Sécurisation du contrôle d'accès pour des documents XML

François Dang Ngoc

Université de Versailles Saint-Quentin-en-Yvelines

THÈSE / PhD THESIS

Pour obtenir le grade de

docteur

discipline : **Informatique**

présentée et soutenue publiquement

par

François DANG NGOC

Le 15 Février 2006

Sur le sujet

A Secure Access Controller for XML Documents

(Sécurisation du contrôle d'accès pour des documents XML)

JURY

Prof. Frédéric CUPPENS, Professeur à l'ENST Bretagne, *Président*

Prof. Elisa BERTINO, Professeur à la Purdue University, *Rapporteur*

Prof. Alban GABILLON, Professeur à l'Université de Pau, *Rapporteur*

Dr. Luc BOUGANIM, Responsable permanent du projet SMIS à l'INRIA, *Co-encadrant de thèse*

Dr. Boutheina CHETALI, Responsable Méthodes formelles & Sécurité chez Axalto, *Examinateur*

Prof. Philippe PUCHERAL, Professeur à l'Université de Versailles, *Directeur de thèse*

To my parents, my sister Trâm and my brother Frédéric

Acknowledgements

I extend my sincere gratitude and appreciation to many people who made this PhD thesis possible. The first persons I would like to thank are my supervisors Philippe Pucheral and Luc Bouganim. Philippe, your general view on research, your rigor, and your mission for providing only high-quality works have made a deep impression on me. Luc, I really appreciated your overly enthusiasm, your kind advice and all the good times working late together on research problems. I owe you two lots of gratitude for having learned so much during these three years.

I would like to thank the members of my PhD committee who monitored my work and took effort in reading and providing me with valuable comments: Professor Elisa Bertino, Doctor Luc Bouganim, Doctor Boutheina Chetali, Professor Frédéric Cuppens, Professor Alban Gabillon and Professor Philippe Pucheral.

This thesis could not have been possible without the support of my family who I owe a lot of thanks: my parents, my brother Frédéric, my sister Trâm and Rémy Card who gave me useful advice and encouragement during my PhD period.

My research started at the PRiSM Laboratory at the University of Versailles where I met many interesting and friendly people. I would like to thank more particularly Xiaohui Xue, Tao Wan and Mathieu Decore whom I have known for three years and showed to be kind, mostly helpful and trustful friends. Thank you also to Huaizhong Kou for teaching me a lot of things and for bringing a cheerful atmosphere at the lab, Lilan Wu, Véronika Péralta, Clément Jamard, Nicolas Travers for being fantastic colleagues and friends. Finally, I would like to ackowledge Michel Cavaillé who initiated me to the magic of smart card programming.

At INRIA, my colleagues of the SMIS project all gave me the feeling of being at home at work: Saïda Medjdoub, Cristian Saita, Aurélian Lavric, Nicolas Dieu, Sophie Giraud and Cosmin Cremarenco. Many thanks for being your colleague and friend and for all the parties that we had together. Special thanks to Varun Arora, Dinial Bensalah, Daniel Calegari, Cosmin Cremarenco, Nicolas Dieu, Juan Diego Ferre and Lilan Wu who worked with me on the development of prototypes. It was nice to work with you in such a friendly environment. I would also like to thank Elisabeth Baque, our project assistant who helped me many times on administrative issues and the newcomers and the interns of the SMIS project: Mehdi Benzine, François Boisson, Sonia Guehis, Christophe Salperwyck and Christophe Schmitz for all the good times and discussions at coffee breaks.

I would also like to acknowledge some people I met when I was preparing my Master thesis at the University of Paris 6: Patrick Valduriez and Hubert Naacke who introduced me to this fabulous world of research, Nicolas Lumineau for your trusted and kind friendship and all your funny stories, Cécile le Pape and Cédric Coulon for being kind colleagues.

I also say thanks to people I met during my studies: Karine Brifault, Thierry Caillet, Bénédicte Sapin, Nicolas Courtay, Nicolas Leclerc, Christian Moranne, Thierry Moutte, Jérôme Mullot, Guilhem Péretié, Fei Sha and Marc Tanguy for their kind friendship.

Last but not least, I would like to thank my dear cousin Minhhang K. Huynh for sharing her insights and for her encouragement during these three years.

François Dang Ngoc,
February 23, 2006
Rocquencourt, France

Table of Content

List of Figures

List of Tables

Chapter 1 – Introduction

Today, information is accessible anywhere, at any time and on an ever growing number of devices. This situation has improved our everyday life, bringing many benefits for people who can share and distribute information all over the world. Regarding the digital content industry, this enables to consider alternatives to traditional mode of distribution of media (e.g., CDs, DVDs) and to provide new business models (e.g., iTune, P2P). Finally, for the business companies, information can be made highly available to partners and customers.

A vast amount of information is exchanged every day ranging from simple text file to complex video files. Depending on their nature, data may be more or less confidential. In the case of medical folders, their confidentiality is of utmost importance and their disclosure may have dramatic consequences for the patients. For the digital content industry, the availability of the media has to be restricted only to registered users and failing in doing so may cause big economic losses. For a business corporation, industrial secrets can be shared only among the partners but a single leakage of information may be disastrous for the company.

1 Context of the study

Traditionally, confidential information is stored on a trusted server in charge of enforcing the sharing policies within a community of users. However, this situation is rapidly evolving due to very different factors: the suspicion about Database Service Providers (DSP) regarding data confidentiality preservation [20][62], the increasing vulnerability of database servers facing external (hackers) and internal attacks (Database or System Administrator) [35] (losses caused by digital attacks have dramatically grown from $800 million in 1996 to $186 billion in 2004 according to mi2g studies [79]), the emergence of decentralized ways to share and process data thanks to peer-to-peer databases [86] or license-based distribution systems [127] and the ever-increasing concern of parents and teachers to protect children by controlling and filtering out what they access on the Internet [120].

The common consequence of these orthogonal factors is to move access control from servers to clients. Due to the intrinsic untrustworthiness of client devices, all client-based access control solutions rely on data encryption. The data are kept encrypted at the server and a client is granted access to subparts of them according to the decryption keys in its

possession. Sophisticated variations of this basic model have been designed in different contexts, such as Database Service Providers (DSP) [62], database server security [63], non-profit and for-profit publishing [18][80][82] and multilevel databases [3][19][100]. These models differ in several ways: data access model (pulled vs. pushed), access right model (DAC, RBAC, MAC), encryption scheme, key delivery mechanism and granularity of sharing. However, these models have in common to minimize the trust required on the client at the price of a static way of sharing data. Indeed, whatever the granularity of sharing, the dataset is split in subsets reflecting a current sharing situation, each encrypted with a different key, or composition of keys. Thus, access control rules intersections are precompiled by the encryption. Once the dataset is encrypted, changes in the access control rules definition may impact the subset boundaries, hence incurring a partial re-encryption of the dataset and a potential redistribution of keys.

Unfortunately, there are many situations where access control rules are user specific, dynamic and then difficult to predict. Let us consider a community of users (family, friends, research team) sharing data via a DSP or in a peer-to-peer fashion (electronic calendars, address books, profiles, research experiments, working drafts, etc.). It is likely that the sharing policies change as the initial situation evolves (relationship between users, new partners, new projects with diverging interest, etc.). The exchange of medical information is traditionally ruled by strict sharing policies to protect the patient's privacy but these rules may suffer exceptions in particular situations (e.g., in case of emergency) [48], may evolve over time (e.g., depending on the patient's treatment) and may be subject to provisional authorizations [72]. In the same way, there is no particular reason for a corporate database hosted by a DSP to have more static access control rules than its home-administered counterpart [20]. Regarding parental control, neither Web site nor Internet Service Provider can predict the diversity of access control rules that parents with different sensibility are willing to enforce. Finally, the diversity of publishing models (non-profit or lucrative) leads to the definition of sophisticated access control languages like XrML, XACML or ODRL [91][91][93][127]. The access control rules being more complex, the encrypted content and the licenses are managed through different channels, allowing different privileges to be exercised by different users on the same encrypted content.

In the meantime, software and hardware architectures are rapidly evolving to integrate elements of trust in client devices. Windows Media [80] is an example of software solution securing published digital assets on PC and consumer electronics. Secure tokens and smart cards plugged or embedded into different devices (e.g., PC, PDA, cellular phone, set-top-box) are hardware solutions exploited in a growing variety of applications (certification, authentication, electronic voting, e-payment, healthcare, digital right management, etc.). Finally, TCG [112] is a hybrid solution where a secured chip is used to certify the software's installed on a given platform, preventing them from hacking. Thus, Secure Operating Environments (SOE) become a reality on client devices [115]. Hardware SOE guarantee a high tamper-resistance, generally on limited resources (e.g., a small portion of stable storage

and RAM is protected to preserve secrets like encryption keys and sensitive data structures).

The objective of this thesis is to exploit these new elements of trust in order to devise smarter client-based access control managers. The goal pursued is being able to evaluate dynamic and personalized access control rules on a ciphered input document, with the benefit of dissociating access rights from encryption. The considered input documents are XML documents, the de-facto standard for data exchange. Authorization models proposed for regulating access to XML documents use XPath expressions to delineate the scope of each access control rule [18][29][39][58].

2 Addressed issues

The issues which are addressed in this thesis are the following:

- *Streaming evaluation of the access rules:* Many researches have been conducted to define different XML access right semantics during these last few years. However, only a limited number of works were interested in the evaluation of such models, and they generally rely on memory consuming methods. At the present, to the best of our knowledge, none of them considers their evaluation in a streaming fashion, thus precluding their use for new publishing models (broadcast, push context) and on lightweight devices (e.g., cell phones). In the meantime, query evaluation on streaming XML data has been addressed by several works. However, the evaluation of access rights in a streaming fashion is more complex considering that access control rules are not independent and can lead to conflicts.

- *Efficient evaluation coping with the SOE constraints*: While *Secure Operating Environments* provide a high level of security, they have drastic constraints. Indeed, they have a very tiny amount of RAM and this resource is unlikely to grow wrt. the other resources of the SOE in the future [5]. Indeed, the RAM competes with the EEPROM and ROM on the micro chip physical space (space limited for security reasons). As the RAM has a very low density wrt. to EEPROM and ROM, increasing the RAM capacity means reducing drastically the other resources. Moreover, the throughput is quite low at the present wrt. processing power and so are the cryptographic processing. Such considerations lead us (i) to consider streaming algorithms to minimize the consumption of RAM and (ii) to provide indexing methods to reduce the volume of incoming data in order to accommodate the low bandwidth and the cost incurred by cryptographic functions.

- *Guarantee the confidentiality of data:* Considering untrusted servers brings many issues. First, in order to ensure the confidentiality of data, the data are stored in an encrypted form on the server. However, this can be enforced using many encryption schemes but a special attention has to be given to prevent cryptographic attacks on the encrypted data

(inference and statistical attacks). The data can be encrypted using a coarse grain scheme, meaning that a whole file needs be retrieved prior to decryption. Conversely, other encryption schemes give more flexibility and allow retrieving only the needed subparts of the encrypted file prior to decryption. However, such scheme may be less secure if no care is taken. Second, a hacker may alter the encrypted data or use replay attacks (e.g., create inconsistency using different versions of data and access rules) in order to mislead the evaluator and get access to unauthorized data.

3 Contribution

In our proposal, we clearly separate the encryption scheme from the access control policies, thus enabling rights to be dynamic and personalized. That leads us to provide a full-integrated solution combining the three issues aforementioned. The following points compose the contribution:

1. *Accurate streaming access control rules evaluator*
   We consider a basic model based on the most popular access control models for XML which uses the XPath language to express personalized and fine-grained access control policies. We propose a streaming evaluator of XML access control rules supporting this model. The choice of a streaming evaluator allows to cope with the SOE memory constraint and to support different dissemination models (e.g., push model) that are likely to consume streaming documents. The proposed evaluator detects and solves accurately conflicts that may appear between the different access rules to produce a consistent authorized view to the user.

2. *Skip Index*
   We design a compact index structure embedded in the XML stream allowing to quickly converge towards the authorized parts of the input document, while skipping the others, and to compute the intersection with a potential query expressed on this document (in a pull context). Indexing is of utmost importance considering the two limiting factors of the target architecture: the cost of decryption in the SOE, and the communication cost between the SOE, the client and the server.

3. *Security techniques to protect the confidentiality of data*
   In order to protect the confidentiality of data, we propose three techniques. First, an encryption scheme to counter inference attacks. Second, a mechanism combining hashing and encryption techniques to make the integrity of the document verifiable despite the forward and backward random accesses generated by the *Skip Index*. The overhead incurred by the checking mechanism is minimized by delegating to the untrusted terminal intermediate computations. Third, we propose a solution to protect the access right refresh mechanism from replay attacks conducted by a hacker to gain unauthorized access.

4 Outline

This thesis is organized as follows:

Chapter 2 gives the background which is necessary for a good understanding of this thesis. First, it provides the basis of XML which is the data format considered in our study, and query languages such as XPath to define our access control model. Then we focus on methods to evaluate a single XPath query in a streaming fashion, exhibiting the difficulties to deal with predicates. Indeed, the delivery of an element may depend on a predicate which may be solved later in the data stream, thus generating pending situations. Then it gives the principles of cryptographic techniques to guarantee the confidentiality, the integrity and authenticity of data, and finally the problem of completeness. Then, it provides an overview of Secure Operating Environments (SOE). Finally we provide a brief overview of different access control models.

Chapter 3 presents the state of the art on XML access control. It provides an overview of the different access control models for XML, including a basic model which is used in our study. While there is a large bunch of works on the specifications of access control for XML, only a few are interested in their evaluation. We present two different methods which evaluates the access control policies on a server: DOM labeling in which each element of the XML document is marked to tell which users has access to it; and query rewriting used in a pull context in which the user queries are rewritten to comply with their access control policies. However, they are not adapted to our context since the evaluation is performed on a server which may be compromised. Therefore, we present different approaches which evaluates the access control policies on client devices by relying solely on encryption: data are encrypted using different keys and access is granted to those having the proper decryption keys. We show how the existing methods tackle the problem of distribution of keys and conclude by giving their limitations wrt. dynamic and personalized access control rules.

Chapter 4 provides a thorough description of our solution giving in details the contribution aforementioned. We describe how an SOE can be integrated in an insecure infrastructure to protect the confidentiality of data and how access control rules can be evaluated in a streaming fashion thanks to automata. Then we present the *Skip Index* embedded in the XML document which allows to reduce substantially the communication and decryption costs. Then we show two solutions to deal with pending situations which occur when the delivery of an element is conditioned by some data which appear later in the stream. Finally, we provide techniques to protect data from tampering and to prevent replay attacks.

Chapter 5 assesses our solution. To this end, it first provides performance evaluation results on advanced smart cards. Then it shows the feasibility of our approach on off-the-shelves smart cards and finally describes how our technology can be beneficial for different application contexts ranging from personal folders to DRM applications.

Chapter 6 proposes some optimizations. While our solution achieves good performance, this chapter analyzes the possible optimizations that can be drawn and sketches solutions to improve the accuracy of our index thanks to some partial knowledge of the schema and by considering access locality.

Chapter 7 concludes and draws research perspectives.

Chapter 2 – Background on underlying technologies

1 Introduction

In this thesis, we exploit three different technologies. First, XML which serves as a versatile and universal format to share and distribute any kind of data. As, all the access control for XML data are based on XPath queries, we give a special attention to them and on their evaluation. However, in order to have a global overview of XML technologies, we present also other query languages. Second, we describe cryptography techniques to protect the data confidentiality. Indeed, as we consider insecure infrastructure, the data can be subject to attack conducted by hackers who can alter or make analysis on the encrypted data. Finally, we use *Secure Operating Environments* (SOE) to enforce these protections. Even if in our solution, we can rely on several SOE, we describe more into details smart cards which have served in our experiments since they are a cost effective and versatile secure component. At the end, we provide a brief overview on access control models.

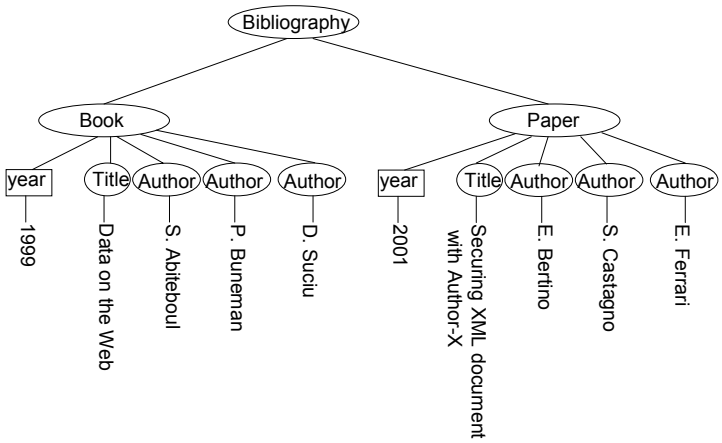


Figure 1. Tree representation of an XML document

2 XML background

The XML (*eXtensible Markup Language*) [119] was standardized by the W3C (*World Wide Web Consortium*) in 1996 to provide a universal versatile language to represent various kind

of data, including text, pictures and multimedia data. An *XML document* consists of *elements*, *attributes* and *text nodes* which are organized as a tree. The content of each element is composed of a set of attributes and a sequence of elements, or a text node. The list of attributes is a list of 2-uple *(name, value);* the *name* being unique within the list. A *text node* is a sequence of characters. An example of an XML document is depicted in Figure 1 where elements are circled and attributes are boxed. We represent here a simplified bibliography containing books and research papers, each of which identified by a year, title and authors.

