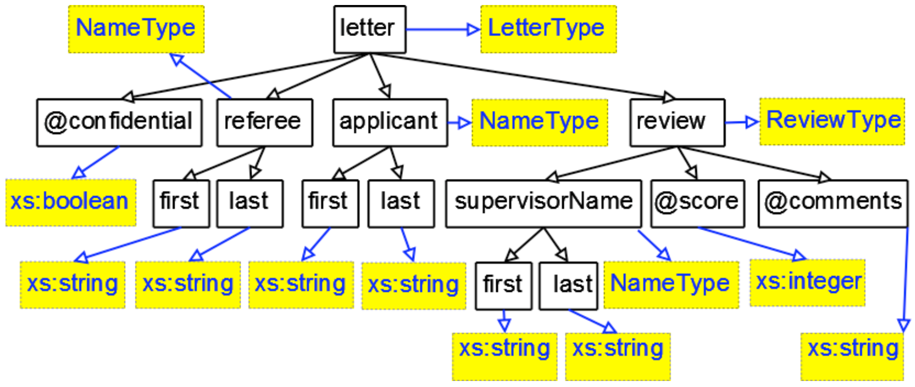
Schema graph for the Schema T (see Figure 2) replicates nodes that represent elements of the same type. This graph shows *types* used in the schema. The alternative representation, used by this paper, (see Figure 3) will focus on *element names* from the schema, and will *optionally* show types of these elements (shown in dotted boxes with gray background).

Figure 2.



Schema graph for schema T (based on types)

Figure 3.



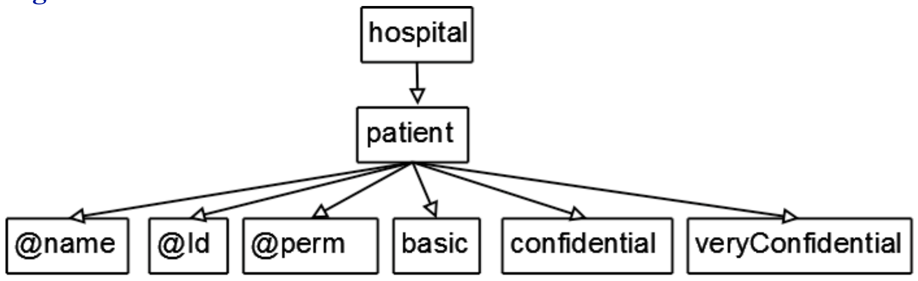
Schema graph for schema T (based on element names)

Example 3.4.

The schema S (see the schema graph in Figure 4) is a schema for which the document D from [Example 3.1] valid.

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <xs:element name="hospital" type="HospitalType"/>
        <xs:complexType name="HospitalType">
                <xs:sequence>
                        <xs:element name="patient" type="PatientType" maxOccurs="unbounded"/>
                </xs:sequence>
        </xs:complexType>
        <xs:complexType name="PatientType">
                <xs:sequence>
                        <xs:element name="basic" type="xs:string"/>
                        <xs:element name="confidential" type="xs:string"/>
                        <xs:element name="veryConfidential" type="xs:string"/>
                </xs:sequence>
                <xs:attribute name="name" type="xs:string"/>
                <xs:attribute name="Id" type="xs:int"/>
                <xs:attribute name="perm" type="xs:boolean"/>
        </xs:complexType>
</xs:schema>
```

Figure 4.



Schema graph for schema S

In Section 3.1, we defined views of XML documents using document paths. To define views of a schema S, we introduce *grammar paths*. For a document D valid in S, a grammar path may be *materialized* to produce the document path for D. This way, views of a schema determine views of documents valid in the schema. Here, we consider only absolute paths, but relative paths can be defined in the same way. We denote by Π_S the set of all grammar paths for the schema S. A triple (schema S, grammar path G, document D valid in S) defines a document path denoted by $\Pi_{S,D}(G)$; we refer to this process as a grammar path materialization.

We start by defining an *atomic grammar path*, which is of the form $/c_1/c_2/.../c_n$. Each c_i is a tag name optionally followed by a predicate in square brackets. (If the predicate is present, then c_i is called the current context node.) A *predicate* is formed using standard Boolean operators: &&, ||, and ! (using standard associativity and precedence rules) and atomic predicates. There are three kinds of atomic predicates:

- equality and inequality comparisons for values of attributes of type string or Boolean
- equality and inequality comparisons for the textual content of the current context node, specified using text()
- arbitrary comparisons for values of numerical attributes

Note that if content of the element is *mixed*, i.e. contains both textual content and sub-elements then text() denotes the concatenation of all textual contents of this element.

In order to support nested conditions, atomic predicates may also be preceded by one of the following unconditional paths, where the i-th component b_i is a tag name:

- $/b_1/b_2/.../b_n/$; i.e. the absolute path
- $b_1/b_2/.../b_n/$; i.e. the relative path that starts from the current context node
- $../b_1/b_2/.../b_n/$; i.e. the relative path that starts from the parent of the current context node

Two examples of atomic grammar paths are:

```
        /H[@x > "1" || @x < "0" &&  @y = "0"]/P[text() = "John"]
        /H/B[../C/@z > "2" || /H/D/text() = "Mary"]/E
```

Atomic paths may have conditions involving nested paths, but they specify only a single node. Often it is convenient to specify subtrees, or parts thereof. Therefore, we define a simple grammar path, which is an atomic grammar path followed by a subtree operator of the form:

```
      <text = O3; att = O3; tag = O3>
```

where

- all fields (text, att, and tag) may appear in any order (and are separated by ";")
- fields are optional, but at least one must be present
- O3 is described below by the following regular expression
  O1=["*" "+"]; i.e. a character class consisting of * and +
  O2 = (O1 | ".") | ("text" (O1?)); i.e. O1 or "." or "text" optionally followed by O1
  O3 = O2 (","O2)*; ie. one or more O2, separated by";"

The meaning of subtree operators is explained in [Example 3.5]. Finally, a *grammar path* is an atomic grammar path, or a simple grammar path, or one or more alternatives of such paths.

For a given schema, a grammar path is valid if it satisfies constraints imposed by the Schema, for example that an element name exists (in the given context), or the attribute value is a correct data type.

Example 3.5.

Below, we use the following terminology:

- all descendants of c, meaning all descendants of c, including c;
- all proper descendants of c, meaning all descendants of c, excluding c;
- children of c, meaning only children of c and not grand-children, etc.

In all examples given below c is the last component of a simple grammar path:

```
  All text descendants of c:                                c<text=*>
  All proper text descendants of c:                         c<text=+>
  Text child of c (concatenated for mixed content):         c<text=.>
  All text descendants of c that are equal to "the":        c<text="the"*>
  All proper text descendants of c that are equal to "the":     c<text="the"+>
  Concatenated text children of c that is equal to "the":   c<text="the">
  All attribute descendants of c:                           c<att=*>
  All proper attribute descendants of c:                    c<att=+>
  All attribute children (attributes) of c:                 c<at =.>
  A specific attribute foo of c:                            c<att="foo">
  All "foo" attribute descendants of c:                     c<att="foo"*>
  All proper "foo" attribute descendants of c:              c<att="foo"+>
  The tag of c:                                             c<tag="."> or c
  All element descendants of c:                             c<tag=*>
  All proper element descendants of c:                      c<tag=+>
  All element children (sub-elements) of c:                 c<tag=.>
  A specific tag foo of c:                                  c<tag = "foo">
  All element descendants called "foo" of c:                c<tag="foo"*>
  All proper element descendants called "foo" of c:         c<tag="foo"+>
  All descendants of c:                                     c<*>
  All proper descendants f c                                c<+>
  All children of c                                         c<.>
Finally, some examples of combined fields:
  All proper text descendants of c and all attributes of c:     c<text=*; att =*>
  Attributes foo and goo of c and the tag of c:             c<att = "foo", "goo"; tag =.>
  All element descendants "foo" and "goo" of c, all "hoo" attribute decendants, and the text of c:
   c<tag="foo"*, "goo"*; att="hoo"*; text=.> □
```

Example 3.6.

Below, we provide three grammar paths R, C, and N for the schema T from [Example 3.3]:

```
        R: /letter[review/score/text() > "7"]/referee/last<text=.>
        C: /letter[!(/referee/last/text() = "Smith" ||
           /referee/last/text() = "Kerry")]<att="comments", "score">
        N: /letter[@confidential = "false"]/{referee, applicant, review/supervisor}<text=+> |
          /letter[@confidential = "true" && review/score/text() > "5"
             || applicant/last/text() = "Brown"]/review/supervisor<text=+>
```

□

Schema-level Access Control Policies, SRBAC

In this section we define schema-level RBAC, SRBAC and the related protection requirement.

Definition 3.3.

For a schema S, and a set Ψ of roles, an SRBAC policy is a mapping $\pi_S:\Psi\rightarrow\Pi_S$.□

Example 3.7.

For the schema T from [Example 3.3] we define the SRBAC policy π_T: [(REFEREE, R), (COMMENT, C), (NAME, N)], where R, C and N are grammar paths from [Example 3.6]. Here:

- role REFEREE gives access to last names of referees for reviews with the score greater than 7
- role COMMENT gives access to all comments and scores, provided that the referee last name is neither Smith nor Kerry
- role NAME gives access to all names for non-confidential letters and for supervisor names provided that the letter is confidential and the score is greater than 5, or last name of the applicant is Brown

Example 3.8.

For the schema S from [Example 3.4], we define the SRBAC policy π_S: [(Nurse, N), (Physician, P), (Resident, R), (Smith, S)], where roles N, P, R, and S are defined as follows:

```
        #define HP      /hospital/patient
        #define HPN     $HP<att="Id">
        #define SS      $HP[@name="Smith"]
        N: $HPN | $HP/[@Id<"0"]/basic<text=.>
        P: /hospital/patient<att="Id","name"; text=*>
        R: $HPN | $HP{/confidential<text=.>, [@Id>"100" &&
                @perm="true"]/veryConfidential<text=.> }
        S: $SS/{@perm, basic<text=.>, confidential<text=.>, veryConfidential<text=.>}
```

The above SRBAC policy gives the patient Smith access to her data. Existence of multiple patients who should have this kind of access will result in role proliferation. A solution to this problem is to use parameterized roles, and this is the subject of our current research, see [Müldner2008].

For every document valid in the schema S, a schema-based ACP defines document-based ACP for this document

Definition 3.4.

For an XML Schema S, instance document $D\in L_S$, and the schema-based policy $\pi_S:\Psi\rightarrow\Pi_S$, we define the schema-induced document policy $\pi_{S,D}:\Psi\rightarrow P_D$, by materializing grammar paths; i.e. $\pi_{S,D}(R)=\Pi_{S,D}(\pi_S(R))$, for $R\in\Psi$.□

Now, we formulate the schema-level protection requirement in a manner similar to that from Def. 3.2:

Definition 3.5.

The *schema-level protection requirement* is said to be satisfied under the following conditions. For the schema S, the schema-based ACP $\pi_S:\Psi\rightarrow\Pi_S$ and role $R\in\Psi$, the user in a role R accessing a document $D\in L_S$, can read precisely the view $\pi_{S,D}(R)$. □

Example 3.9.

For any document D valid in the schema S from [Example 3.4], grammar paths from [Example 3.8] will be materialized as follows:

- $\Pi_{S,D}(N)$ consists of Ids of all patients, and the textual content of the basic test for patients whose Id is negative

- $\Pi_{S,D}$(P)consists of Ids and names of all patients, and the textual content of the basic, confidential and very confidential tests
- $\Pi_{S,D}$(R) consists of Ids of all patients, and the textual content of the very confidential test for patients whose Id is greater than 100 and perm is set to true
- $\Pi_{S,D}$(S) gives access to permissions, and the textual content of the basic, confidential and very confidential test for these patients whose name is Smith.

For the specific document D from [Example 3.1], these materializations will produce respectively paths ND, PD, RD and SD from [Example 3.2], and so for the policy π_S from [Example 3.4], the induced policy $\pi_{S,D}$ is the policy π_D from [Example 3.2]. □

Key Generation and Document Encryption

In this section, we describe key generation at the schema level, and also explain how these keys are used to encrypt documents valid in this schema. As we indicated in the introduction, actions that result in key generation at the schema level may afford less efficient processing because they are performed only once, before any document is to be encrypted. In addition, a typical schema definition (and therefore, its schema graph) is quite small; hence, we assume that that the schema graph is stored as a DOM tree in main memory. However, for multi-encrypting documents (which may be very large), we will only consider single pass, SAX-based parsing. In our approach, we generate keys at the schema level. However, one can also use a *lazy key generation*: instead of pre-generating keys at the schema level, key generation will be triggered by the first document to be multi-encrypted. This would generate only the schema-based keys required for multi-encrypting this specific document (and they will be properly recorded). When the subsequent document is to be multi-encrypted, first we would determine if all necessary keys (at the schema level) have already been generated, and only generate any missing keys (at again recorded).

We start this section with basic definitions. Then, we provide our most important result, which is a schema-based key generation algorithm, and we define schema-based role-accessible keyrings. Next, we describe an annotated schema and proceed to the second most important algorithm which uses an annotated schema to efficiently multi-encrypt a document valid in this schema. Our description will frequently refer to the schema S from [Example 3.4] (which we will call a "running example"), its graph (see Figure 4), and grammar paths from [Example 3.8].

Introduction

Introduction

Encryption and Keyrings

There are two kinds of encryption: symmetric encryption, which uses a shared secret key, and asymmetric encryption, which uses a public-private key pair. For asymmetric encryption, we will call the public key part a *decrypting key*, and the private key part an *encrypting key*. If it does not lead to confusion, then by a key we mean a decrypting key in an asymmetric encryption and a key in a symmetric encryption. In our approach, we allow either kind of encryption. When asymmetric encryption is used, the user in role R can only decrypt the document fragment. But, with symmetric encryption, they can also encrypt the decrypted m-document. To avoid this problem, the digital signature of the m-document would have to be made available.

Definition 4.1.

A keyring K is a finite set of keys. For an asymmetric encryption, each key is a triple (key name, encrypting key, decrypting key), and for a symmetric encryption it is a pair (key name, key). By K_S we denote a schema-level keyring for the schema S and its policy π_S. □

Basic Notations

Let us first consider various ways keys may be assigned to documents. Since views may be overlapping, we can not assign keys per view. Indeed, if we did so, then for two overlapping views V_1 and V_2 we would have two corresponding keys k_1 and k_2, and the intersection of the two views would have to be super-encrypted (assigning keys per view would mean that for i=1,2, the user who has access to the view V_i would be given a key k_i and so other options such as encrypting V_1 with one key and V_2-V_1 with the other key would provide access to the part that is not allowed). What we need to do is to partition the set D (and at the same time each view) into disjoint sets, and then assign one key for each set in this partition. Therefore, for an arbitrary partition $\{P_1, P_2,...,P_n\}$ of a set D, we need to find a *disjoint partition*. One way to achieve this goal is to create all possible differences of intersections of sets in the partition and remaining unions. Note that in our paper by a *partition* we mean a family of subsets of a set D that does not necessarily cover D.

Here, we work at the schema level, rather than at the document level, but we need to generate keys in such a way that *materialized grammar paths* are disjoint. Since set operations for grammar paths (such as a union) are not defined, we now introduce symbolic grammar paths that allow such operations:

Definition 4.2.

Consider a set $\{P_i$: i=1,2,...,k$\}$ of grammar paths. By a *symbolic grammar path expression* we denote a finite set of string of the form:

Figure 5.

$$\prod_{i\in\{i_1,i_2,\dots,i_m\}} P_i \ - \left(\sum_{j\in\{j_1,j_2,\dots,j_n\}} P_j \right)$$

where $\{i_1,i_2,\dots,i_m\}$ and $\{j_1,j_2,\dots,j_n\}$ are finite subsets of the set $\{1,2,\dots,k\}$, $\{j_1,j_2,\dots,j_n\}=\{1,2,\dots,k\}$ - $\{i_1,i_2,\dots,i_m\}$, $\prod_{i\in\{i_1,i_2,\dots,i_m\}} P_i$ stands for $P_{i_1}*P_{i_2}*\dots*P_{i_m}$ and $\sum_{j\in\{j_1,j_2,\dots,j_n\}} P_j$ stands for $P_{j_1}+P_{j_2}+\dots+P_{j_n}$. □

Since document paths support the above mentioned set operations, symbolic grammar paths can be materialized.

Definition 4.3.

Figure 6.

$$\prod_{i\in\{i_1,i_2,\dots,i_m\}} P_i \ - \left(\sum_{j\in\{j_1,j_2,\dots,j_n\}} P_j \right)$$

where $\{i_1,i_2,\dots,i_m\}$ and $\{j_1,j_2,\dots,j_n\}$ are finite subsets of the set $\{1,2,\dots,k\}$, $\{j_1,j_2,\dots,j_n\}=\{1,2,\dots,k\}$ - $\{i_1,i_2,\dots,i_m\}$, $\prod_{i\in\{i_1,i_2,\dots,i_m\}} P_i$ stands for $P_{i_1}*P_{i_2}*\dots*P_{i_m}$ and $\sum_{j\in\{j_1,j_2,\dots,j_n\}} P_j$ stands for $P_{j_1}+P_{j_2}+\dots+P_{j_n}$. □

□

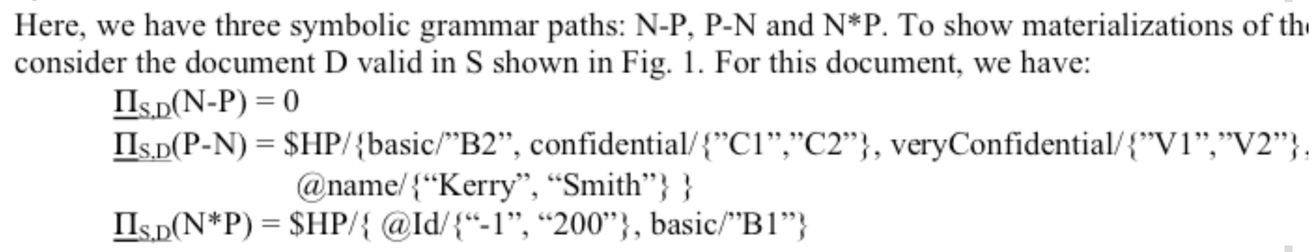
Example 4.1.

Consider first two grammar paths from [Example 3.8], and recalled below:

```
        #define HP      /hospital/patient
        #define HPN     $HP<att="Id">
        N: $HPN | $HP/[@Id<"0"]/basic<text=.>
        P: /hospital/patient<att="Id","name"; text=*>
```

Here, we have three symbolic grammar paths: N-P, P-N and N*P. To show materializations of the paths, consider the document D valid in S (see Figure 1. For this document, we have:

Figure 7.



Here, we have three symbolic grammar paths: N-P, P-N and N*P. To show materializations of th
consider the document D valid in S shown in Fig. 1. For this document, we have:

$\underline{\Pi_{S,D}}$(N-P) = 0

$\underline{\Pi_{S,D}}$(P-N) = \$HP/{basic/"B2", confidential/{"C1","C2"}, veryConfidential/{"V1","V2"},
@name/{"Kerry", "Smith"} }

$\underline{\Pi_{S,D}}$(N*P) = \$HP/{ @Id/{"-1", "200"}, basic/"B1"}

Note that the above materializations cover the same subset of D as materializations of original grammar paths, but unlike the original grammar paths, they are disjoint. □

To generate keys for the schema S one can use the following technique. For every symbolic grammar path, generate a key for that path, and then use these keys to multi-encrypt materializations of document valid in S. However, by following this approach, we would not avoid generating keys for symbolic grammar paths that will always be materialized to *empty document paths*. Therefore, we use a different approach, described below.

A *target* of a grammar path G is a single node in SG, or a node and a subgraph rooted at the node. An *abstract value* of G is a list of targets with the associated information about values of conditions that appear in G. Our algorithm finds abstract values of symbolic grammar paths corresponding to grammar paths in the SRBAC policy. We then use these values to avoid generating keys for these symbolic grammar paths whose abstract values are empty. Note that our algorithm additionally avoids creating redundant keys for two or more grammar paths whose abstract values are subsets; for example if the abstract value of G_1 is a subset of the abstract value of G_2 then the symbolic grammar path G_1 - G_2 evaluates to empty. Since at the schema level we *do not know values of conditions* that appear in grammar paths, we proceed as follows. By a *configuration* we refer to a bit-vector that represents true/false values of all conditions that appear in symbolic grammar paths (i.e., if the i-th condition evaluates to true, the i-th bit is set to 1, and 0 otherwise). We annotate every node of the schema graph with the list of pairs of the form (configurations, key).

Configurations

A *configuration* is a binary value of size k that represents true/false values of conditions. For a given grammar path X and a configuration C, values of all conditions in X are known, and therefore we can find an abstract value of X, which consists of one or more targets for the configuration C. A *configuration index value* is the integer decimal value corresponding to the binary value of this configuration. When it does not lead to confusion, we identify a configuration and its index value. Some configurations should be *excluded* e.g. if they specify true values for two conditions (involving the same attribute) that cannot be true at the same time (and arbitrary values for other conditions).

Algorithm 4.1

A *condition* (with k variables) is a Boolean Expression $B(x_1,x_2,...,x_k)$ built from comparisons of the form:

x_i RelationalOperator c; where x_i and a constant c are of the same data type that supports linear order (such as real or integer) and three standard Boolean operators: and, or and not, with the standard associativity and precedence rules, and brackets, and a RelationalOperator is one of: >, <, <=, >=, =, !=. For example, the expression B(x,y): x>"0" && y>"100" || x>="4" && !(x="99") is a condition.

For given conditions $B_1,...,B_k$ with k variables $x_1,x_2,...,x_k$, Algorithm 4.1 *tests* if for any subset $(i_1,...,i_m)$ with 0<=m<=k of the set (1,...,n) the expression of the form $D(x_1,x_2,...,x_k) = B_{i_1}\&\&B_{i_2}\&\&...\&\&B_{i_m}\&\&!B_{j_1}\&\&!B_{j_2}\&\&...\&\&!B_{j_r}$ where $\{j_1,j_2,...,j_r\} = \{1,2,...,k\} - \{i_1,i_2,...,i_m\}$, is a *falsehood*; i.e. is false for all variables $x_1,x_2,...,x_k$.

Finding abstract values

An *abstract value* of a grammar path G is defined to be a list consisting of the pairs (target, list of configuration ids). If this list contains all non-excluded configurations, then it is omitted.

Algorithm 4.2.

Input: A schema S, and an SRBAC policy $\pi_S:\Psi\rightarrow\Pi_S$, where $\Pi_S =\{G_i: i=1,2,...,n\}$ is a set of grammar paths, and role R_i is associated with the path G_i.

Output: A table called Results storing abstract values of symbolic grammar paths.

Outline of the algorithm. While here we say a "table", the actual implementation may use a different, more efficient data structure. However, in our explanation of the algorithm, we will assume that the table is used. There are two steps of the algorithm:

- Create a list E of all non-excluded conditions C that appear in grammar paths in the policy, using Algorithm 4.1
- Compute abstract values of all grammar paths in the SRBAC. (In the description of an abstract value, we skip the list of configuration ids if this list contains every possible configuration.) Here, we compute symbolic products of pairs G_i*G_j and sums G_i+G_j, products and sums of triples, and so on; finally values of symbolic grammar paths. In this process, maintain three tables storing respectively products, sums and resulting grammar paths.

In more details, the table Products[i] stores an abstract value of products $G_{i_1}*...*G_{i_j}$ where $1<=i_1<=...<=i_j<=n$ and 1<=j<=k. Here, the index i is a decimal value of the binary word of size k in which bits at positions i_m are set to 1 and remaining bits are set to 0. For example, for k=4, Products[5] stores an abstract value represented by 0101, i.e. of the product G_2*G_4 and Result[5] will store the value of $G_2*G_4 - (G_1+G_3)$.

Execution of step 2 involves:

- compute (and store in arrays Products and Sums) abstract values of all G_i, then products (and sums) of two G_i. If any product is empty then store in Products the value 0 (Empty) for any i-tuple that uses this product. Also, store in Results 0 for symbolic grammar paths that use such products. For example, if G_2*G_4 is empty, store 0 in Products at positions 5, and 7 and in Results at the same positions. Note that before any products are computed, we check if the value 0 has already been stored for this product
- repeat the above step for triples, etc. In computing unions, use previously computed values; for example to compute $G_1+G_2+G_4$, we use the value of G_1+G_2 which has already been computed.

Example 4.2.

In our running example, we have four conditions:

C1: (@Id<"0"), C2: (@Id>"100"), C3: (@perm="true") and C4: (@name="Smith")

and there are 16 configurations; starting with configuration 0, for which all conditions are false, and ending with configuration 15, for which all conditions are true. Now, consider configuration (1100) with the index value equal to 12. However, C1 and C2 cannot be true at the same time, and so any configuration of the form 11xx is excluded. Therefore, the list of non-excluded configurations consists of integers from 0 to 11, and an abstract value of the grammar path N is (Id, basic/text(), (9,10,11)).

Abstract values of all symbolic grammar paths are shown below (starting with abstract values of four paths), using the following abbreviations:

- I : {1,3,5,9,11}
- B : basic/text()
- C : confidential/text()
- V : veryConfidential/text()