In order to store or exchange an XML document, it has to be encoded using a compatible format. This is the XML format, which is a plain-text format and defined as follows. Elements are encoded thanks to opening tags (e.g., <*Bibliography*>) and closing tags (e.g., <*/Bibliography*>) which delimit their content. Attributes are appended to element in the form of *name=”value”* (e.g., year=”1999”) inside an opening tag. Finally, characters are encoded as is. The encoding of the XML document pictured in Figure 1 is represented in Figure 2.

```
<Bibliography>
  <Book year=”1999”>
    <Title>Data on the Web</Title>
    <Author>S. Abiteboul</Author>
    <Author>P. Buneman</Author>
    <Author>D. Suciu</Author>
  </Book>
  <Paper year=”2001”>
    <Title>Securing XML documents with Author-X</Title>
    <Author>E. Bertino</Author>
    <Author>S. Castagno</Author>
    <Author>E. Ferrari</Author>
    <Conference>IEEE Internet Computing</Conference>
  </Paper>
</Bibliography>
```

Figure 2. Sample XML document

An XML document can be specialized to support only some tags and structural patterns. For instance, one may define a bibliography to contain only books and papers. This can be achieved using a schema. The schema aims at defining the legal building block of an XML document. The DTD (*Document Type Definition*) [119] language can be used to specify a basic document structure with a list of legal elements. In Figure 3, the DTD associated to the XML document described above says that a *Bibliography* can only have *Book* and *Paper* elements which can have in turn a *Title* followed by zero or more *Authors*. Additionally the *Paper* element has a *Conference* element. For both *Paper* and *Book*, they are required to have the attribute *year* specified and the element *Title*, *Author* and *Conference* must contain a text node.

The DTD provides a basic description of the structure of the document but do not provide a way to specify data typing (e.g., an element must have an integer) and constraints on the cardinality of elements (e.g., an element must have between two and three child elements).

```
<!ELEMENT Bibliography (Book|Paper)*>
<!ELEMENT Book (Title, Author*)>
<!ELEMENT Paper (Title, Author*, Conference)>
<!ATTLIST Paper year CDATA #REQUIRED>
<!ATTLIST Book year CDATA #REQUIRED>
<!ELEMENT Title #PCDATA>
<!ELEMENT Author #PCDATA>
<!ELEMENT Conference #PCDATA>
```

Figure 3. DTD Sample

That led the W3C to define a new standard, the XML Schema [123] which addresses these problems. Unlike the DTD, the XML Schema is defined using the XML format. An example is given in Figure 4. The definition of the schema is basically structured in the same way that the XML document it describes, and for each element, it specifies its type as well as the underlying elements (thanks to the elements prefixed by "*xs:*"). In the example, we specify that *Bibliography* is made up of elements *Book* and *Paper* which may occurs in different orders (*xs:all*). The *Book* element can occur many times and is composed of an attribute *year* defined as an integer, a sequence of an element *Title* and of at least one *Author* which contains a string *text node*. In an XML schema, the cardinality of element can be mentioned explicitly (if not, it means that the element appears only once) and the data typing is more precise (we can even define *year* to be an element which has values between 1990 and 2005).

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="Bibliography">
    <xs:complexType>
      <xs:all>
          <xs:element name="Book" minOccurs="0">
            <xs:attribute name="year" type="xs:integer"/>
            <xs:complexType>
              <xs:Sequence>
                  <xs:element name="Title" type="xs:string"/>
                  <xs:element name="Author" type="xs:string" minOccurs="1"/>
              </xs:Sequence>
            </xs:complexType>
          </xs:element>
          <xs:element name="Paper" minOccurs="0">
            <xs:attribute name="year" type="xs:integer"/>
            <xs:complexType>
              <xs:Sequence>
                  <xs:element name="Title" type="xs:string"/>
                  <xs:element name="Author" type="xs:string" minOccurs="1"/>
                  <xs:element name="Conference" type="xs:string"/>
              </xs:Sequence>
            </xs:complexType>
          </xs:element>
      </xs:all>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Figure 4. Sample of an XML Schema

2.1 Query languages

To exploit XML data, the W3C provides high level languages to locate elements in the XML

document such as XPath [122] and perform operations on it (e.g., aggregation, restructuring) such as XQuery [125] and XSLT [124].

2.1.1 XPath

Given an XML document, one may want to locate some elements which satisfies a set of conditions. These conditions can operate on attributes and text nodes, etc. Such conditions can be easily captured using regular path expressions. A basic XPath expression (e.g., */Bibliography/Book/Author*) consists of location steps containing the tag name of traversed elements to reach the requested subtrees (in the same way that a path expression locates files in a file system). The elements which are traversed may be related together by different axis. The most common ones are the *child axis* (*/*), *descendant-or-self* (*//*) and *attribute* (@). For instance, if we consider the XPath */Bibliography//Authors*, all the elements *Authors* that are descendant of the *Bibliography* elements are a match. Additionally, some conditions, called predicates can be defined over the XPath expression. For instance, the expression *//Book[@year="1999"]/Author* returns the *Authors* having a parent element *Book* which has an attribute *year* equals to 1999. Finally, one can use a wildcard (*) to identify any elements and XPath expressions recursively into predicates.

2.1.2 XQuery

XQuery is a powerful and convenient language which has been designed for processing XML data. The main idea of this language is to provide a language to navigate into different documents using conditions and to produce formatted results. The structure of an XQuery expression is defined by the following clauses: *FOR, LET, WHERE and RETURN*. The *FOR* or *LET* clause associates one or more variables using XPath expressions. The *WHERE* clause imposes additional conditions. Finally, the *RETURN* clause tells which data is to be returned and how they are presented.

```
<Author name="S. Abiteboul">
  <Publication>Data on the Web</Publication>
  <Publication>Exchanging intensional XML data</Publication>
  …
</Author>
```

Figure 5. List of publications of S. Abiteboul

Suppose that we want to build a new document that would provide for the author *S. Abiteboul* all his books and papers as presented in Figure 5. One can issue an XQuery as depicted in Figure 6 that first defines the element *Author* to be the root element of the resulting document. For this element, we define a *FOR WHERE RETURN* structure which returns a sequence of elements *Book* or *Paper* containing all the publications authored by *S. Abiteboul*. The *FOR* clause considers each child element of *Bibliography* iteratively. These elements are filtered out by the *WHERE* clause which checks if it contains an element *Author* equals to *S. Abiteboul*. In the positive case, the title is returned inside an element *Publication*. One of the interesting aspects of XQuery is that an XQuery expression can be

nested in any of the clauses. Obviously, this adds to its expressiveness but makes its evaluation more complex. For this reason, in the case of a simple restructuring of document, such as a conversion from XML to HTML, the XSLT language is more appropriate.

```
<Author name=”S. Abiteboul”>
     {
       FOR $i in /Bibliography/*
       WHERE $i/Author=’S. Abiteboul’
       RETURN
        <Publication>{$i/Title/text()}</Publication>
     }
</Author>
```

Figure 6. XQuery statement

XSLT

In the *eXtended Stylesheet Language Transformations* (XSLT) [124], document processing is performed using a stylesheet which contains different rules to convert an XML file into another format. These rules are defined by templates (*xsl:template*) which purpose is to locate (e.g., *match=”/”*) element and to tell how to transform them. We depict in Figure 7 an example of an XSLT stylesheet which produces the same result obtained in Figure 5. The transformation stylesheet is described in XML using special elements (prefixed by *xsl:*).

```
1. <xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">

2.   <xsl:template match="/">
3.    <Author name=”S. Abiteboul”>
4.     <xsl:apply-templates select="Bibliography/*[Author=’S. Abiteboul’]"/>
5.    </Author>
6.   </xsl:template>

7.   <xsl:template match="/Bibliography/*[Author=’S. Abiteboul’]">
8.    <Publication>
9.     <xsl:value-of select=”Title/text()”/>
10.   </Publication>
11.  </xsl:template>

12. </xsl:stylesheet>
```

Figure 7. XSLT sample

In this example, there are two templates (*xsl:template*). In the first one, we tell the transformer to replace the document root element (line 2) by the element *Author* (line 3). In this element, we tell to apply the second template operating on books or papers written by S. Abiteboul (line 4). For each matching, we create a new element *Publication* (line 8) containing the title of the book or paper (line 9).

2.2 Data processing

Unlike relational data, XML data is structured in a hierarchical way and can be seen as a tree of elements. This tree can be processed easily using the *Document Object Model* or DOM API [118] provided by the W3C. In the DOM approach, the entire XML document tree is first materialized in main memory before it can be processed by a program. Then the DOM

API provides methods to navigate (e.g., go from an element to its child element, its sibling, its parents, etc...) and to locate a particular element in the tree.

While the DOM API is convenient for programmers, it suffers from many weaknesses. First, the materialization of the tree is memory consuming (typically 5 to 10 times the space of the equivalent XML file) precluding its use on lightweight devices. This is mostly due to the fact that for each element a set of pointers (e.g., to parent, sibling, child elements, descendant elements) has to be maintained. Second, the startup time to materialize the whole document in memory can be high for big documents. Indeed, all the XML data need be read before the processing can start. All these limitations make it clearly inappropriate for streaming data.

```
<Bibliography>              → startElement(“Bibliography”, {})
  <Book year=”1999”>        → startElement(“Book”, {year=”1999”})
    <Title>                 → startElement(“Title”, {})
      Data on the Web       → characters(“Data on the Web”)
    </Title>                → endElement(“Title”)
    <Author>                → startElement(“Author”)
       S. Abiteboul         → characters(“S. Abiteboul”)
    </Author>               → endElement(“Author”)
  </Book>                   → endElement(“Book”)
</Bibliography>             → endElement(“Bibliography”)
```

Figure 8. Sequence of events raised by a SAX parser

These technical limitations can be tackled using the *Simple API for XML* or SAX [102] which has been designed to process XML data in a streaming fashion. In the SAX model, the document is seen as a sequence of events which are raised along the parsing. The document is parsed from start to end and every time a starting tag is encountered, the event *startElement* (coming with the tag name as well as the list of attributes) is raised. Whenever an ending tag is encountered the event *endElement* (coming with the tag name) is raised. Finally, the event *characters* (coming with a text string) is raised for text nodes. The sequence of events is depicted in Figure 8. The user application is then in charge of taking actions based on these events.

In this model, the processing of data can start as soon as the parsing begins. The RAM consumption is bounded by the maximum size of the elements and characters, thus making it particularly appropriate for lightweight devices. However, in this simple model, locating an element (e.g., the element *Title* of a *Book* written by “*S. Abiteboul*”) is particularly complex considering that an element can occur anywhere in the stream. We discuss query processing in the next section.

To conclude, DOM provides an easy way for the programmer to process data but it does not accommodate streaming data and is time and space consuming. On the other hand, SAX processes data in a streaming fashion, minimizes the memory consumption and has no startup time. However, it offers a very basic parsing method resulting in a more complex management for the programmer. Clearly, using DOM or SAX depends on the kind of

applications considered (big vs. small document, non-streaming vs. streaming document, much vs. few RAM).

2.3 XML streaming evaluation

In this thesis, we consider constrained devices (e.g., smart cards) which have a very limited amount of RAM. As all the access control models for XML are based on XPath queries, we focus on the different techniques based on automata to evaluate them in a streaming fashion and to minimize the consumption of RAM. The XPath evaluation problem can be classified in two contexts. First, the query routing problem described by Diao et al. [43] and Green et al. [61] which purpose is as follows: given a large set of XPath, identify the XPath expressions which apply on an XML document. Such solutions can be used for filtering out documents of interest for a user in the push context (e.g., publish/subscribe, alarms). Second, the delivery problem addressed by Peng and Chawathe [95], Chen et al. [33] and Olteanu et al. [92] which is interested in delivering the subparts of a document matching a single XPath.

While they provide interesting and some common features with our work, they are not adapted to access control evaluation. Indeed, in the access control evaluation, we are interested in delivering subparts of a document which issues depends of several XPath access rules which are not independent and may conflict. This is clearly different from the query routing problem since they are not concerned by the delivery of subparts of documents. Moreover, it cannot be a simple extension of the delivery problem, to many queries since in our case, the XPath access control rules are not independent and may generate conflicts. We will address the precise problem of access control evaluation in Chapter 4.

In the following, we provide an overview of different approaches to evaluate XPath queries thanks to automata. Because our work is closer to the delivery problem, we get more into details in the works addressing this issue, while borrowing some techniques from the routing problem. For ease of understanding, we preferred an intuitive presentation rather than a formal explanation.

2.3.1 Automata overview

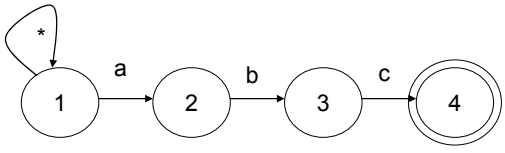


Figure 9. Sample automaton

An *automaton* is a machine having a collection of *states* which is used to recognize a pattern of events. The automaton starts from an initial state and evolves over different states which

can be *normal* or *final*. When the automaton is on a final state, it means that a pattern has been identified (matching). The states are linked altogether by transitions connecting a state to another and is associated to a label. The *label* determines the conditions (generally based on events) for the automaton to go from a state to another. Upon receiving an event, the automaton determines the state to visit next based on its current state and the event. A normal state is represented by a circle while a final state is represented by a double circle. Transitions are represented by directed edges. A transition marked with a * accepts any event.

In the following, we consider different kind of automata. We can distinguish between two major types of automata. First, the Non-*Deterministic Finite State Automata* (*NFA)* which can consider multiple choices when receiving an event. In Figure 9, when the NFA is in state 1 and encounter an event *a*, it may go either to state 1 or state 2. In order to explore all the possible choices, it considers many instances of it at the same time. Second, the *Deterministic Finite State Automata* (*DFA)* which always consider only one choice when receiving an event. However, as illustrated in Figure 10, this may result in a high complexity for the number of transitions and states which may grow exponentially. Figure 9 and Figure 10 represents respectively an NFA and DFA which recognize the same sequence of events. In these examples, there are more transitions in the DFA since it has to consider the case of an event *a* which may occurs at any time and thus every state must handle the event *a*.

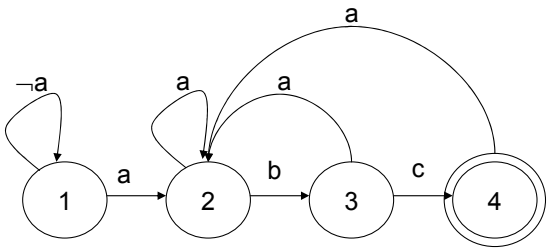


Figure 10. DFA sample

In the NFA approach, the automaton is simpler (thus, it consumes less memory) but its evaluation is more complex since it has to consider many instances at the same time. Conversely, in the DFA approach, the automaton is more complex (exponential number of states and transitions) but the evaluation is simpler since it has to consider only one instance. Recently, Green et al. [61] proposed a way to reduce the space overhead of DFA which are constructed incrementally at execution time in order to build only the necessary states and transitions whenever needed. While it reduces the number of states and transitions to consider in a DFA, the number of states and transitions are still higher than in the NFA approach. As in our study, we are interested in memory constrained environment, we rely on NFA.

2.3.2 Basic streaming evaluation

In this subsection, we explain how to evaluate an XPath query using NFA and SAX events. In a streaming evaluation, the XML document (e.g., Figure 11.b) is parsed by a SAX parser which raises events (e.g., *startElement*, *endElement*, *characters*). These events serve to feed an evaluator in charge of managing an automaton representing the XPath query. Based on extra structures and the automaton, the evaluator determines the elements matching the XPath query.

To have a better understanding of how automata work, we consider tokens, each of which identifies a possible progression in the automaton. At the beginning, a token is placed in the initial state. Whenever receiving an input, all the tokens in the automaton are checked to see if they can move to a another state. In the case of multiple choices, it forks (we get multiple tokens).

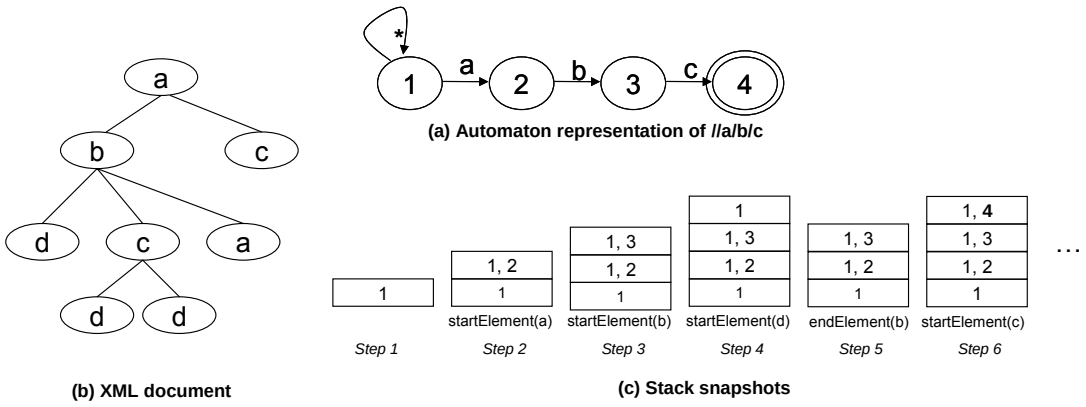


Figure 11. Step by step Execution for //a/b/c

For the sake of clarity, let us first consider XPath without predicate branches. An XPath expression is a regular path expression which can be represented as a Non-deterministic Finite state Automaton (NFA). For instance, the XPath expression *//a/b/c* can be represented by the automaton depicted in Figure 11.a. The *parent-child* relationship is represented by a simple transition (e.g., transition *b*) while the *ancestor-descendant* relationship is represented by both a transition to its descendant and a * self-transition (e.g., transitions *a* and *). The extra self-transition is used to consider all the nested elements which may potentially occurs later in the parsing. We describe here the approach followed in YFilter developed by Diao et al. [43]. In their solution, they use a stack (pictured in Figure 11.c) to keep track of the progression of the tokens in the automaton. Each level of the stack corresponds to a depth in the document and the active tokens are on the top of the stack. The basic idea is as follows:

- When the evaluator receives the SAX event *startElement(tag)*, that is, when it enters in a subtree, the evaluator examines all the possibilities to reach the final state. To this end, all

the tokens on top of the stack are tested to see if they can pass a transition. If the outgoing transition of the token state is *tag* or *, then it moves to the next state (e.g., in Figure 11, when receiving the event *startElement(b)* at Step 3, the token in state 1 goes to state 1 and those in state 2 goes to state 3). The tokens which passed a transition are pushed on the stack, and thus represent the tokens which may lead to a final state.