```
        {N} = <Id, (B,8,9,10,11)>
        {P} = <name,Id,B,C,V>
        {R} = <Id, C, (V,6,7)>
        {S} = <(B,I), (V,I), (C,I), (Perm,I)>

        Results[1] = S-(P+R+N) = (perm,I)
        Results[2] = R-(N+P+S) = 0
        Results[3] = R*S-(P+N) = 0
        Results[4] = P-(N+R+S) = <name,(B,0,2,4,6),(V,0,2,4,8,10)>
        Results[5] = P*S-(N+R) = <(B,1,3,5,7),(V,I)>
        Results[6] = P*R-(N+S) = <(C,0,2,4,6,8,10),(V,6)>
        Results[7] = P*R*S-N = <(C,I), (V,7)>
        Results[8] = N-(P+R+S) = 0
        Results[9] = N*S-(P+R) = 0
        Results[10] = N*R-(P+S) = 0
        Results[11] = N*P*R-S = 0
        Results[12] = N*P-(R+S) = <B,8,10>
        Results[13] = N*P*S-R = <B,9,11>
        Results[14] = N*P*R-S= <Id>
        Results[15] = N*P*R*S = 0
```

Schema-based Keyrings and Role-accessible Keyrings

For a schema S and a schema-level access control policy $\pi_S:\Psi\rightarrow\Pi_S$, we use Algorithm 4.1, and for each non-empty value from the table Results, we generate one key. The keyring K_S consists of all keys generated this way. Note that in our running example, out of 16 symbolic grammar paths, only 8 are non-empty. Therefore, where a brute-force approach would generate 16 keys for four roles for our running example, our optimized algorithm will instead generate only 8 keys. Now, we show how for each role $R\in\Psi$ the keyring K_S defines the set $K_S(R)$ of R-accessible keys. A user in role R will be provided with the R-accessible keys so that for any document D valid for S, she can decrypt the view $\pi_{S,D}(R)$, where $\pi_{S,D}$ is the schema-induced document policy.

Algorithm 4.3.

Input: A schema S, a schema-level access control policy $\pi_S:\Psi\rightarrow\Pi_S$ and a role $R\in\Psi$.

Output: Role R-accessible keyring $K_S(R)$.

Method: $K_S(R) = \{k \in K_S$ for some symbolic grammar path G, the abstract value of {G} is a subset of the abstract value of the grammar path $\pi_S(R)\}$ ☐

Example 4.3.

For our running example, let us denote by ri a key corresponding to the value of Results[i] which is not empty. Therefore, K_S= {r1, r4, r5, r6, r7, r12, r13, r14}. Two examples of role-accessible keyrings are K_S(Nurse) ={r12, r13, r14}, and K_S(Physician) ={r4, r5, r6, r7, r12, r13, r14}.☐

One can show that given a schema-level access control policy $\pi_S:\Psi\rightarrow\Pi_S$ and XML document D valid in S, if the user U in role R is provided only with R-accessible keys created by Algorithm 4.3 then the protection requirement is satisfied; in particular U can access precisely the view $V = \pi_{S,D}(R)$. ☐

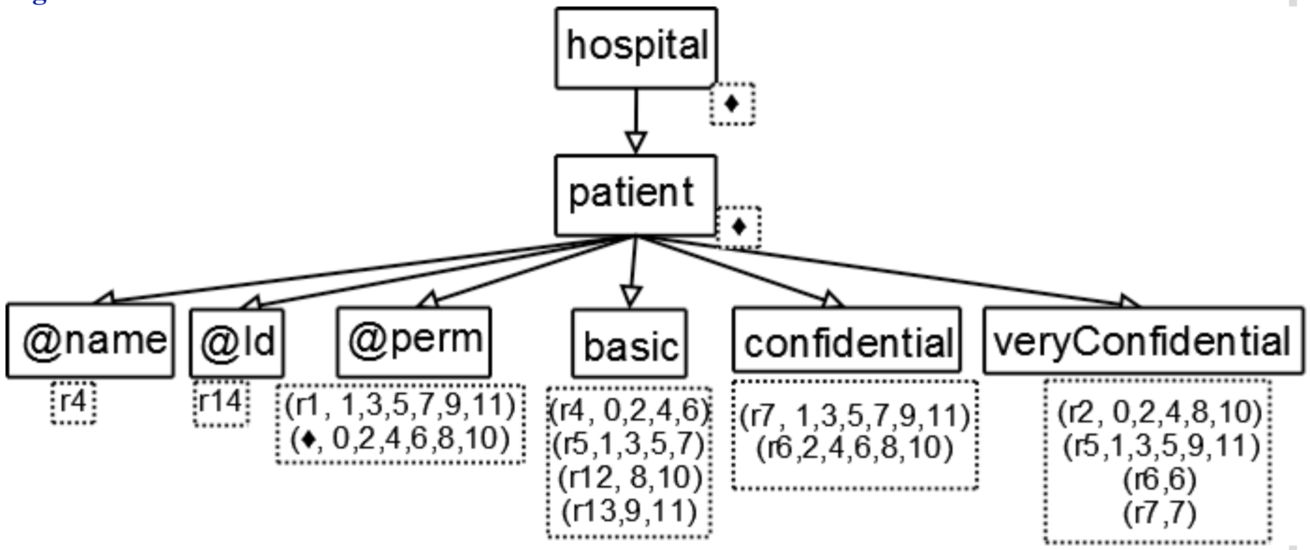
Schema Graph Annotations

Based on the information stored in the table Results, we now annotate the syntax graph with a list of pairs (key, list of integers representing configuration index values). We use ------ to denote a key for the default role; one of • or ❙ . If the list consists of *all* permissible index values, then it is omitted. The annotated schema graph will be used to guide the process of multi-encrypting an XML document D during its SAX-based parsing. To annotate the schema graph, we traverse the table Results, and collect information about all keys. Next, we traverse the schema graph again, and if the annotation is missing at any node we add the annotation of the form (------). For nodes that have already been annotated, we check if these annotations contain all permissible index values, if any such values are missing they are added with the default key ------.

Example 4.4.

Consider our running example. To annotate the node "veryConfidential" in the schema graph, we look in Results[] for the value V, and so its annotation is <(r4,0,2,4,8,10), (r5,I), (r6,6), (r7,7)>. For the complete annotated schema graph see Figure 8.

Figure 8.



Annotated syntax graph for schema S

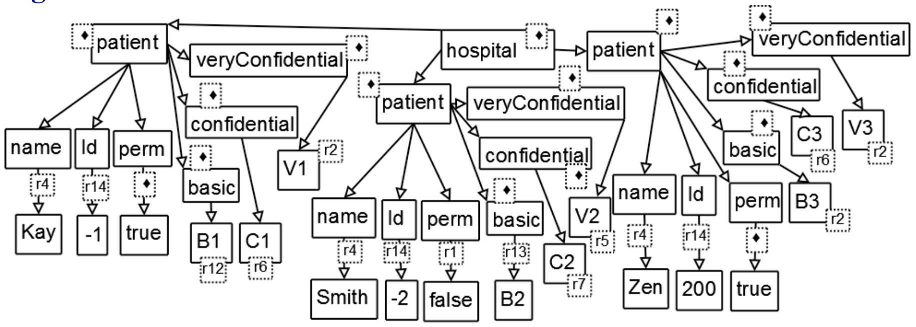
Multi-encryption of Documents

The encryption of the document D valid in the Schema S results in single well-defined XML m-document MD, which always has a root element <encrypteddocument>. Each part of the document D that is encrypted with a key "k" will be represented as a child of the <encrypteddocument> node using this format: <edata key = "k"> *encrypted_data* </edata>. Note that the structure of the document D is hidden inside of the *encrypted_data* part and is therefore not visible to users who are not authorized to decrypt the document (the <edata> tags are not nested inside of other <edata> tags). The encryption is performed by efficient, single-pass SAX parsing of D (performed in parallel with the traversal of the previously annotated schema graph GS). For each node d of D, the corresponding node s in GS is located, and its annotation is examined to determine which key should be used to encrypt this node (note that at that time we can determine values of all conditions and so we know which configuration should be used; unless nested conditions are used, which require additional treatment). Recall that the same key always encrypts the attribute and its value.

Example 4.5.

Consider the document D from [Example 3.1], (see Figure 9), in which dotted boxes attached to elements and attributes show the key that is used to encrypt this element. To explain the choice of keys, note that there are three patients; when the attributes of the first patient are processed, we know that that the name is not "Smith", Id is negative, and perm is true, and so the configuration 10 is used. Similarly, configuration 9 is used for the second patient, and configuration 6 for the third one. This information is used to retrieve the key to be used for encryption from the annotated schema graph.

Figure 9.



Document D and keys used to encrypt this document.

The m-document M_D looks like this (assuming that by default ┈ indicates that the node is not available to any user, so the corresponding <edata> nodes are omitted; below, we show details of the leftmost "patient" subtree and leave out other subtrees; encrypted data are also omitted):