- When the evaluator receives the event *endElement(tag)*, that is, when it leaves a subtree, it means that no other elements can be found in the current subtree. In this case, none of the tokens at the top of the stack can lead to a final state and thus must be discarded. In this case, the stack is popped (e.g., Step 5) and goes back to the previous situation as just before entering into the subtree (backtracking).

Considering a stack made the evaluation of an XPath without predicate branches quite simple. In the execution snapshot described in Figure 11.c, the XPath //a/b/c is recognized at Step 6 since the final state (state 4) of the corresponding automaton is reached. It means that all the elements inside the current subtree have to be delivered. This can be done easily by marking the level of the stack when the final state is reached. In this case, elements will be delivered until the stack is popped from the marked level.

2.3.3 Dealing with predicates

In the following, we refer as *Simple Path* (*SP)* of an XPath *Q*, the XPath without its associated predicate branches (e.g., the *SP* of */a[b]/c[d]* is /a/c).

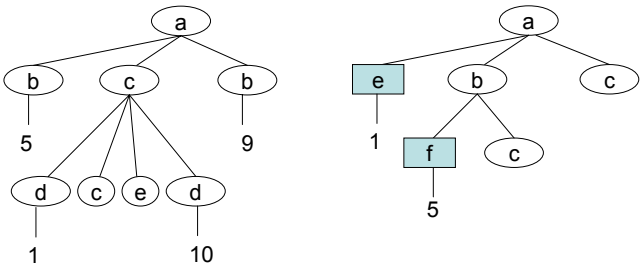
The problem when dealing with predicates (e.g., */a[b=9]/c[d=1]/e*) is that at parsing time the current element which matches the *SP* (e.g., /a/c) can be conditioned by a predicate which can be encountered later in the parsing (e.g., *b=9* and *d=1*). Such predicate is said to be pending. When getting to an element (e.g., *e)* matching the *SP* of an XPath *Q*, there are three situations to consider:

Case 1. All the predicates are found to be true and so the element matching the *SP* as well as the underlying subtree is an answer to the query (e.g., in Figure 12.a, all the predicates of */a[b=5]/c[d=1]/e* are true when the element *e* is parsed).

Case 2. Some of the predicates are found to be false so the element matching the *SP* is not an answer to the query (e.g., in Figure 12.b, the first predicate of */a[@e=2]/b[@f=5]/c* is false when the element *c* is parsed). This can only happen when the considered predicates have special constrained in the schema. For instance, in the case of attributes we can tell that a predicate is false or true since they are unique within an element (e.g., the predicate *@e=2* is false since in the element *a* we have *@e=1*). Moreover, if the schema tells that an element is unique within a subtree, then as soon as the element is parsed, we can tell that the predicate related to it is false or true.

Case 3. Some of the predicates are not resolved yet but can be later in the parsing. These predicates are said to be pending and may generate pending situations (e.g., in Figure 12.a,

the second predicate of */a[b=5]/c[d=3]* is not resolved when the element *e* is parsed but may become true later in the parsing. Even if there is a *d=1*, we cannot say that the predicate is false since another occurrence of *d* such that *d=5* may occur later). The issue of a predicate is eventually known when leaving the subtree related to it (e.g., when leaving the subtree *c*, the issue of predicate *d=3* is known to be false) or earlier if it is found to be true.



(a) XML Sample 1 (b) XML Sample 2

Figure 12. Illustrations of the predicate problems

In case 1 and 2, the issues of elements matching an *SP* is known when it is parsed and so no pending situations is considered. However, for each of them, there must be a mechanism to check if the predicates are true or false when reaching the elements matching the *SP* to decide to discard (if at least one predicate is false) or delivered (if all the predicates are true). We describe in the following the mechanism proposed in YFilter [43] to manage predicates on attributes. Then, in order to comply with more complex predicates on elements resulting in pending situations as mentioned in case 3, we present the work of Peng and Chawathe [95] and some other related works.

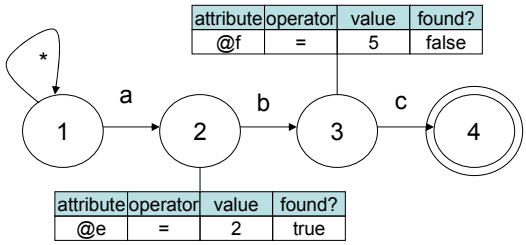


Figure 13. Bookkeeping method

YFilter Diao et al. [43] introduce a bookkeeping method on attributes. The basic idea is to keep in memory the predicates which are solved along the parsing and to check that they all have been solved when getting to the final state. To this end, the authors consider for states having a predicate (on attributes) a 4-uple which stores the conditions (e.g., *@e=2*) and if they have been fulfilled (as illustrated in Figure 13 for the XPath */a[@e=2]/b[@f=5]/c)*. Upon receiving an event *startElement* which comes with the element tag and its subsequent attributes, the parser checks if the conditions on the attributes are satisfied. In the positive case, it puts a *true* in the found field and a *false* otherwise. This field is reset when the

automaton backtracks (e.g., for state 2 when a closing tag *a* is parsed). When reaching the final state, the automaton checks whether all the predicates have been satisfied. If so, the current element (and the subsequent subtree) matches the XPath, otherwise it does not. While this solution is interesting, it does not accommodate predicates on elements (e.g., /a[b/c=5]/d) which may generate pending situations.

Peng and Chawathe [95] were the first to provide a technique to manage pending predicates. Their idea is to consider a complex automaton (as pictured in Figure 14) which compiles all the different possibilities regarding the predicates which are pending (2^n cases are considered for *n* predicates). For each of the pending cases (i.e., cases which considers at least one pending predicate, that is 2^n-*1*), a buffer is defined and is in charge of storing all the subparts of the document which delivery is conditioned by the same set of pending predicates. These buffers are organized hierarchically so that the content of a buffer can be transferred to another (e.g., if predicate *d* is solved, then the buffer considering predicates *b* and *d* to be pending is flushed to the buffer considering only *b* to be pending). These buffers are appended to different groups of automaton states being in charge of the same set of pending predicates.

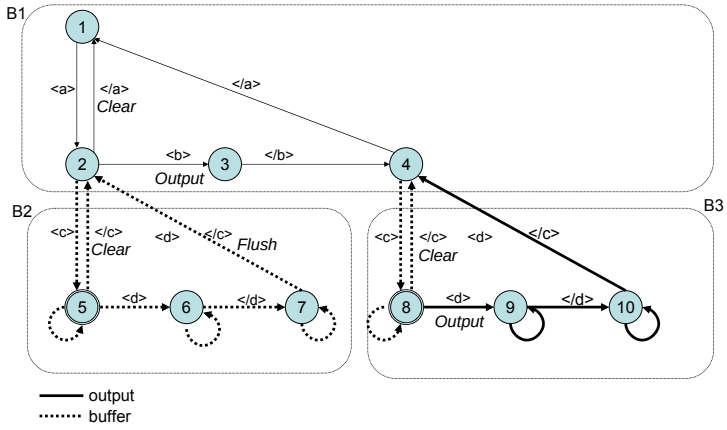


Figure 14. Peng and Chawathe automaton

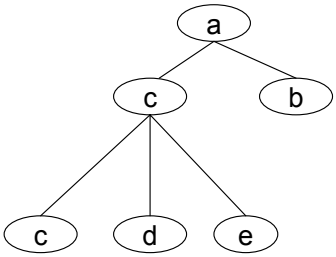


Figure 15. XML Sample

In their model, they consider more complex automata (an example representing the query *Q=/a[b]/c[d]* is depicted in Figure 14) which can take actions when a transition is passed (pushdown transducer). Bold transitions output the current element/text event (meaning that a match is found and all the predicates have been found to be true). Dotted transitions (representing a potential match depending on pending predicates) buffer the current element/text event in the corresponding buffer encompassing the state. Some transitions are labeled with the tag *Output* which means that when the transition is passed, the current buffer is to be output and emptied (all the predicates are found to be true). Some others are tagged with *Flush* which merges the buffer to its parent buffer (caused by a predicate which has been found to be true) and *Clear* which discards the buffer (when a predicate is found to be false). Unmarked transitions are catch-all transitions which are fired if no other

transitions can be passed. The management of the automaton is different because they do not consider backtracking when a closing tag is encountered, rather they handle this event with a transition (e.g., *</d>*). For ease of understanding, we did not represent the conditions on the depth to pass a transition. This is however necessary to check the parent-child relationship and to make sure that opening and closing tag events result from the same element (e.g., the transition *</d>* from state 6 to 7 must not be passed if a nested element *d* is found in the subtree *d*). These information are compiled in a (stack) vector appended to the token which stores the tag and depth of the elements in the path from the root to the current element.

If we consider the XML document described in Figure 15, then the automaton will go into the following states: 1, 2, 5, 6, 7, 2, 3, 4, 1. To manage the two predicates *b* and *d* of query *Q*, we have three buffers, *B1* and its two child buffers *B2* and *B3*. *B1* is responsible for buffering contents which delivery is conditioned by predicate *b*, *B2* is a buffer in charge of contents which are conditioned by predicates *b* and *d*. *B3* is a buffer in charge of the delivery of contents which depends solely on predicate *d* (the predicate *b* has already been found to be true). When the element *c* is parsed, its subtree is buffered in buffer *B2*. Then, as soon as the element *d* is parsed, the buffer *B2* (containing the subtree *c)* which now depends solely on the predicate *b* is therefore flushed in *B1*. Finally, *B1* is output when the element *b* is parsed (from state 2 to state 3).

More difficult cases can occur when dealing with recursive queries (i.e., using //). We just sketch here how it is solved. If we consider the query */a[b]//c[d]*, then during parsing time we have to consider nested potential results (the two subtrees *c* in Figure 15). To manage //, a little modification is done in the automaton: when getting from state 2 to 5 it would have to consider two executions at the same time. If we modeled this execution with a token then this token is forked: it goes from state 2 to state 2 and 5. The token in state 5 considers the buffer *B2* and when another token will get from state 2 to 5, it will consider a buffer *B2'* different from *B2* (since they are in charge of two different instances of the same predicate). In this situation, some elements may be part of both *B2* and *B2'*. When they are flushed, a merging mechanism is defined to have a coherent delivery.

The main advantage of this solution is that it minimizes the use of the buffers which are output as soon as all the predicates are solved. Beside, the management of predicates is compiled in the automaton and no other extra structures are needed (e.g., a stack). However, for each predicate branch, the automaton considers two different cases meaning that the number of states grows exponentially with the number of predicate branches.

Some other works have been proposed to improve this model. In eXpedite [33], Chen et al. propose to use simpler automata and to delegate the complexity of predicates to tokens which are associated to a map of predicates (which tells the predicates which are true). Another approach developed by Olteanu et al. [92] considers also pushdown transducer as in Peng and Chawathe approach [95] but their main idea is to mark every element in the stream and trigger their delivery when predicates are solved. Although the mechanisms developed

in these works are interesting, they do not fit in our context since access rules are not independent and may generate conflicts, so going more into details on these works is clearly outside of the scope of this chapter.

2.4 Summary

In this section, we gave an overview of the XML language as a standard which can be used to exchange various kind of data using different query languages: XPath, XQuery and XSLT. Since, most of the access control models for XML are based on XPath, we chose to focus on XPath evaluation. The evaluation is done in a streaming fashion because we consider in our study memory constrained environments. The streaming evaluation becomes complex as soon as predicates are considered. Indeed, the delivery of data may depend on the values of elements which may occur later in the stream. The evaluation of an access control policy is even more complex since it is composed of several XPath rules which are not independent and may generate conflicts. We propose in chapter 4 a method compliant with the memory constraints of the SOE to manage them efficiently.

3 Cryptography background

Cryptography is a science using both mathematical and linguistic techniques to secure the exchange of information. Historically, it started with encryption to protect secrets from unauthorized parties. Modern cryptography provides mechanisms for more than just keeping secrets and covers a wide range of applications (e.g., electronic voting, digital cash, etc.). At the present, everyone uses cryptography, which is often built transparently into much of computing and telecommunications infrastructure.

In our study, we are interested in sharing and distributing data securely. To this end, our solution has to guarantee the confidentiality of data, i.e., data must not be disclosed to any unauthorized party. As we rely on untrusted infrastructure, data may be tampered by hackers which may alter the data to mislead the system and gain access to unauthorized data. We see in the following how we tackle this problem thanks to integrity and authenticity checking. Finally, in the pull context, the server can be malicious and provide to the user an incomplete answer, once again to mislead the system. This problem is known as completeness. We detail in the following some solutions to tackle this problem.

3.1 Data confidentiality

The idea of encryption is to preserve the confidentiality of data thanks to a reversible function depending on an encryption key which transforms a clear text data into a meaningless data representation. Then, only the users in possession of the decryption key can make the reverse operation and recover the original data.

3.1.1 Encryption scheme

We can distinguish between two types of encryption scheme: symmetric and asymmetric. Symmetric encryption or secret key encryption uses the same key to encrypt and decrypt the data. It can be used to share a secret within a community of users. In this approach every party sharing the secret must share the same key. The problem which arised was that a secret key had to be distributed securely (using physical means) to every user.

This problem was solved in 1978 by Ron Rivest, Adi Shamir, and Leonard Adleman who provided an asymmetric encryption scheme called RSA [101]. The idea is to use a public key to encrypt the data and a corresponding private key to decrypt it. In this model, every user has a *private key* which is kept secret, and a *public key* which can be distributed publicly. If Alice wants to send a secret data to Bob, she encrypts it using Bob's public key and sends it to him. Bob will then decrypt it using his private key. Because, asymmetric encryption is relatively slow, it is generally used in combination with symmetric encryption. To secure a communication between two users Alice and Bob, Alice first generates a random secret key and sends it to Bob using the asymmetric encryption scheme. Then, Alice and Bob communicate using the secret key to secure their messages.

Hierarchical keys scheme is another way to simplify the management of keys distribution. The idea is to consider a hierarchy of keys by setting partial orders among them (e.g., key K1 dominates K2 and K3). In the hierarchical keys concept, a key K can be used to decrypt any data encrypted with a key dominated by K. This is particularly suitable for hierarchical organization. For instance, if the director has access to his private data as well as those of his secretary, then a hierarchical key scheme will enable the data owned by the secretary to be decrypted using either her decryption key or the director's decryption key. This scheme has been studied in many works [3][15][19][99][100].

3.1.2 Encryption modes of operation

Encryption is generally performed on blocks (e.g., 64-bit blocks) and can be processed in a streaming fashion (one block at a time). In a basic scheme, ECB (*Electronic Code Book*) encoding scheme, blocks are encrypted independently. In this scheme, two data blocks having the same value have the same encrypted value. This raises the problem of statistical attacks which take advantage of frequencies of occurrences in the encrypted data. For instance, let us consider an encrypted database containing records of persons, each of which containing their nationality and other fields. It is likely that in France, as most of the persons are French, most of the records will have the same encrypted value of their nationality. However, a few of them, foreigners will have different encrypted values. By making statistics on the encrypted database, we can easily identify which encrypted tuples represent French persons and those which do not. Using such techniques on other record fields and extra information from other clear-text databases (e.g., electoral lists) may reveal, to some extents, the content of the whole encrypted database. To alleviate this problem, the CBC

(Cipher Block Chaining) can be used. The principle is to chain a block to the previous one. For the first one, we consider an initialization vector (IV) which can be public. Data blocks are encrypted as follows: $B_i = E_K(B_{i-1} \oplus b_i)$ where b_i denotes the clear text of the block to encrypt and B_i the encrypted block. E_K is the encryption function. For the first block, we use IV instead of B_{i-1} which can be set by the programmer. In this case, even if two data blocks have the same value, they will have different encrypted values preventing any statistical attacks on data.

3.1.3 Encryption Standards

One of the most famous symmetric encryption is DES (Data Encryption Standard) [88] which was invented by IBM in 1974. It was adopted by the NIST (National Institute of Standard and Technology) in the United States in 1976 as an open standard and has been massively used for years. The DES algorithm operates on 64 bit blocks and uses a 56 bit key. It has known no weaknesses except the length of the key which has become too small wrt. computer power, thus becoming weak against brute-force attacks. For this reason, the NIST abandoned the support for DES in 1997 and launched an international contest to develop the Advanced Encryption Standard (AES) [87] as a replacement for DES. In the meantime, the Triple DES has been endorsed by NIST as a temporary standard to be used until the AES was completely defined. Triple DES considers 3 subkeys of 56 bits which, like DES operates on 64 bit blocks. It first encrypts the block using the first key, then decrypts it using the second one and then encrypts it again using the third one. As a result, the triple DES proved to be more secure since it relies on a 168-bit key. Moreover, it keeps a backward compatibility with the DES standard because if we consider three identical subkeys (56 bits), it results in a simple DES encryption. However, it turned to be slow compared to the AES invented by Joan Daemen and Vincent Rijmen who won the contest and which was adopted as a standard by the NIST in 2001. AES operates on 128-bit block, can use 128, 192 or 256 bit keys, is fast either in software and hardware solutions, and requires little memory.

All these standards are implemented in most cryptographic libraries and can be used interchangeably with negligible changes in a program.

3.1.4 Integrity and authenticity

When data is transmitted from a user to another, it may be altered on purpose by an attacker. To detect tampering, *Cryptographic hash functions* (CHF) may be used. A CHF is a function H which takes in input data x of any length and returns a relatively small fixed-size bit string (e.g., 160 bits) called message digest. The basic requirement of a CHF is that $H(x)$ must be relatively easy to compute, $H(x)$ is *one-way* and *collision-free*. A hash function H is said to be one-way if it is hard to invert, where "hard to invert" means that given a hash value h, it is computationally infeasible to find some input x such that $H(x) = h$. *Collision-free* means that

it should be unfeasible to find two different messages *x* and *y* such that *H(x) = H(y)*.