```
<encrypteddocument>
        <edata key="r4">...</edata>
        <edata key="r14">...</edata>
        <edata key="r6">...</edata>
        <edata key="r2">...</edata>
        <edata key="r4">...</edata>
</encrypteddocument>
```

Decryption of Multi-Encrypted Documents

When the user U can prove that is entitled to play role R, she will first obtain the R-accessible keyring $K_S(R)$ via a secure channel, and then proceed to decrypt an m-document M_D, to produce a document $M_{D,R}$. The software system available for U will first identify a sub-keyring K1 of $K_S(R)$ consisting of keys needed to decrypt M, and then apply keys available in K1 to perform decryption. The output from the decryption is a single XML document. Given this document, U may use the system to create a sanitized schema (showing only accessible parts of schemas), which would allow them to efficiently apply XML tools, such as XQuery for XML databases. One can show that given a schema-level access control policy and XML document D valid in S, if the user U in role R is provided only with R-accessible keys $K_S(R)$, then the protection requirement is satisfied. □

Example 4.6.

Consider the m-document from [Example 4.5], using the schema-based ACP policy from [Example 3.8]. Recall from [Example 4.3] that K_S(Physician) ={r4, r5, r6, r7, r12, r13, r14}. Assuming that the user plays the role Physician, the decrypted file will look as follows (note that in this example, tags are encrypted and so their respective values have been concatenated):