The most popular CHF is the *SHA-1* which produces a 160 bit digest from data of any size from (streaming) 512-bit blocks. Other algorithms such as MD5 which was popular in the past has not been considered to be secure since a Chinese team discovered a complete collision in this algorithm [116] in 2004.

To check the integrity of a message, we cannot rely solely on a CHF. Indeed, integrity requires the data (i.e., *x*) or the hash representation (i.e., *h*) to be encrypted in order to prevent a hacker from altering the data and re-computing a valid hash. In our context, the data must be at least encrypted to guarantee the confidentiality of data.

There are two interesting integrity schemes to consider. First, the hash value is computed from the clear-text value and then encrypted. Second, the hash value is computed from the encrypted data and encrypted again. The first case is to be precluded because when considering two identical clear text data, the encryption will provide two different encrypted data (using a CBC mode) but the hash values will be the same, making the data weak against inference attacks. The second case is particularly interesting considering that the data needs not be decrypted to check its integrity.

In addition, if the hash is encrypted using an asymmetric scheme, then it will authenticate the origin of the message. The encrypted hash is then called the digital signature of the message. To check the origin and the integrity of a message, Bob computes the digest *D1* of the message. He then decrypts the signature thanks to Alice's public key and obtain *D2*. If *D1* equals *D2* then the signature is valid and this proves that the message has not been altered and that it was created by Alice.

Therefore, signatures can also serve for non-repudiation proofs. For instance, when one agrees on a contract, he signs it using his private key. This signature proves that he has accepted the contract.

3.1.5 Completeness

We consider here the context of an insecure server which stores data that its owner wants to share. When a user issues a query, the server may be malicious and return an erroneous answer. The correctness of the answer can be checked thanks to a signature computed by the owner of data. If we consider the case where the user is interested in a subset of the data (e.g., he wants all the blocks between blocks 5 and 7) then he would also have to consider as many signatures as there are blocks in the response. However, we could not make sure that none of them are missing (e.g., it returns only block 3 while there should be block 3, 4 and 5). This problem is known as completeness of data. In the following, we present the basis of this problem and put aside works related to completeness of indexed values as studied by Devanbu et al. [41].

To alleviate this problem, we may think of a solution which integrates the block index in the

computation of the signature (e.g., concatenation of the block content and the block index). Such solution works if we use a contiguous numbering of the index (e.g., between block 2 and 7, there are blocks 3, 4, 5 and 6). However, if we do not have a contiguous numbering of the blocks (simply because a block has been deleted), the proposed solution does not work. Indeed, suppose that there are only blocks 3 and 5 between blocks 2 and 7. If the server returns only block 2 and its signature, we could not tell that block 5 is missing.

We present below two methods which allow to solve this problem.

Trivial solution

A trivial solution would be that the signature depends on the block content and the signature of the previous block as depicted in Figure 16.

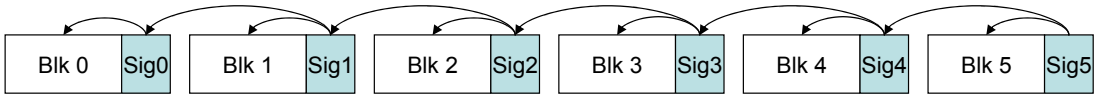


Figure 16. Signature chaining

If the user Bob is interested in getting the blocks between block 1 and block 3 from the server, he gets blocks 1, 2 and 3 but also the signatures *Sig0* and *Sig3*. Thanks to *Sig0* and *Blk1*, he will compute *Sig1*. Thanks to *Sig1* and *Blk2* and then to Blk3 he computes *Sig3* and checks that is equals to the *Sig3* that he received from the server. If they are not equal, then it means that a block has been altered or that a block is missing.

If the server omits a block, e.g., *Blk2* on purpose, it would have to produce a false signature *Sig0* so that computed with *Blk3*, it generates *Sig3*. This is unfeasible since the server would have to know the private key of the owner to generate the signatures.

Consequently, 2 extra signatures have to be sent by the server along the query answer when the user is interested in consecutive blocks.

The major problem of this solution is that it does not support well updates. Indeed, if a block is updated or inserted, then all the signatures of the block on its right have to be recomputed. Therefore, when the owner update/insert a block, he will have to compute on average *n/2* extra signatures and send them to the server. Such solution is indeed not viable in a solution considering updates on data since it can result in a large traffic of data.

Merkle Hash Tree

To alleviate this problem, *Merkle Hash Tree* [78] can be used. The idea is to consider a binary tree as pictured in Figure 17 on top of the blocks as follows:

- leaves of the tree (e.g., h_0) are computed as $h_i=H(B_i)$, where H denotes a hash function (e.g., SHA-1) and B_i a data block.

- internal nodes (e.g., h_{01}) of the binary tree are computed as follows: $h_{lr}=H(h_l \mid h_r)$, where | denotes the concatenation operator and h_l (resp. h_r) the value of the left child (resp. right child).

Finally, we consider the MHT signature computed as the encryption of the root hash value using the private key of the owner of the data. When a user requests blocks found between two blocks, he will get in addition the MHT signature and some hashes along the response. Thanks to the hashes and the response, he will compute the hash of the Merkle Hash Tree root and compare it with the MHT decrypted signature. If they are different, then a block has been altered or is missing.

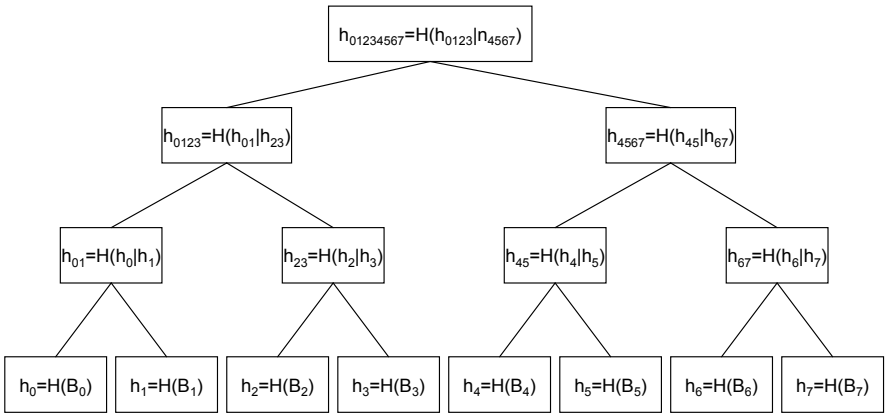


Figure 17. Merkle Hash Tree

Let us illustrate how it works on a simple example. When a user asks for blocks found between block 2 and 4, he will get B_2, B_3, B_4 but also h_{01}, h_5, h_{67} and the MHT signature. Bob will compute h_{23} thanks to B_2 and B_3, and h_{45} thanks to B_4 and h_5. Thanks to the newly computed h_{45} (resp. h_{23}) and h_{67} (resp. h_{01}) he will compute h_{4567} (resp. h_{0123}) and finally compute $h_{01234567}$ that he compares with the decrypted MHT signature. Imagine that the server sends only blocks B_2 and B_4 (omitting B_3). In order to mislead the client, the server would have to provide hashes which, once computed with the returned blocks B_2 and B_4 will result in the root hash. This operation is unfeasible considering that CHF functions are collision resistant functions.

The number of extra information depends on the consecutive blocks returned by the server and can varies from 0 to $Log_2(n)$ hashes plus one signature, n denoting the number of blocks in the document.

When updating a block, only $Log_2(n) + 1$ hashes (hash of tree nodes in the path from the root to the updated block) have to be recomputed. When inserting a block, we would have to consider more operations to keep the tree balanced and such operation can be quite costly (e.g., all the hashes of the Merkle Hash Tree). However, the hash values can be computed by the untrusted server. In this case, the owner of data will have to send only one signature to the server.

Merkle Hash Tree provides an efficient way to check the completeness of data on an untrusted server. Indeed, when a user issues a query, the server returns in addition at most $Log_2(n)$ extra hashes and one signature to check the correctness of the answer. Moreover, when the owner updates the data, the hash values can be computed by the server and only one signature has to be sent by the owner. In this case, the update operation result in a low traffic between the owner and the server since most of the computation can be done at the server side.

3.2 Summary

We showed in this section that to secure data, they have not only to be encrypted but also to be made resistant against malicious alteration using techniques combining hashing and encryption. This becomes even more complex when considering queries since we have to consider the completeness problem which consists in checking that no data is missing.

4 Secure Operating Environments

4.1 Overview

A Secure Operating Environment (SOE) is a component, either software or hardware which can be seen as a black box and which provides high tamper resistance. According to ITSEC (Information Technology Security Evaluation Criteria) which defines the criteria that has to be fulfilled by secure systems, we can define a SOE as a secure component which fulfills the following criteria:

- *confidentiality*: prevention of the unauthorized disclosure of information.
- *integrity:* prevention of the unauthorized modification of information.
- *availability:* prevention of the unauthorized withholding of information or resources

Softwares such as Microsoft Windows Media Player [80] or systems such as iTune [7] can be seen as SOE since they provide some guarantees on the criteria aforementioned. However, with the ever-increasing number of digital frauds, many vendors turned towards the use of hardware SOE which offer a higher level of security. Hardware SOE became popular thanks to smartcards which are detailed in the next subsection. Moreover, other devices such as the crypto processor 4758 developed by IBM [65] offers the highest level of security and are used in the banking industry to secure transactions (e.g. ATM). As the ARM microprocessor manufacturer (PDA, smartphones) stated [9], the operating systems have become too large to be trusted and proven. Consequently, any software running on top of them are likely to have security flaws. That leads ARM to develop TrustZone [9] which adds

some secure extensions to the ARM microprocessors to provide a secure operating environment available to any applications. This was a step towards secure computing as defined by the Trusted Computing Group [112] founded by the main hardware and software vendors Intel, Microsoft, IBM, AMD, HP, etc.; their purpose was to secure any architecture by adding a secure chip.

The solution proposed in this thesis can accommodate several SOE. In order to validate our solution, we choose to use smart cards as SOE to make our experiments since they are representative of the SOE technologies. Indeed, they have been massively used over the two last decade and proved to be a versatile (used in the telecom, banking, pay TV, authentication industries) and a cost effective technology (few dollars). Moreover, all the SOE shared to some extents the same principles and the same constraints. In the following, we give a brief overview of the smart card technology.

4.2 Smart card overview

The smart card was patented in 1974 by Roland Moreno, a French inventor. It consists of a credit card size plastic token within which a microchip has been embedded; the microchip being the intelligence of the smart card. The first generation of smart card chips were *memory-only* chips used to offer prepaid phone calls. The generation of microprocessor chips came next and lead to the design of crypto cards which have built-in cryptographic modules.

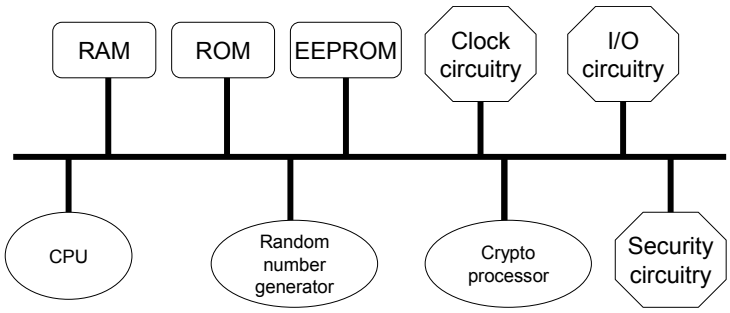


Figure 18. Smart card architecture

The smartcard market is now booming and is now represented essentially by European groups which are present all around the world. The main microchip manufacturers are ST Microelectronics, Infineon and Philips and the main smart card manufacturers are Axalto, Gemplus and Giesecke & Devrient. New analysis from Frost & Sullivan [56], World Smart Card IC Markets, reveals that revenues in this market totaled $2.1 billion in 2004 and projects to reach $4.1 billion in 2010 especially generated by the Telecom (SIM cards from the GSM industry) and Financial Services (banking cards).

4.2.1 Smart card environment

At the present, advanced smart cards include a 32-bit CPU running at 33Mhz, 96 KB of EEPROM as stable memory, 4 KB of RAM and security modules (e.g., cryptographic processor, random number generator) as represented in Figure 18. The ROM is used to store the operating system and the RAM to run system and users applications. The EEPROM stores the keys and the applications. The throughput which was originally of 9600 Kbs for the first generation of smart cards can reach hundred of Kbs for newer smartcards thanks to the USB connectivity (e.g., ST22 which has a 440 Kbs communication speed [106]). The communication between the smart card and a terminal is half-duplex and is initiated by the terminal. Data are exchanged through APDU (Application Protocol Data Unit) of 256 bytes. To hide the APDU communication complexity, protocols have been implemented on top of this layer, including RMI, .NET, HTTP.

4.2.2 Tamper resistance

We present in this paragraph the different attacks that can be conducted by hackers and the different techniques which have been found to counter them. These attacks can be classified in four categories as follows [73][126]:

- *Logical attacks.* They consist mainly in finding hidden commands left from a previous installed application, provoke a buffer overflows, insert a malicious software in the smartcard to gain access to unauthorized data (e.g., Trojan horse) and take advantage of flaws in cryptographic libraries. These attacks are countered first, by guaranteeing the correctness of the applications (user applications and system libraries) using formal methods and second, by providing a firewall mechanism to separate memory space of the applications.

- *Physical attacks.* They are performed by opening the package, access the chip surface with semiconductor test equipment, observe and manipulate the internal data. These attacks are conducted using chemicals and very expensive optical devices to get a precise map of the EEPROM. To combat these attacks, the size of transistors and wires are reduced to nanometers to make them invisible for advanced optical devices. In addition, protective layers and sensors are added to erase the memories when an intrusion is detected.

- *Eavesdropping.* It consists in getting access to protected information by analyzing emanated electromagnetic radiation and protocol timings (which may reveal the operation in progress). To face these attacks, timing noise is introduced and the chip is equipped with metal shield to lower the electromagnetic radiation.

- *Fault injection.* Because the smart card is not autonomous and has to rely on an external power supply and clock, a hacker may try to generate fault on purpose by supplying voltage variations and spikes, and clock phase jumps. These attacks can be prevented thanks to a voltage sensor and balancing circuits to regulate the input power and the clock.

4.2.3 Discussions on the limitations

The volatile memory is obviously a limiting factor. Current smart cards have a RAM capacity of 4 KB but most of it is used by the operating system, so only a few amount of RAM is left to the application. It is likely that the RAM capacity will not evolve for two reasons. First, a byte of RAM requires around twice the physical space as a byte of EEPROM which in turn requires around twice the space of a byte of ROM [98]; Since the size of the chip is limited by the ISO7816 standard [69], increasing the RAM means reducing the capacity of the EEPROM and ROM on the chip. Second, smart cards are often used to perform short operations such as identification and secure transactions and to support fast connection/disconnection. Because, the smart card is not autonomous, the RAM has to be reported to EEPROM before disconnections. In this case, it is more interesting to write directly on EEPROM.

Regarding the EEPROM, its capacity is increasing and the EEPROM technology is now being replaced by the Flash technology which provides bigger capacities of memory (MB). However the stable memory EEPROM or Flash have both slow write operations and thus cannot palliate the lack of RAM.

4.3 Summary

We gave an overview of SOE which are a trusted components which offer a high tamper resistance to attacks. We described the smart card environment since it is now an off-the-shelve technology present in various contexts, making it suitable for our experiments. The main limitations of the SOE are the RAM capacity and the throughput. In the future, the main limitations will still be the RAM and probably the cryptographic function. In both cases, the challenges are on designing algorithms minimizing the consumption of RAM, and techniques to reduce the volume of data to be processed in order to reduce both the communication and decryption costs.

5 Background on access control

The three main requirements of a secure sharing system are to guarantee the confidentiality of data (prevent from disclosure of unauthorized information), the integrity of data (prevent from tampering of data), and the availability of data (prevent from denial of service). An access control is a means of obtaining data confidentiality specifying which users can perform which actions on which data. Many access control models have been defined so far to simplify and secure the management of the access control rights. The three most famous models are as follows: Discretionary Access Control (DAC), Mandatory Access Control (MAC), and more recently Role Based Access Control (RBAC).

According to TCSEC [45], Discretionary Access Control (DAC) is defined as "a means of restricting access to objects based on the identity of subjects and/or groups to which they belong. The controls are discretionary in the sense that a subject with a certain access permission is capable of passing that permission (perhaps indirectly) on to any other subject (of the organization)". In this model, we can consider that the organization delegates parts of his access rights to each subject [37]. However, the organization generally regulates the way a user is authorized to pass the access rights on to the other subjects.

MAC [14][37] is generally used in multilevel systems that process highly sensitive data such as classified and military information. In this context, we can suppose that all the information belongs to an organization which determines the access that is to be granted. To this end, MAC mechanisms assign a sensitivity level to all information and assign a security clearance to each subject. The subject clearance level specifies its level of trust. An object's sensitivity specifies the level of trust required for access. In order to access a given object, the subject must have a clearance level equal to or higher than the requested object sensitivity. Such a system prevents an authenticated user or process at a specific classification or trust level to access information, processes, or devices in a different level. This provides a containment mechanism of users and processes, both known and unknown (an unknown program would be an untrusted application where device and file accesses should be monitored and/or controlled). This allows to avoid some problems incurred by the Discretionary Access Control systems where a subject may by accident or malice give access to unauthorized subjects.

In Role-Based Access Control (RBAC) [89], the permission to perform certain operations (permissions) are assigned to specific roles. These operations are based on individual's roles and responsibilities within an organization. The process of defining roles is usually based on analyzing the fundamental goals and structure of an organization and is usually linked to the security policy. Since users are not assigned permissions directly, but only acquire them through their role(s), the management of individual user rights consists in assigning the appropriate roles to the user.

Although MAC offers a high level of security, it is too rigid and cannot handle exceptions (e.g., a user with a certain level of security cannot exceptionally have an access to a data requiring a higher level). RBAC provides a way to define access rights just by assigning roles. For the sake of simplicity we choose to rely on the DAC model in our study.

6 Conclusion

We gave an overview of the XML standard and the different techniques to query and process XML streaming data. Then we showed how cryptography can guarantee the confidentiality of data and protect them from different kind of tampering. Then, we gave a description of *Secure Operating Environment* to enforce security features. Finally, we

provided a brief description of some access control models.We addressed in particular, streaming XML data because of the limitations of the SOE (few RAM and low throughput) and will see in chapter 4 how we comply with this constrained environment.

Chapter 3 – From XML access control models towards secure evaluation

1 Introduction

As XML has become a de facto standard to exchange data, the need for the definition of a universal XML Access Control Model to regulate access to XML data has become a necessity. That led many researchers to propose different models which consider various issues for XML access control. In the meantime, few solutions were proposed to guarantee that these models are enforced properly, i.e. guarantee that users have only access to what they are authorized. In the following, we give an overview of the different XML access control models which have been proposed up to now. Then we address the problem of the evaluation of the access control, first on a secure server as it was done traditionally and finally on an insecure environment such as the Internet.

2 XML access control semantics

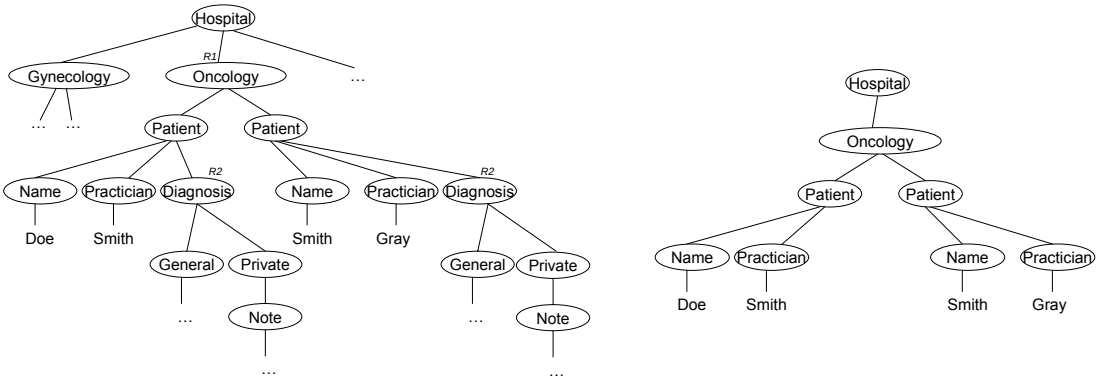
Several authorization models have been proposed for regulating access to XML documents. Most of these models follow the well-established Discretionary Access Control (DAC) model [18][39][58]. We introduce in this section a simplified access control model for XML, inspired by Bertino's model [18] and Damiani's model [39]. Subtleties of these models are ignored for the sake of simplicity. Then we provide the different variations of this model.

2.1 Basic model

In this basic model, access control rules, or access rules for short, take the form of a 3-uple <*sign*, *subject*, *object*>. *Sign* denotes either a permission (positive rule) or a prohibition (negative rule) for the read operation. *Subject* is the user or the group of users for which the rule applies. *Object* corresponds to elements or subtrees in the XML document, identified by an XPath expression. The expressive power of the access control model, and then the granularity of sharing, is directly bounded by the supported subset of the XPath language. In this thesis, we consider a rather robust subset of XPath denoted by $XP^{\{[],*,//\}}$ [83]. This

subset, widely used in practice, consists of node tests, the child axis (/), the descendant axis (//), wildcards (*) and predicates or branches […]. Attributes are handled in the model similarly to elements and are not further discussed.

The cascading propagation of rules is implicit in the model, meaning that a rule propagates from an object to all its descendants in the XML hierarchy. Due to the propagation mechanism and to the multiplicity of rules for a same user, a conflict resolution principle is required. Conflicts are resolved using two policies: *Denial-Takes-Precedence* and *Most-Specific-Object-Takes-Precedence*. Let assume two rules *R1* and *R2* of opposite sign. These rules may conflict either because they are defined on the same object, or because they are defined respectively on two different objects *O1* and *O2*, linked by an ancestor/descendant relationship (i.e., *O1* is ancestor of *O2*). In the former situation, the *Denial-Takes-Precedence* policy favors the negative rule. In the latter situation, the *Most-Specific-Object-Takes-Precedence* policy favors the rule that applies directly to an object against the inherited one (i.e., *R2* takes precedence over *R1* on *O2*). Finally, if a subject is granted access to an object, the path from the document root to this object is granted too [39] (names of denied elements in this path can be also replaced by a dummy value [53][59]). This *Structural* rule keeps the document structure consistent with respect to the original one. The resulting view will then be like the original document but with omitted branches (e.g., Figure 19.b) and can have some nodes replaced by dummy nodes [53][59].



(a) Original XML document (b) XML view

Figure 19. Example of access control

The set of rules attached to a given subject on a given document is called an *access control policy*. This policy defines an authorized view of this document and, depending on the application context, this view may be queried. We consider that queries are expressed with the same XPath fragment as access rules, namely $XP^{\{[],*,//\}}$ [83]. Semantically, the result of a query is computed from the authorized view of the queried document. Thus, predicates cannot be expressed on denied elements even if these elements do not appear in the query result. Indeed, if it was the case, it would have been easy to infer information (e.g., if only the ages of the patients are granted, then we can find that a certain patient is in a hospital simply by issuing a

query using the name of a patient in a predicate). However, access rules predicates can apply on any part of the initial document.

An example is given in Figure 19 for a medical document. The access control for the secretary of the *Oncology* department is defined as follows:

```
R1: <+, Smith, //Oncology>
R2: <-, Smith, //Patient/Diagnosis>
```

This access control policy gives her the right to read information on the Oncology department but deny access to diagnosis. The closed access control policy forbids her access to other departments. The rule *R2* applies at a lower depth than *R1* and takes precedence because of the *Most-Specific-Object-Takes-Precedence* policy. The resulting authorized view is depicted in Figure 19.b. In this view, we apply the *structural* rule which forces access to the path from the root to authorized elements in order to keep the view consistent with the original document.

2.2 Variations of the model

In this subsection, we give the different features provided by other models even though they may not rely strictly speaking on the same semantics.

Access control on the schema. Damiani et al. [39] extended the model so that access rules can be defined on DTDs. In this case, all the documents conforming to the schema inherits from these rules.

Explicit conflict resolutions. Bruno and Gabillon [58] defined a model adding numeric priority to resolve conflicts when multiple rules apply to the same object. Some others propose automatic conflict resolution giving precedence to a certain class of rules. For instance, Murata et al. [85] defined the negative rules to take precedence over any positive rules, and the XACML [66] developed by IBM lets the administrator decides if the negative or positive rules take precedence.

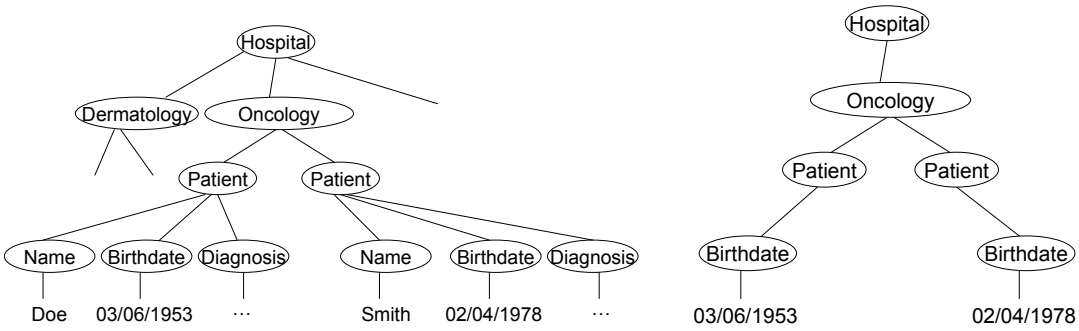
Explicit propagation. Many of the works on XML access control consider different kind of propagations which can be (1) *local:* the rule applies only on the *object*; (2) *cascade:* the rule applies on the *object* and its descendants; and (3) *one-level*: the rule applies on the object as well as on its child elements.

Update operations. In [17][74], the authors extended the basic model to consider access control on update operations. The following updates are considered: insertion before and after/deletion/update of an element/attribute. In addition, they also considered update operations on the DTD.

Security views. In the basic model, the structure of the resulting view remains consistent with the original document. There are pro and cons in doing this. By keeping the structure, it is more likely that an application which operated previously on the original document will

still work on the document view considering only minor changes. However, there are some cases where the structure may reveal information using inference analysis. Let us consider the medical XML document depicted in Figure 20.a. Suppose that Bob has been given access to all the birthdate elements to make some statistics on the patients of the hospital (rule *R1*). The resulting authorized view is depicted in Figure 20.b. In the basic model, the tags along the path connecting the birthdate element to the root element are visible in the view (as depicted in Figure 20) or replaced by dummy nodes. In the first case, Bob can easily see that the Patient who was born on 03/06/1953 has a cancer because it is found under the element *Oncology*. Crossing this information with another database such as an electoral list, he may infer that a certain Mr. Doe has a cancer. In the second case, which uses dummy nodes, then the cardinality of the elements and their structure would have probably led to the same conclusion. This problem is addressed in [53][54][84][108] by extending access control to relationships between elements. In these models, the structure of the view can be different from the original document (e.g., attach all the patient elements to the hospital element in the view) to limit inference on the structure.

Access control:
R1: <+, Bob, //Birthdate>



(a) Original document **(b) Authorized view**

Figure 20. Illustration of inference from an XML view

Provisional access. Kudo and Hada [72] addressed the problems of provisional access. Access rules are augmented using conditions including: (1) access is granted if the user accepts that her access are logged, (2) access is granted if she agrees to sign (digitally) a terms, (3) access is granted but the data are to be encrypted and (4) if unauthorized access is detected, a warning message is sent to the administrator. These conditions are particularly useful in the case of emergency: a nurse may be granted exceptional access to the confidential medical folder of a patient provided that all her actions are logged. Moreover, it may be useful in a DRM application, e.g., one may be allowed to see a bonus scene of a movie provided she previously accepted to fill up a survey.

Knowledge based access. In [82], Miklau and Suciu defined an access control based on the

knowledge of the XML document. For instance, in a medical document, one may get access to the address of the patient only if he can provide his name and his social security number.

Formal model. Fundulaki and Marx [57] tried to unify all the access control that have been defined so far using formal specifications based on XPath. Their idea is to define a generic formal filter which tells the conditions for a node to be authorized (e.g., no negative rule should have been defined over an ancestor).

Emerging standards. Even though emerging standards are not new models by themselves, we give a brief description of them. These standards have been motivated by DRM vendors and e-commerce industries willing to have a common definition of access control. The two major standards are XACML [91] initiated by Oasis, and XrML [127] by Microsoft and ContentGuard. They tried to compile in their language all the requirements that can be needed in an access control solution.

We provided in this section a basic access control model which will serve as a reference in the sequel of this thesis. Then we gave an overview of the variations of this basic model. In the following, we see how these models can be enforced on secure servers as in traditional approaches, and on insecure infrastructure.

3 Traditional evaluation of the access control

Traditionally, the access control is evaluated on the trusted server which hosts the data. In order to provide for every user his authorized view according to his privileges, the first idea which comes to mind, is to materialize a personalized view for each of them. However, such solution is not viable as soon as we consider a large number of users and/or updates. Indeed, whenever a data is updated, all the views have to be recomputed. Moreover, if an access right is updated, the corresponding view has to be reconstructed. While, incremental updates on the views may be considered, the space overhead remains. In the following, we explore two different approaches. First, an approach based on DOM labeling which computes for each user an accessibility map based on the access rights. Second, we present an approach which rewrites queries according to the access control policy of the user to form a new query which is run against the original document.

3.1 DOM approach

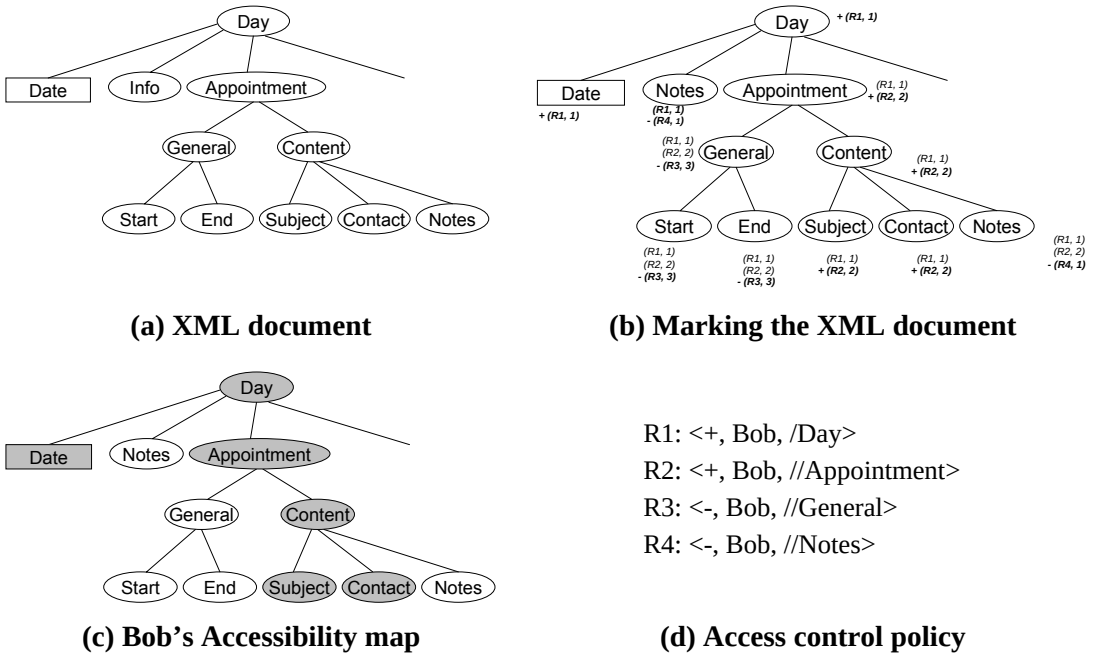
The first solutions [18][39][40] to evaluate the access control for XML document are based on the DOM approach. The basic idea is to provide for each element of an XML document an accessibility map that tells the set of authorized users.

3.1.1 DOM labeling

We describe in the simplified algorithm the different steps to create the accessibility map for a given user. Extending this algorithm to support many users is trivial.

1. For every rule: identify elements matching the rule and mark them with the couple *(ruleId, depth)*; *depth* denoting the depth of the element.
2. For every matching element *e*, propagate down the rule by marking every elements of its subtree with the *(ruleId, depth)* of element *e*.
3. For every node, consider the couple with the highest depth (or the negative rule in case of equality). If the rule is positive, then mark the node as '+' and '–' otherwise.
4. Mark all the ancestors of nodes marked '+' with a '+' (force access to ancestor nodes to apply structural rule).

Figure 21. DOM labeling algorithm



(a) XML document **(b) Marking the XML document**

R1: <+, Bob, /Day>
R2: <+, Bob, //Appointment>
R3: <-, Bob, //General>
R4: <-, Bob, //Notes>

(c) Bob's Accessibility map **(d) Access control policy**

Figure 22. View construction process

This algorithm is illustrated in Figure 22.b. The computation of the accessibility map is built for Bob according to access rules defined in Figure 22.d. from a document owned by Alice containing her personal calendar. For the sake of clarity, we omit the text nodes. At the end, all the nodes marked with a '+' are authorized to the user (see Figure 22.b) and result in an accessibility map (see Figure 22.c) which tells the authorized elements (in gray in the figure) for a user.

In the following we give a brief overview on how the authorized view can be computed using DOM labeling as well as an alternative solution. Then we explain the different solutions to evaluate queries using DOM labeling.

Authorized view computation

In order to return to the user the content of the whole authorized view, the server constructs a document which includes all the elements marked with a '+' and returns it to the user. However, such solution can be quite costly, because every elements of the tree has to be checked.

In order to tackle this problem, Bruno and Gabillon [58] proposed to compute the view using XSLT. They generate an XSLT stylesheet for each user according to her access rights. Whenever she requests a document, the original document is processed by the XSLT processor using the XSLT stylesheet and is delivered to the user. The advantage of such approach is that only roots of subtrees which matches a rule and not all the elements of the subtrees are checked, leading to good performance. This solution has the advantage to rely on existing technologies and to benefit from their optimizations. However, even if it performs well to compute the complete authorized view, it does not support query evaluation.

3.1.2 Dealing with queries

The evaluation of a query is particularly complex to manage in the DOM labeling context. Indeed, a special care has to be taken considering that semantically, the query returns the same result as if it was issued on the authorized view. This consideration in mind, predicates cannot be expressed on denied elements even if these elements do not appear in the query result. To implement this, when a user issues a query, all the elements, attributes and text nodes which are touched by a query are checked. Therefore, accessibility to an element is authorized if in the labeled DOM tree, not only the elements/attributes/text elements part of the result are marked '+' but also all the elements touched by the predicates. For instance, the evaluation of query *//a[b > 5]/*c will first identify all the elements *c* marked '+' which are child of element *a* and which have a sibling $b > 5$ which is also marked '+'. In this case, all the descendants marked '+'of the identified elements *c* are authorized. Needless to say that this process can be dramatically costly.

To reduce this overhead, Yu et al. [128] proposed to use a *Compressed Accessibility Map*. Their idea is as follows: rather than keeping the complete set of authorized elements (those marked with a '+') and unauthorized elements, they only keep the set of a few crucial nodes. In these approaches, they identify four types of so-called crucial nodes: (1) root of a complete forbidden subtrees, (2) root of a complete authorized subtrees, (3) root of authorized subtrees containing forbidden parts, and (4) root of forbidden subtrees containing some authorized parts. The case (1) and (2) are particularly interesting and can improve significantly performance. Indeed, if an element of a query is found to be in case (1) (resp (2)), then so are all its descendants which need not be checked.