```
        <encryptedtag name="Kay" Id="-1">B1C1V1</encryptedtag>
```

□

Implementation

Our implementation uses the C++ language, with external libraries to support XML usage and cryptographic functions; specifically the Xerces-C++ XML toolkit from Apache [Xerces2008], and Crypto++ [Crypto++2008]. First, the schema file is parsed and loaded into memory represented as a tree like structure (a schema graph). Once the tree is constructed, the access control policy file will be parsed and can be validated for semantic correctness against the schema in memory (although this is not currently completed). Lastly, all of the information that the encryption process requires will be saved into a file that can be loaded at a later point in time. Currently, this file is a simple xml file that contains the generated keys, schema graph and annotations, as well as the set of conditions specific in the access control policy.

The encryption process works by firstly re-loading (if necessary) the data saved to the xml file back into memory, and next the document to be encrypted is parsed with a SAX parser. Each node in the document will be encrypted in the order they appear in the document, with nodes being queued if there is not enough information available to encrypt them at the time they are first encountered (this can happen when the access control policy specified a condition for this node which cannot be calculated until a node later in the document is parsed). The encryption process is implemented in such a way that the algorithm used to perform the encryption can be changed very easily, so either a symmetric or asymmetric algorithm can be used based on user preference. It is also possible to extend the implementation to provide a run time facility for selecting an encryption algorithm.

Nested paths to arbitrary nodes in conditions are supported through a combination of data caching and a queue. During encryption, each node to be encrypted is added to the end of the *encryption queue*. Additionally, if the node is an attribute or text node that is referenced by conditions, the value of this node is cached in the *global value table*. The encryption queue is then flushed, by trying to encrypt as many nodes from the front of the queue as possible. A node can only be encrypted and removed from the queue if all of its dependencies are met. A node with no condition has no dependencies, and can be always encrypted. A node with a condition has dependencies, and the global value table is searched to ensure that all dependent attribute and text nodes have values within this table. If any dependent node is missing a value, this node cannot be encrypted, and no further nodes flushed from the queue. Otherwise, the condition can be evaluated (determining the key to use for encryption) and then the node is encrypted.

Results of Testing

We performed testing of our implementation, comparing the time required to multi-encrypt (and decrypt) against the time spent when super-encryption was used. The detailed description of steps performed in testing is provided below:

- For the schema S from [Example 3.4], we considered the SRBAC policy S from [Example 3.8] , and determined the number of keys generated by our implementation.
- For the same schema S, we considered two documents valid in this schema; document D from [Example 3.1] and document D2, which is similar to D (except it had much longer text values and different values of attributes). For both of these document, we also considred the induced DRBAC policies for these documents. .
- For each of the two documents D and D2, we multi-encrypted them four times using our approach, measured the time needed to encrypt, and recorded the average multi-encryption time
- For each of the two multi-encrypted documents M_D and M_{D2}, we decrypted themfour times for all four roles, measured the time needed to decrypt, and recorded the average decryption time
- For each of the two documents, we generated keys and super-encrypted these documents by performing the SAX traversal of the document, using the following approach:

  a) for every node x in the intersection of document paths there is a keyring consisting of m keys (one for each path); and the node x is super-encrypted with all these keys. The user in any role R associated with the path P from the intersection will receive the entire keyring.

  b) to satisfy the protection requirement, keys are reused based on the following principle. Suppose that in performing the SAX traversal of a document, we have encountered the node x as described above (and generated multiple keys), and then we have encountered the node y in the intersection of the same document paths. Then, the same keyring is used (i.e. no new keys are generated), and the node y is super-encrypted with keys from this keyring.
- For each of the two documents D1 and D2, we super-encrypted them using the approach described above, and measured the time needed to encrypt
- For each of the two super-encrypted documents M_D and M_{D2}, we decrypted them four times for all four roles, measured the time needed to decrypt, and recorded the average decryption time

Results.

The total number of keys generated at the schema level was 8, while the number of keys generated at the document level (for super-encryption) was 17 for both documents. The time needed to generate keys at the schema level was 0.0279024. The average time needed respectively for multi-encryption and super-encryption of the document D was 0.0007776830 and 0.0019404000. For the document D2, the average time was 0.0022022350 and 0.0042952375. The average time needed respectively for decryption for the role Physician for the multi-encrypted and super-encrypted document M_D was 0.00125512 and 0.00368630. For the document M_{D2}, the average time was respectively 0.00633886 and 0.00972875.