Kudo and Hada [72] studied cases where this checking process is not necessary. They

basically proposed a solution to detect if the whole query result is authorized to the user. If so, the query is executed directly on the XML repository. Conversely, if none of the data requested by the query are authorized, an empty answer is returned. The detection is based on static analysis of XPath containment between those of the query and those of the access control rules. For instance, if we consider the positive rule */a* and the query */a/b*, then the query is completely authorized to the user and can be evaluated directly on the XML repository. Although this solution may improve performance in some cases, it has two limitations. First, the time-complexity of running containment algorithms statically is very high, and second, there are many cases, where the issue is unknown. For instance if there is a positive rule *//a* and another negative rule *//b*, the issue will always be unknown.

3.2 Query rewriting techniques

As said earlier, the main problems when dealing with DOM document are that it consumes a high amount of RAM and that the construction of the DOM tree can be rather slow in the case of big document. Several works consider to use a normal XML repository, and to rewrite queries according to the user's privileges. This way no extra cost is incurred by the access control evaluation.

QFilter [75]. The authors extended the work of Murata and Kudo [85] to rewrite queries in a way that it filter out forbidden parts. In their approach, negative rules take always precedence. Their basic idea is to rewrite the query as follows:

$$Q' = Q \text{ INTERSECT } (\cup \text{positive rules}) \text{ EXCEPT } (\cup \text{negative rules})$$

In this approach, the resulting query *Q'* returns all the nodes matching the query which are also parts of the authorized view. However, this query does not return semantically the same result as the query *Q* would produce from the authorized view. The reason is that they did not consider the case where a predicate is posed on an unauthorized element.

The authors of QFilter [75] introduced also a post-processing phase. The idea is to execute the query as is on an XML repository and to apply the access control on the result. However, in this case, a user can easily exceed her rights. For instance, let us consider a document containing all the information about the employees of a company, that is their name, salary and other fields. The access control only authorizes Alice to see all the salaries of all the employees but no other information. When Alice poses the query: *//Employee[name='Bob']*, the XML repository returns the record *Employee* related to Bob and the post-processing phase will keep Bob's salary. This obviously violates the original access control which only allows Alice to get the list of salaries of all the employees but not to put a name on them. A post-processing phase is thus to be precluded.

Wang et al. [117] provide a solution based on the use of relational databases. The XML document is stored in a relational database and when a user issues a query, it is rewritten into

an SQL query. This query integrates the access control of the user to be executed on the relational databases. The resulting XML view is then computed from the relational data result. However, in their solution, nothing is said about the correctness of the approach (e.g., if predicates can be executed over forbidden elements to infer information).

Fan et al. [53] consider security views created from a DTD augmented with transformation rules. In their model, they define for each user a different security view which can have a different structure from the original document. Based on this model, they provide a way to rewrite XPath queries posed on the security view into a new XPath query executed on the original document. However, the expressiveness of the security annotation is limited to hiding node values and not to complete subtree. This limitation is incurred by the XPath expressivity. Indeed, queries can only be posed on values and not on subtrees since XPath does not provide features to restructure a subtree according to the security view.

Mohan et al. [84] alleviated this limitation. Their solution, also based on security views, consists of rewriting an XPath Query into an XQuery statement. Compared to an XPath query, an XQuery statement is more expressive (e.g., returning a subtree without an inner subtree is possible in XQuery but not in XPath) and allows to restructure the result to make it compliant with the security view.

We presented two different families of techniques to evaluate access control policies on XML documents. First, DOM labeling solutions based on a global representation of the documents associated to an accessibility map. This map tells for each parts of the documents, the set of users who are authorized to access to it. These solutions which were initially space and time consuming are improving to reduce these overheads. Second, query rewriting techniques which rewrite a user query according to her access control policy. These solutions are more flexible considering expressive language such as XQuery. Moreover, they enabled to consider more complex access control models based on security views.

4 Encryption based access control

The solutions described in the previous section are not suitable when it comes to share very sensitive information such as medical information and industrial secrets. Indeed, all the data stored on the server can be easily read by an untrustworthy database administrator or by an intruder who gains access to the database. To face this attack, many solutions rely on encryption.

The first generation of solutions aimed at securing the database footprint on the server by means of encryption. The data are encrypted on the server using different encryption keys. Whenever a user issues a query, she provides the decryption key to the server which decrypts the data and evaluates the query. However, if encryption provides an effective

answer to attacks conducted on the database footprint by an Intruder, it does not enforce data confidentiality on its own. Indeed, the server being still responsible for query execution and access right management, encryption makes just a bit more tedious the Administrator attacks. Thanks to her privileges and to the DBMS auditing tools, the DBA can change the encryption package, can get the cryptographic keys, can modify the access right definition and can even snoop the memory to get the data while it is decrypted. Thus, as Oracle confesses [97], encryption is not the expected "armor plating" because the DBA (or an Intruder usurping her identity) has all privileges.

To alleviate this problem, client-based solutions have recently emerged as an alternative. In these solutions, not only are the data encrypted on the server but also decrypted on client devices. In this case, because the data are never decrypted on the server, even if the server is malicious or hacked, the data are not compromised. In the following, we explore the different encryption schemes based on this idea for XML document.

4.1 Direct encryption

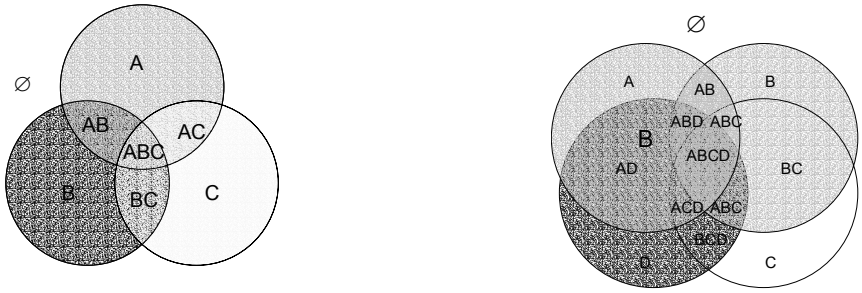
Direct encryption refers to methods translating an access control policy applied to an XML document into a collection of XML fragments and encryption keys such that: (i) a partition of the document is defined according to the set of authorizations (i.e., positive or negative access control rules) forming the policy, (ii) each fragment resulting from this partition is encrypted with a different key, (iii), each subject receives the keys needed to decrypt the fragment he/she is granted access to. The Author-X [18] framework is representative of this approach. It considers a publish/subscribe model where encrypted XML documents are pushed towards subscribers. The encryption scheme follows an element-wise encryption (i.e., tags, attributes and values are encrypted in place in the document with an element granularity) as pictured in Figure 23.

```
<aRD2S342>
      .
      .
  <dz3212454 z3FZES32=”23242DEZ”>
    <SD45EFR5>
      <vG3456GT ertd4323=”432R2112”>
        <zeFEzdez>FD342134343</zeFEzdez>
          <dsSEf34S>
            lferfe3FKFKEZ2JFDJSFJD
            csdcqdopoer343RSDFDSFD
          </dsSEf34S>
      </vG3456GT>
    </SD45EFR5>
  </dz3212454>
      .
      .
</aRD2S342>
```

Figure 23. Element-wise encryption model

The decryption keys can be provided to the subscribers in different ways [16] (e.g., through an LDAP directory or within the document itself, encrypted with the public key of each

subscriber, or sent separatedly, encrypted with the public key of the subscriber). A potential problem with this approach is the number of keys to consider. Indeed, this number may grow exponentially with the number of users. This is illustrated in Figure 24 for 3 and 4 users. When considering *n* users, we have to take into account all the different kind of sharing that can occur between the users (e.g., data shared by A and B, A and C, A and B and C, etc.). Each data shared by a different set of users is to be encrypted with a different key (including the data accessible by nobody). While this approach proposed a practical solution, it may consider in the worst case 2^n keys. This is exemplified in Figure 24.



(a) 3 users (b) 4 users

Figure 24. Key distribution problem

4.2 Encryption with compatible keys

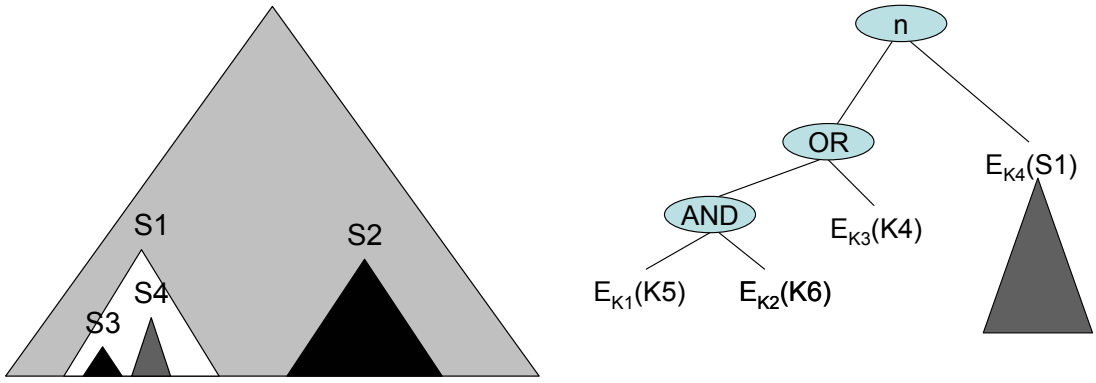
The preceding issue (number of keys) can be solved using hierarchical keys and an encryption scheme allowing the same encrypted data to be decrypted using different keys. This problem has been addressed in several papers [3][15][19][99][100]. In most of them, the intuition is the following. Subjects and objects are grouped into security classes such that a subject having a clearance level L_1 is granted access to all objects classified at level L_2, assuming that $L_2 \preceq L_1$ (i.e., L_1 dominates L_2). A pair of asymmetric keys (Ki, Ki^{-1}) is generated for each class *Ci*, such that an object encrypted with *Ki* can be decrypted with any Kj^{-1} / $i \preceq j$. This approach requires a single key per subject if a total order exists among the security classes and a few numbers of keys otherwise. However, these solutions impose a rigid (MAC-style) administration of access rights. More recently, Ray et al. [99] developed a new approach called compatible keys which removes the hierarchical constraints proposed in the preceding solutions in the context of data publishing. In this approach, each user has one key to decrypt any data for which she has access. This makes the concept of compatible keys more flexible and allows an easier management of keys. However, their compatible keys scheme relies on a costly asymmetric encryption (roughly three orders of magnitude slower than symmetric encryption). This problem is alleviated by Bertino et al. [15] who proposed a new form of compatible keys based on symmetric encryption. In their approach which integrates temporal constraints, the number of compatible keys depends solely on the number of access control policies. Unlike the previous approach, the user does not manage keys but information that will allow him to generate the proper decryption keys to get access to the

authorized data. Even though these methods facilitates for the user the management of keys (one key or key information per user), it does not simplify the global encryption scheme which still needs to consider the same total number of keys (including derived keys) than in previous solutions.

4.3 Super encryption

Miklau and Suciu [1][81][82] proposed another encryption scheme based on super-encryption (i.e., recursive encryption of the same data with different keys) as depicted in Figure 25.a (to access *S4*, one first needs to decrypt *S1* and then *S4*). Inner keys are used to encrypt subparts of the document and are themselves embedded in the document. Inner keys are encrypted with user's keys or provisional information (e.g., birthdate, social security number) and can be combined together (e.g., XORed) to form a new key corresponding to a – potentially complex – logical expression. For instance, the access control can be specified so that one needs *K1* and *K2* or simply *K3* to get access to a subtree *S1* in Figure 25.b. This is achieved by encrypting the subtree *S1* with a key *K4* and by using extra nodes next to *S1* as follows: a node containing a key *K5* encrypted with *K1*, another containing the key *K6* encrypted with *K2* and the last one containing the key *K4* encrypted with K3 such that K4=K5 ⊕ K6, where ⊕ denotes the XOR operator. If the user has got the key *K3*, he will have access to the key *K4* and so decrypt the subtree *S1*. In the same way, if he has the key *K1* and *K2*, he will have access to *K5* and *K6* and compute *K4* and so get access to *S1*.

By this way, logical conditions to access the data can be directly compiled into the encryption process. When receiving a document, a user decrypts the subparts of it he/she is primarily granted access to and can keep decrypting the following subparts recursively as long as he/she gets the proper decryption keys.



(a) Super encryption model **(b) cryptographically based condition**

Figure 25. Miklau and Suciu model

This solution provides an elegant way to implement complex conditions and provisional access, and relies on a simple key distribution. However, it suffers from

important limitations in our context. First, the cost incurred by super-encryption and by the cryptographic initialization of inner keys makes this solution inappropriate for devices with low processing capacities. Second, as no compression is considered, the space overhead incurred by the XML encryption format and inner keys can be important. Finally, as Geueur-Pollman [60] outlined, a user can be aware of subtrees for which he does not have the decryption keys and can infer information. In order to alleviate this issue, Geueur-Pollmann introduces the XML pool encryption which principle is to append to an additional node in the tree, all the nodes which have to be encrypted. These nodes are encrypted separately and associated to information to integrate them properly in the document after decryption.

4.4 Query-aware encryption

The previous solutions do not perform well when a user is interested in (or is granted access to) a small subset of the document. Indeed, there is no indexation structure to converge towards relevant parts of the document wrt. a potential query and/or the access control rules. The idea developed by Carminati et al. [29] is to delegate part of the query evaluation to an insecure server hosting the encrypted data. To this end, element-wise encryption is considered. A query on the XML structure can be processed easily by the server, encrypting in place tags and attributes in the XPath expression (e.g., */a/b* can be evaluated on the encrypted data as */E(a)/E(b)* where *E* is the encryption function). Selection on values are tackled by index partitioning in a way similar to the work developed by Hacigumus et al. [62]. Their solution is to append to each encrypted value a plaintext index value telling to which interval the value belongs to (the bounds of the intervals remain hidden to the server). This allows a coarse filtering on the server side subsequently refined by the client after data decryption.

Delegating computation on an untrusted server requires enforcing the integrity of the result, the most difficult issue being checking the completeness of the result. This problem is solved to some extents by Devanbu et al. [42] who rely on Merkle Hash Tree [78] built on the queried document. However, they consider completeness only for complete subtrees, thereby precluding the use of negative authorizations in an access control policy. Carminati et al. extended the Merkle Hash Tree towards an XML hash tree by considering for each internal XML node, a hash value built from its tag name, its content and the hash of all its children nodes.

In our resource-limited context, this solution suffers from two overheads. First, the encoding scheme may incur a significant space overhead, considering that tags and values have to be padded at encryption time (e.g, 3DES and AES produce respectively 64 bit and 128-bit blocks). Index and schema information contribute also to this space overhead. Second, extending the Merkle Hash Tree which originally operates on binary tree incurs an important overhead: when requesting an element having *n* siblings, their *n* hashes are sent back along with the answer (SHA-1 produces hash of 20 bytes).

4.5 Limitations of existing approaches

While these methods offer several interesting features, they do not perform well in the context considered in this thesis. Indeed, by compiling the sharing scheme into the encryption scheme, subparts of the data need be re-encrypted whenever the access control policies change. This problem is illustrated in Figure 26. Initially, there are 3 users who have different access on the data. Subpart AB is accessible only to user A and B while subpart C is only accessible to user C. When a user is removed access to a data (e.g., user A), all the data which were previously accessible to this user need be reencrypted to comply with the new sharing scheme. This problem gets even more complex when a user is granted access to new subparts of the data. In this case, they will have to be encrypted with new encryption keys (a different key for each intersection with existing regions) which would have to be distributed to the users accordingly. This process which is costly in the direct encryption is even worse when considering super-encryption. Indeed, all the outermost subtrees of a data affected by the access right changes are reencrypted.

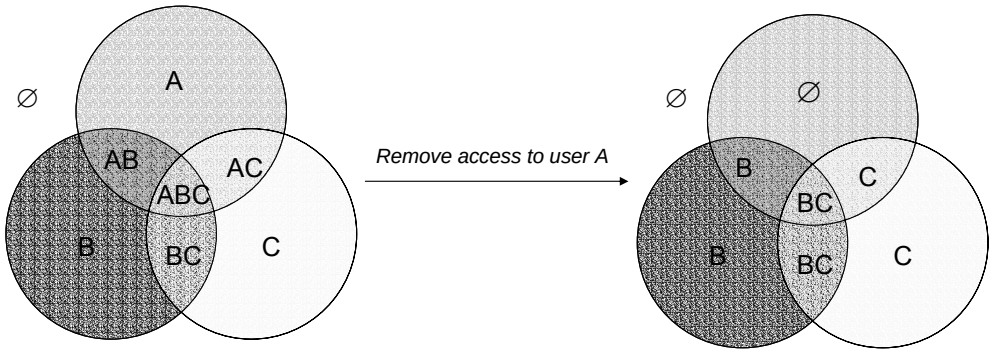


Figure 26. Access right removal problem

These limitations lead to two major restrictions. First, the access rights may not be updated frequently and, second, the access rights may not be fine grained and personalized without resulting in a complex encryption scheme management and costly operations. Indeed, as all the data shared by different users have to encrypted using a different key (or a combination of keys), then the finer is the access control, the more complex is the encryption scheme.

5 Conclusion

In this chapter, we made an overview of the access control that have been defined for XML and explained the techniques to enforce them, considering secure servers then insecure infrastructure. While these solutions solved to some extents the problem of confidentiality, they do not support well dynamic rights and personalized and fine-grained access. This is an inherent problem of access right models which compile access rights in encryption. In the

next chapter, we propose a complete solution to separate the access control from encryption thanks to *Secure Operating Environments*.

Chapter 4 – Secure evaluation of personalized and dynamic access control policies

1 Introduction

Existing solutions relying solely on encryption failed in providing dynamic and personalized access control for XML. These features are however of utmost importance in various contexts. In the medical context, a nurse may be granted temporary access to very sensitive data of a patient in the case of emergency. As these data are protected by the need-to-know principle which is to give only the necessary information to a user, access control policies have to provide fine-grained and personalized access (e.g., researchers can have access only to some limited results of patients who have subscribed to a protocol, doctor can have access only to folders of patients he has treated, etc.). In collaborative work applications, access rights may evolve as users make new relations. In the DRM context, a special access may be granted for a limited amount of time and have to provide personalized access for special classes of persons or depending on the situation (special offers).