Conclusions.

Our testing proved the expected superiority of multi-encryption over super-encryption. Approximately half the keys were generated (although at the schema levels we generate keys for all valid documents). While multi-encryption incurs the initial cost of time spending on key generation, our tests show that on average for multi-encryption, the encryption time and the decryption times were between 30 and 50 percent shorter.

Conclusions and Future Work

In this paper, we provided a description of role-based ACPs for defining permissions for fragments of XML documents, both at the document and the schema level. Using our approach, keys can be generated at the schema level and then only required keys may be assigned to specific documents valid for the schema. A complete system was designed and implemented.

Our future work includes full optimization of the algorithm which generates keys at the schema level, removing some restrictions on XML documents, such as IDREF attributes, and designing and implementing re-usability of keyrings when a document or ACP is modified. Also, we will start using our approach to develop secure publishing solutions in areas such as social networks.

Acknowledgements

The algorithm to find excluded configurations was suggested by Raimond Seidel, Saarbruecken University. Greg Leighton, a PhD student from the University of Alberta provided many useful comments. Finally, the first author would like to acknowledge help of Tommie Usdin in resolving various issues when converting the document in thw Word format to XML.

Bibliography

[Baldano2002] Baldonado, M., Bertino, E. and Ferrari, E. Secure and Selective Dissemination of XML Documents. ACM Transactions on Information and System Security (TISSEC), 5(3):290—331, (2002).

[Bertino2004] Bertino, E., Carminati, B., Ferrari, E., Thuraisingham B. and A. Gupta. Selective and Authentic Third-Party Distribution of XML Documents. IEEE Transactions on Knowledge and Data Engineering (TKDE), 16(10), 2004, pp. 1263—1278.

[Bertino2001] Bertino, E., Carminati, B. and Ferrari, E. A temporal key management scheme for secure broadcasting of XML documents. Conference on Computer and Comm. Security. Proc. of the 9th ACM conference on Computer and communications security (2002): 31—40.

[Bertino2002] Bertino, E., Carminati, B. and Ferrari, E. Securing XML Documents with Author-X. IEEE Internet Computing Volume 5, Issue 3 (2001): 21 — 31.

[Cramption2004] Crampton, J. Applying hierarchical and role-based access control to XML documents. Proc. of the 2004 workshop on Secure web service (2004): 37 — 46.

[Damiani2005] Damiani, E. De Capitani di Vimercati, S.D.C. and Samarati, P. New paradigms for access control in open environments. Signal Processing and Information Technology, Proc. of the Fifth IEEE International Symposium (2005): 540—545.

[Damiani2002] Damiani, E., De Capitani di Vimercati, S., Paraboschi, S. and Samarati, P. A Fine-grained Access Control System for XML Documents. ACM Transactions on Information and System Security,5(2): 169—202,(2002).

[De Capitani2003] De Capitani di Vimercati, S., Paraboschi, S. and Samarati, P. Access control: principles and solutions. Software Practice and Experience, Vol, 33, Issue 5 (April 2003): 397—421. John Wiley and Sons, Inc

[Devanbu2001] Devanbu, P., Gertz, M., Kwong, A., Martel, C., Nuckolls, G. and S.G. Stubblebine. Flexible Authentication of XML documents. In Proc. of the 8th ACM Conference on Computer and Communications Security, ACM Press, (2001).

[Ferraiolo2001] Ferraiolo, D.F., Sandhu, R., Gavrila, S., Kuhn, D.S. and Chandramouli, R. Proposed NIST Standard for Role-Based Access Control. ACM Trans. on Information and System Security, 4 (3), (2001), 224—274.

[Fundulaki2004] Fundulaki, I. and Marx, M. Specifying access control policies for XML documents. Proceedings of the ninth ACM symposium on Access control models and technologies (2004) 61 — 69.

[Goel2003] Goel, S K., Clinton, C. and Rosenthal, A. Derived access control specification for XML. Proc. of the 2003 ACM workshop on XML security (2003): 1 — 14.

[Kudo2000] Kudo, M. and Hada S. XML document security based on provisional authorization. Proc. of the 7th ACM conference on Computer and communications security (2000): 87 —96.

[Kuper2005] Kuper, G., Massaci, F. and Rassadko, N. Generalized XML security views. Proc. of the tenth ACM symposium on Access control models and technologies. (2005):77—84.

[Miklau2003] Miklau, G. and Suciu, D. Controlling Access to Published Data Using Cryptography, In Proc. of the 29th VLDB Conference, Berlin, Germany, (2003).

[Müldner2006] Müldner, T., Leighton, G. and Miziolek, J.K. Using Multi-Encryption to Provide Secure and Controlled Access to XML Documents. Extreme Markup Languages 2006, (2006), Montreal, Canada.

[Müldner2008] Müldner, T., Leighton, G. and Miziolek, J.K. Succinct Access Control Policies for Published XML Datasets. 10th International Conference on Enterprise Information Systems. 12 —16, June 2008, Barcelona, Spain.

[Ramaswamy2003] Ramaswamy C. A Policy Validation Framework for Enterprise Authorization Specification. 19th Annual Computer Security Applications Conference ACSAC, (2003): 319—329.

[XML2008] Extensible Markup Language (XML) 1.0 (Fourth Edition) http://www.w3.org/TR/REC-xml/.

[XPath2008] XML Path Language. http://www.w3.org/TR/xpath.

[XML-Schema2008] XML Schema http://www.w3.org/TR/xmlschema-0/.

[Zhang2003] Zhang, X., Park, J. and Sandhu, R. Schema based XML Security: RBAC Approach, 17th IFIP 11.3. Working Conference on Data and Application Security, 2003.

[Xerces2008] http://xerces.apache.org/xerces-c/.

[Crypto++2008] http://www.cryptopp.com/.