We propose in this chapter an integrated solution based on a *Secure Operating Environment* to enforce the access control policies (basic model as described in Section 2.1 of Chapter 3) on the client device with the benefit of separating the encryption scheme from the access right policies in order to enable personalized and dynamic access control rights. Most of the elements of this chapter are taken from a work published at VLDB 2004 [22] and another submitted for publication in an international journal.

1.1 Target architecture

As stated in Chapter 1, SOE can rely on software or hardware elements. However, only hardware-based SOE provide strong anti-tampering guarantees. In the sequel of this chapter we make no assumption on the hardware SOE, except the traditional ones: 1) the code executed by the SOE cannot be corrupted, 2) the SOE has at least a small quantity of secure stable storage (to store secrets like encryption keys), 3) the SOE has at least a small quantity of secure working memory (to protect sensitive data structures at processing time). In our context, the SOE is in charge of decrypting the input document, checking its integrity and evaluating the access control policy corresponding to a given (document, subject) pair. This

access control policy as well as the key(s) required to decrypt the document can be permanently hosted by the SOE, refreshed or downloaded via a secure channel from different sources (trusted third party, security server, parent or teacher, etc). The architecture is depicted in Figure 27.

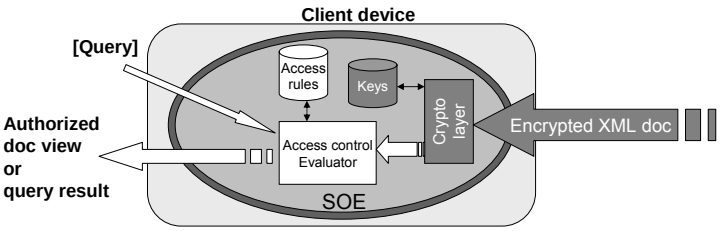


Figure 27. Abstract target architecture

The SOE being the unique element of trust, the access control rules evaluator must be embedded within the SOE and lead us to make the following considerations. First, the limited amount of SOE secured memory precludes any technique based on materialization (e.g., building a DOM [118] representation of the document). Second, limited CPU power and limited communication bandwidth lead to minimize the amount of data to be processed and decrypted in the SOE. Finally, we provide strong guarantees to preserve the confidentiality of data to combat data tampering and replay attacks conducted by a hacker to mislead the evaluator.

1.2 Outline

This chapter is organized as follows. In Section 2, we propose an accurate streaming evaluation of the access control based on automata to accommodate the constraints of the SOE. More precisely, we provide an evaluator capable of managing propagation and conflicts of access rules in a streaming fashion, and to detect situations where access rules become irrelevant. In Section 3, we exploit these situations by providing a compact *Skip Index* embedded in the XML data that enables to skip irrelevant or forbidden data in order to reduce the amount of data to be processed by the SOE.

The evaluation in a streaming fashion brings a new difficulty. Indeed, the delivery of elements may be conditioned by predicates which may be resolved later in the data stream, thus generating pending situations. In our context, the pending parts cannot obviously be buffered within the SOE. Section 4 provides two different ways to tackle this problem, yet without disclosing any unauthorized data, and an elegant way to cope with the management of multiple pending predicates. In Section 5, we propose techniques to guarantee the confidentiality of data while minimizing the cost ascribed to it. To this end, we provide first an encryption scheme and integrity checking methods supporting random accesses generated by the *Skip Index* and second, a technique to secure the refreshment of access rights. Finally, Section 6 concludes.

2 Streaming the access control

While several access control models for XML have been proposed recently, few papers address the enforcement of these models and, to the best of our knowledge, no one considers access control in a streaming fashion. Streaming is a prerequisite in our context considering the limited SOE secure storage capacity. At first glance, streaming access control resembles the well-known problem of XPath processing on streaming documents. There is a large body of work on this latter problem in the context of XML filtering [30][43][61]. These studies consider a very large number of XPath expressions (typically tens of thousands). The primary goal here is to select the subset of queries matching a given document (the query result is not a concern) and the focus is on indexing and/or combining a large amount of queries. One of the first works addressing the precise evaluation of complex XPath expressions over streaming documents is due to Peng and Chawathe [95] who proposed a solution to deliver parts of a document matching a single XPath. While access control rules are expressed in XPath, the nature of our problem differs significantly from the preceding ones. Indeed, the rule propagation principle along with its associated conflict resolution policies (see section 2.1 of Chapter 3) makes access control rules not independent. The interference between rules introduces two new important issues:

- *Access control rules evaluation:* for each node of the input document, the evaluator must be capable of determining the set of rules that applies to it and for each rule determining if it applies directly or is inherited. The nesting of the access control rules scopes determines the authorization outcome for that node. This problem is made more complex by the fact that some rules are evaluated lazily due to pending predicates.
- *Access control optimization:* the nesting of rule scopes associated with the conflict resolution policies inhibits the effect of some rules. The rule evaluator must take advantage of this inhibition to suspend the evaluation of these rules and even to suspend the evaluation of all rules if a global decision can be reached for a given subtree.

2.1 Access control rules evaluation

As streaming documents are considered, we make the assumption that the evaluator is fed by an event-based parser (e.g., SAX [102]) raising *open*, *value* and *close* events respectively for each opening, text and closing tag in the input document.

We represent each access control rule (i.e., XPath expression) by a non-deterministic finite automaton (NFA). Figure 28.b pictures the *Access control rules Automata* (*ARA*) corresponding to two rather simple access control rules expressed on an abstract XML document. This abstract example gives us the opportunity to study several situations (including the trickiest ones) on a simple document. In our *ARA* representation, a circle denotes a state and a double circle a final state, both identified by a unique *StateId*. Directed edges represent transitions, triggered by *open* events matching the edge label (either an

element name or *). Thus, directed edges represent the child (/) XPath axis or a wildcard depending on the label. To model the descendant axis (//), we add a self-transition with a label * matched by any *open* event. An *ARA* includes one *navigational path* and optionally one or several *predicate paths* (in grey in the figure). To manage the set of *ARA* representing a given access control policy, we introduce the following data structures:

- *Tokens and Token Stack:* we distinguish between *navigational tokens* (*NT*) and *predicate tokens* (*PT*) depending on the *ARA* path they are involved in. To model the traversal of an *ARA* by a given token, we actually create a token proxy each time a transition is triggered and we label it with the destination *StateId*. The terms token and token proxy are used interchangeably in the rest of this chapter. The navigation progress in all *ARA* is memorized thanks to a unique stack-based data structure called *Token Stack*. The top of the stack contains all active *NT* and *PT* tokens, i.e. tokens that can trigger a new transition at the next incoming event. Tokens reaching their final states are then not considered in the stack since they cannot generate new tokens. Tokens created by a triggered transition are pushed in the stack. The stack is popped at each *close* event. The goal of *Token Stack* is twofold: allowing a straightforward backtracking in all *ARA* and reducing the number of tokens to be checked at each event (only the active ones, at the top of the stack, are considered).
- *Rule status and Authorization Stack:* Let assume for the moment that access control rule expressions do not exploit the descendant axis (no //). In this case, a rule is said to be *active*, – meaning that its scope covers the current node and its subtree – if all final states of its *ARA* contain a token. A rule is said to be *pending* if the final state of its navigational path contains a token while the final state of some predicate path has not yet been reached. The *Authorization Stack* registers the *NT* tokens having reached the final state of a navigational path, at a given depth in the document. The scope of the corresponding rule is bounded by the time the *NT* token remains in the stack. This stack is used to solve conflicts between rules. The status of a rule present in the stack can be fourfold: *positive-active* (denoted by $\oplus$), *positive-pending* (denoted by $\oplus^?$), *negative-active* (denoted by $\ominus$), *negative-pending* (denoted by $\ominus^?$). By convention, the bottom of the stack contains an implicit *negative-active* rule materializing a closed access control policy (i.e., by default, the set of objects the user is granted access to is empty).
- *Rule instances materialization:* Taking into account the descendant axis (//) in the access control rules expressions makes things more complex to manage. Indeed, the same element names can be encountered at different depths in the same document, leading several tokens to reach the final state of a navigational path and predicate paths in the same *ARA*, without being related together[1]. To tackle this situation, we label navigational and predicate token proxies with the *depth* at which the original predicate token has been

[1] The complexity of this problem has been highlighted by Peng and Chawathe [95].

created, materializing their participation in the same *rule instance*[2]. Consequently, a token (proxy) must hold the following information: *RuleId* (denoted by *R*, *S*, …), Navigational/Predicate status (denoted by *n* or *p*), *StateId* and *Depth*[3]. For example, $Rn2_2$ and $Rp4_2$ (also noted 2_2, 4_2 to simplify the figures) denotes the navigational and predicate tokens created in Rule *R*'s *ARA* at the time element *b* is encountered at depth 2 in the document. If the transition between states 4 and 5 of this *ARA* is triggered, a token proxy $Rp5_2$ will be created and will represent the progress of the original token $Rp4_2$ in the *ARA*. All these tokens refer to the same rule instance since they are labeled by the same depth. A rule instance is said to be *active* or *pending* under the same condition as before, taking into account only the tokens related to this instance.

- *Predicate Set:* this set registers the *PT* tokens having reached the final state of a predicate path. A *PT* token, representing a predicate instance, is discarded from this set at the time the current depth in the document becomes less than its own depth.

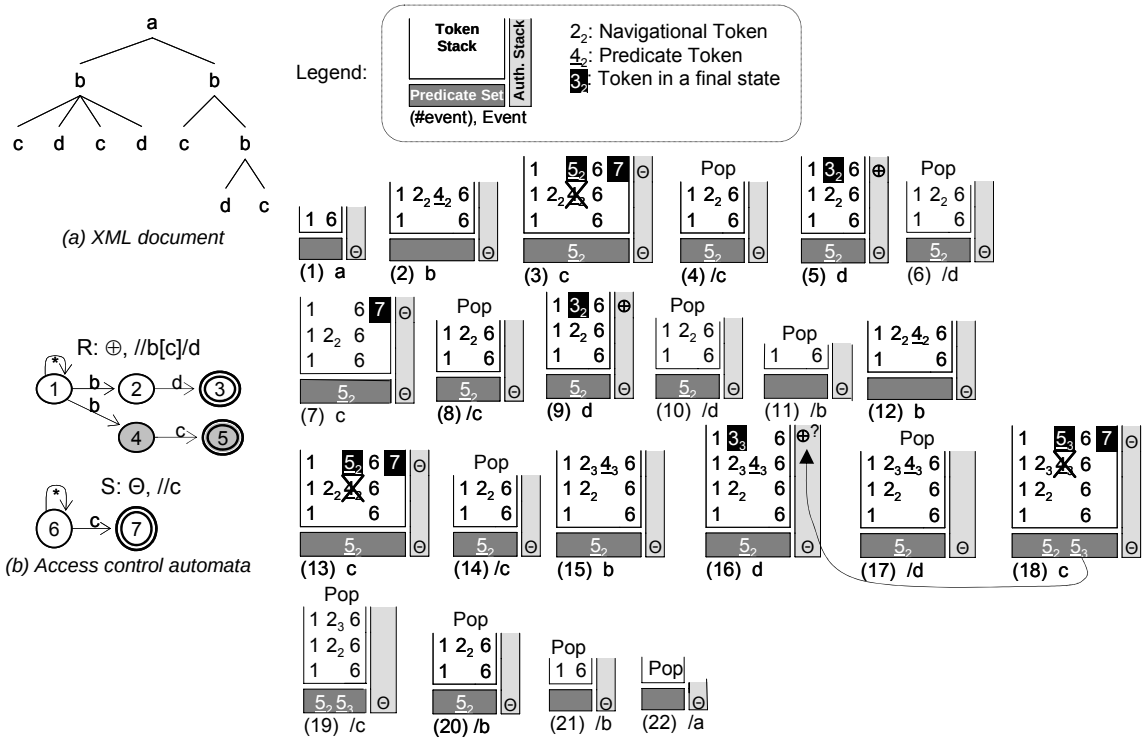


Figure 28. Execution snapshot

Stack-based data structures are well adapted to the traversal of a hierarchical document.

[2] To illustrate this, let us consider the rule *R* and the right subtree of the document presented in Figure 28. The predicate path final state 5 (expressing *//b[c]*) can be reached from two different instances of *b*, respectively located at depth 2 and 3 in the document, while the navigational path final state 3 (expressing *//b/d*) can be reached only from *b* located at depth 3. Thus, a single rule instance is valid here, materialized by navigational and predicate tokens proxies labeled with the same depth 3.

[3] If a same ARA contains different predicate paths starting at different levels of the navigational path, a *NT* token will have in addition to register all *PT* tokens related to it.

However, we need a direct access to any stack level to update pending information and to allow some optimizations detailed below. Figure 28.c represents an execution snapshot based on these data structures. For ease of understanding, we represent in the figure the tokens which reach their final states (in a black box) in the *Token Stack* even though they should not appear since they cannot generate new tokens. These tokens are thus not being considered in the algorithms. This snapshot being almost self-explanatory, we detail only a small subset of steps.

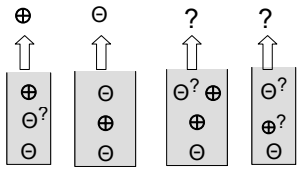
- *Step 2*: the *open* event *b* generates two tokens $Rn2_2$ and $Rp4_2$, participating in the same rule instance.
- *Step 3*: the *ARA* of the negative rule S reaches its final state and an active instance of S is pushed in the *Authorization Stack*. The current authorization remains negative. Token $Rp5_2$ enters the *Predicate Set*. The corresponding predicate will be considered true until level 2 of the *Token Stack* is popped (i.e., until event */b* is produced at step 9). Thus, there is no need to continue to evaluate this predicate in this subtree and token $Rp4_2$ can be discarded from the *Token Stack*.
- *Step 5*: An active instance of the positive rule *R* is pushed in the *Authorization Stack*. The current authorization becomes positive, allowing the delivery of element *d*.
- *Step 16*: A new instance of *R* is pushed in the *Authorization Stack*, represented by token $Rn3_3$. This instance is pending since the token $Rp5_2$ pushed in the *Predicate Set* at step 12 (event *c*) does not participate in the same rule instance.
- *Step 18*: Token $Rp5_3$ enters the *Predicate Set*, changing the status of the associated rule instance to *positive-active*. The management of pending predicates and their effect on the delivery process is more deeply studied in section 4.

2.2 Conflict Resolution

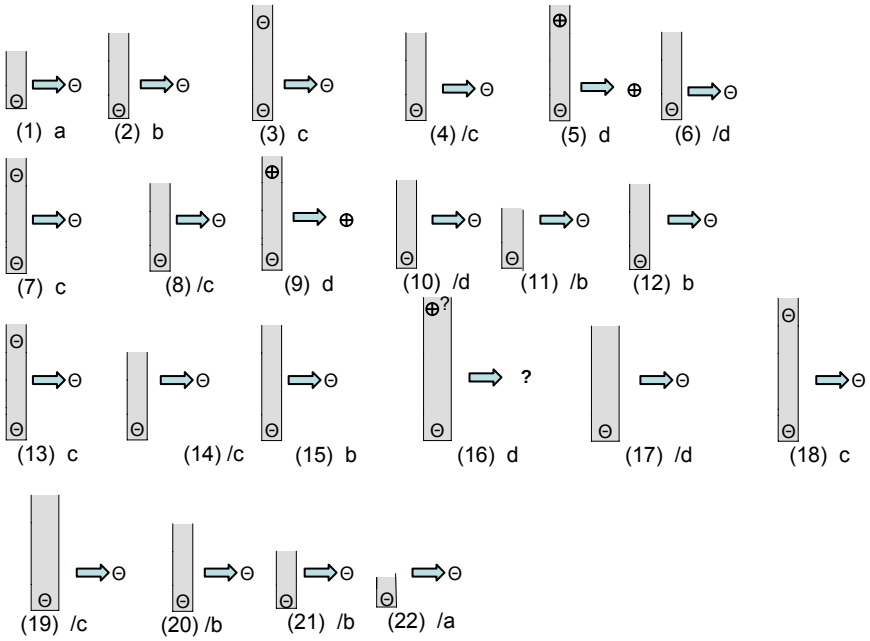
From the information kept in the *Authorization Stack*, the outcome of the current document node can be easily determined. The conflict resolution algorithm presented in Figure 29 integrates the closed access control policy (line 1), the *Denial-Takes-Precedence* (line 2) and *Most-Specific-Object-Takes-Precedence* (lines 5 and 7) policies to reach a decision. In the algorithm, *AS* denotes the *Authorization Stack* and *AS[i].RuleStatus* denotes the set of status of all rules registered at level *i* in this stack. In the first call of this recursive algorithm, *depth* corresponds to the top of *AS*. Recursion captures the fact that a decision may be reached even if the rules at the top of the stack are pending, depending on the rule status found in the lower stack levels. Note, however, that the decision can remain pending if a pending rule at the top of the stack conflicts with other rules. In that case, the current node has to be buffered, waiting for a delivery condition. This issue is tackled in section 4. The rest of the algorithm is self-explanatory and examples of conflict resolutions are given in the figure. The *DecideNode* algorithm presented below considers only the access control rules. Things

are slightly more complex if queries are considered too. Queries are expressed in XPath and are translated in a non-deterministic finite automaton in a way similar to access control rules. However, a query cannot be regarded as an access control rule at conflict resolution time. The delivery condition for the current node of a document becomes twofold: (1) the delivery decision must be true and (2) the query must be interested in this node. The first condition is the outcome of the *DecideNode* algorithm. The second condition is matched if the query is *active*, that is if all final states of the query *ARA* contain a token, meaning that the current node is part of the query scope. We represent in Figure 29 an execution snapshot of the *DecideNode* algorithm based on the execution depicted in Figure 28.

DecideNode(depth) → Decision ∈ {⊕, ⊖,?}
1: If depth = 0 then return '⊖'
2: elseif '⊖'∈ AS[depth].RuleStatus then return '⊖'
3: elseif '⊕' ∈ AS[depth].RuleStatus and
4: '⊖$^{?}$' ∉ AS[depth].RuleStatus then return '⊕'
5: elseif DecideNode(depth -1) = '⊖' and
6: ∀t∈{'⊕$^{?}$','⊕'} t∉ AS[depth].RuleStatus then return '⊖'
7: elseif DecideNode(depth -1) = '⊕' and
8: '⊖$^{?}$' ∉ AS[depth] RuleStatus then return '⊕'
9: else return '?'



Examples of conflict resolution



Execution snapshots of the DecideNode algorithm

Figure 29. Conflict resolution algorithm

2.3 Optimization issues

The first optimization that can be devised is doing a static analysis of the system of rules composing an access control policy. Query containment property can be exploited to decrease the complexity of this system of rules. Let us denote by ⊆ the containment relation

between rules *R, S …T*. If $S \subseteq R \wedge (R.Sign=S.Sign)$, the elimination of S could be envisioned. However, this elimination is precluded if, for example, $\exists T / T \subseteq R \wedge (T.Sign \neq R.Sign) \wedge (S \subseteq T)$. Thus, rules cannot be pairwise examined and the problem turns to check whether some partial order among rules can be defined wrt. the containment relation, e.g., $\{T_i, \ldots T_k\} \subset \{S_i, \ldots S_k\} \subseteq \{R_i, \ldots R_k\} \wedge \forall i, (R_i.Sign=S_i.Sign \wedge S_i.Sign \neq T_i.Sign) \Rightarrow \{S_i, \ldots S_k\}$ can be eliminated. Note that this strong elimination condition is sufficient but not necessary. For instance, let *R* and *S* be two positive rules respectively expressed by */a* and */a/b[P1]* and *T* be a negative rule expressed by */a/b[P2]/c*. *S* can still be eliminated while $T \not\subset S$, because the containment holds for each subtree where the two rules are active together. The problem is particularly complex considering that the query containment problem itself has been shown co-NP complete for the class of XPath expressions of interest, that is $XP^{\{[],//,*\}}$ [83]. This issue could be further investigated since more favorable results have been found for subclasses of $XP^{\{[],//,*\}}$ [83], but this work is outside the scope of this thesis. Some other works such as developed by Mohan et al. [83] (as described in Section 3.2 of Chapter 3) can be used to compile all the access control rules and the user query in an XQuery statement. However, embedding an XQuery evaluator is at the present impractical considering the drastic constraints in terms of memory of the SOE.

A second form of optimization is to suspend dynamically the evaluation of *ARA* that become irrelevant or useless inside a subtree. The knowledge gathered in the *Token Stack*, *Authorization Stack* and *Predicate Set* can be exploited to this end. The first optimization is to suspend the evaluation of a predicate in a subtree as soon as an instance of this predicate has been evaluated to true in this subtree. This optimization has been illustrated by Step 3 of Figure 28.c. The second optimization is to evaluate dynamically the containment relation between active and pending rules and take benefit of the elimination condition mentioned above. From the *Authorization Stack*, we can detect situations where the following local condition holds: $(T \subset S \subseteq R) \wedge (R.Sign=S.Sign \wedge S.Sign \neq T.Sign)$, the stack levels reflecting the containment relation inside the current subtree. *S* can be inhibited in this subtree. If stopping the evaluation of some *ARA* is beneficial, one must keep in mind that the two limiting factors of our architecture are the decryption cost and the communication cost. Therefore, the real challenge is being able to take a common decision for complete subtrees, a necessary condition to detect and skip prohibited subtrees, thereby saving both decryption and communication costs.

Without any additional information on the input document, a common decision can be taken for a complete subtree rooted at node *n* iff: (1) the *DecideNode* algorithm can deliver a decision *D* (either $\oplus$ or $\ominus$) for *n* itself and (2) a rule *R* whose sign contradicts *D* cannot become active inside this subtree (meaning that all its final states, of navigational path and potential predicate paths, cannot be reached altogether). These two conditions are compiled in the algorithm presented in Figure 30. In this algorithm, *AS* denotes the *Authorization Stack*, *TS* the *Token Stack*, *TS[i].NT (resp. TS[i].PT)* the set of *NT* (resp. *PT*) tokens registered at level *i* in this stack and top is the level of the top of a stack. In addition,

t.RuleInst denotes the rule instance associated with a given token, *Rule.Sign* the sign of this rule and *Rule.Pred* a boolean indicating if this rule includes predicates in its definition.

The immediate benefit of this algorithm is to stop the evaluation for any active *NT* tokens and the main expected benefit is to skip the complete subtree if this decision is ⊖. Note however that only *NT* tokens are removed from the stack at line 4. The reason for this is that active *PT* tokens must still be considered, otherwise pending predicates could remain pending forever. As a conclusion, a subtree rooted at *n* can be actually skipped iff: (1) the decision for *n* is ⊖, (2) the *DecideSubtree* algorithm decides ⊖ and (3) there are no *PT* token at the top of the *Token Stack* (which turns to be empty). Unfortunately, these conditions are rarely met together, especially when the descendant axis appears in the expression of rules and predicates (e.g., rules such as defined in Figure 28). The next section introduces a *Skip Index* structure that gives useful information about the forthcoming content of the input document. The goal of this index is to detect a priori rules and predicates that will become irrelevant, thereby increasing the probability to meet the aforementioned conditions.

```
DecideSubtree() → Decision ∈ {⊕, ⊖,?}
1: D = DecideNode(AS.top)
2: if D = '?' then return '?'
3: if not (∃ nt ∈TS[top].NT / nt.Rule.Sign ≠ D and (not nt.Rule.Pred
4:       or (∃ pt ∈TS[top].PT / pt.RuleInst = nt.RuleInst)) then TS[top].NT = ∅; return (D)
5: else return '?'
```

Figure 30. Decision on a complete subtree

When queries are considered, any subtree not contained in the query scope is candidate to a skip. This situation holds as soon as the *NT* token of the query (or *NT* tokens when several instances of the same query can co-exist) becomes inactive (i.e., is no longer element of *TS[top].NT*). This token can be removed from the *Token Stack* but potential *PT* tokens related to the query must still be considered, again to prevent pending predicate to remain pending forever. As before, the subtree will be actually skipped if the *Token Stack* becomes empty.

3 Skip index

This section introduces a new form of indexation structure, called *Skip Index*, designed to detect and skip the unauthorized fragments (wrt. an access control policy) and the irrelevant fragments (wrt. a potential query) of an XML document, while satisfying the constraints introduced by the target architecture (streaming encrypted document, scarce SOE storage capacity).

In the context of XML filtering [13][33] and XML routing [30][43][61], the authors devised a streaming index which consists of appending to each subtree its size, giving the possibility to skip it when no more query can apply to this subtree. However, since no extra information on the content of the subtree is provided, the use of such index is rather limited (e.g., a query

of the form *//a* precludes any skip).

The first distinguishing feature of the required index is the necessity to keep it encrypted outside of the SOE to guarantee the absence of information disclosure. The second distinguishing feature (related to the first one and to the SOE storage capacity) is that the SOE must manage the index in a streaming fashion, similarly to the document itself. These two features lead to design a very compact index (its decryption and transmission overhead must not exceed its own benefit), embedded in the document in a way compatible with streaming. For these reasons, we concentrate on indexing the structure of the document, pushing aside the indexation of its content. Structural summaries [8] or XML skeleton [28] could be considered as candidate for this index. Beside the fact that they may conflict with the size and streaming requirements, these approaches do not capture the irregularity of XML documents (e.g., medical folders are likely to differ from one instance to another while sharing the same general structure).

In the following, we propose a highly compact structural index, encoded recursively into the XML document to allow streaming. An interesting side effect of the proposed indexation scheme is to provide new means to further compress the structural part of the document.

3.1 Skip Index encoding scheme

The primary objective of the index is to detect rules and queries that cannot apply inside a given subtree, with the expected benefit to skip this subtree if the conditions stated in Section 2.3 are met. Keeping the compactness requirement in mind, the minimal structural information required to achieve this goal is the set of element tags, or tags for short, that appear in each subtree. While this metadata does not capture the tags nesting, it reveals oneself as a very effective way to filter out irrelevant XPath expressions. We propose below data structures encoding this metadata in a highly compact way. These data structures are illustrated in Figure 31.a on an abstract XML document.

- *Encoding the set of descendant tags*: The size of the input document being a concern, we make the rather classic assumption that the document structure is compressed thanks to a dictionary of tags [8][109][4]. The set of tags that appear in the subtree rooted by an element *e*, named $DescTag_e$, can be encoded by a bit array, named $TagArray_e$, of length N_t, where N_t is the number of entries of the tag dictionary. A recursive encoding can further reduce the size of this metadata. Let us call *DescTag(e)* the bijective function that maps $TagArray_e$ into the tag dictionary to compute $DescTag_e$. We can trade storage overhead for computation complexity by reducing the image of *DescTag(e)* to $DescTag_{parent(e)}$ in place of the tag dictionary. The length of the *TagArray* structure decreases while descending into the document hierarchy at the price of making the

[4] Considering the compression of the document content itself is out of the scope of our study. Anyway, value compression does not interfere with our proposal as far as the compression scheme remains compatible with the SOE resources.

DescTag() function recursive. Since the number of element generally increases with the depth of the document, the gain is substantial. To distinguish between intermediate nodes and leaves (that do not need the *TagArray* metadata), an additional bit is added to each node.

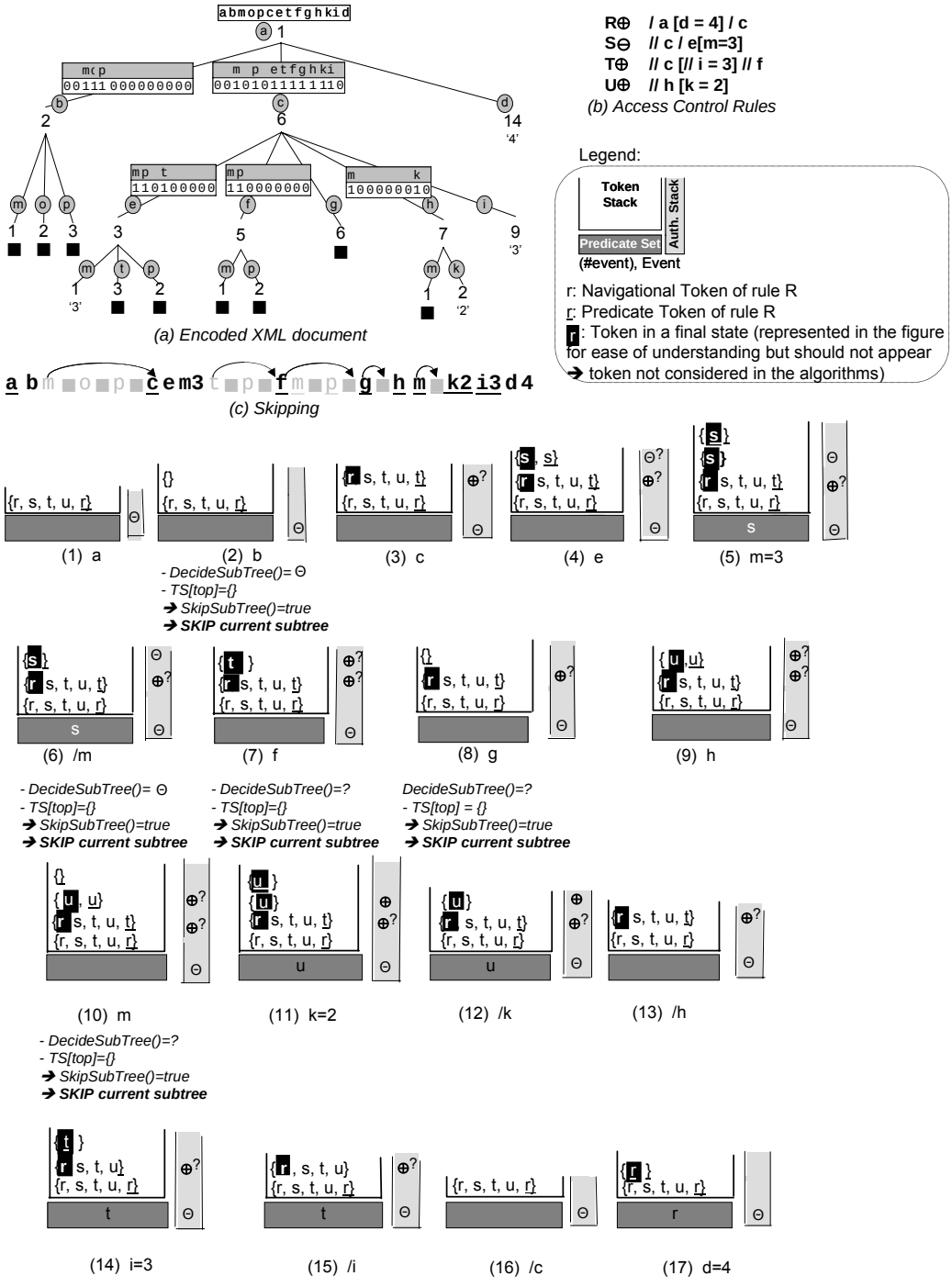


Figure 31. Skip Index example

- *Encoding the element tags*: In a dictionary-based compression, the tag of each element *e* in the document is replaced by a reference to the corresponding entry in the dictionary. $Log_2(N_t)$ bits are necessary to encode this reference. The recursive encoding of the set of descendant tags can be exploited as well to compress further the encoding of tags themselves. Using this scheme, $Log_2(DescTag_{parent(e)})$ bits suffice to encode the tag of an element *e*.
- *Encoding the size of a subtree:* Encoding the size of each subtree is mandatory to implement the skip operation. At first glance, $Log_2(size(document))$ bits are necessary to encode $SubtreeSize_e$, the size of the subtree rooted by an element *e*. Again, a recursive scheme allows to reduce the encoding of this size to $Log_2(SubtreeSize_{parent(e)})$ bits. Storing the *SubtreeSize* for each element makes closing tags unnecessary.
- *Decoding the document structure*: The decoding of the document structure must be done by the SOE, efficiently, in a streaming fashion and without consuming much memory. To this end, the SOE stores the tag dictionary and uses an internal *SkipStack* to record the *DescTag* and *SubtreeSize* of the current element. When decoding an element *e*, $DescTag_{parent(e)}$ and $SubtreeSize_{parent(e)}$ are retrieved from this stack and used to decode in turn $TagArray_e$, $SubtreeSize_e$ and the encoded tag of *e*.
- *Updating the document:* In the worst case, updating an element *e* induces an update of the *SubtreeSize*, the *TagArray* and the encoded tag of each *e* ancestors and of their direct children. In the best case, only the *SubtreeSize* of *e* ancestors need be updated. The worst case occurs in two rather infrequent situations. The *SubtreeSize* of *e* ancestor's children have to be updated if the size of *e* father grows (resp. shrinks) and jumps a power of 2. The *TagArray* and the encoded tag of *e* ancestor's children have to be updated if the update of *e* generates an insertion or deletion in the tag dictionary.

3.2 Skip Index usage

As said before, the primary objective of the *Skip Index* is to detect rules and queries that cannot apply inside a given subtree. This means that any active token that cannot reach a final state in its *ARA* can be removed from the top of the *Token Stack*. Let us call *RemainingLabels(t)* the function that determines the set of transition labels encountered in the path separating the current state of a token *t* from the final state of its *ARA*, and let us call *e* the current element in the document. A token *t*, either navigational or predicate, will be unable to reach a final state in its *ARA* if $RemainingLabels(t) \not\subset DescTag_e$. Note that this condition is sufficient but not necessary since the *Skip Index* does not capture the element tags nesting. Once this token filtering has been done, the probability for the *DecideSubtree* algorithm to reach a global decision about the subtree rooted by the current element *e* is greatly increased since many irrelevant rules have been filtered. If this decision is negative (⊖) or pending (?), a skip of the subtree can be envisioned.

SkipSubtree () → Decision ∈ {true,false}

1:For each token t ∈TS[top].NT ∪ TS[top].PT
2: if RemainingLabels(t) ⊄ $DescTag_e$ then remove t from TS[top]
3: if DecideSubTree() ∈ {'⊖', '?'} and (TS[top].NT = ∅) and(TS[top].PT = ∅) then return true
4: else return false

Figure 32. Skipping decision

This skip is actually possible if there are no more active tokens, either navigational or predicate, at the top of the *Token Stack*. The algorithm *SkipSubtree* given in Figure 32 decides whether the skip is possible or not. Let us remark that this algorithm should be triggered both on *open* and *close* events. Indeed, each element may change the decision delivered by the algorithm *DecideNode*, then *DecideSubtree* and finally *SkipSubtree* with the benefit of being able to skip a bigger subtree at the next step.

Figure 31 shows an illustrative XML document and its encoding, a set of access control rules and the skips done while analyzing the document. The information in grey is presented to ease the understanding of the indexing scheme but is not stored in the document. Let us consider the document analysis (for clarity, we use below the real element tags instead of their encoding). Execution snapshots are provided in Figure 31.d. In this representation, tokens in their final states are represented (in black boxes) in the *Token Stack* for ease of understanding even though they should not appear since they cannot generate new tokens. These tokens are thus not considered in the algorithms. At the time element *b* (leftmost subtree) is reached (Step 2), all the active rules are stopped thanks to $TagArray_b$ and the complete subtree can be skipped (the decision is ⊖ due to the closed access control policy). When element *c* is reached (Step 3), Rule *R* becomes pending. However, the analysis of the subtree continues since $TagArray_c$ does not allow more filtering. When element *e* is reached (Step 4), $TagArray_e$ filters out rules *R*, *T* and *U*. Rule *S* becomes negative-active when the value '3' is encountered below element *m*. On the closing event, *SkipSubtree* decides to skip the *e* subtree. This situation illustrates the benefit to trigger the *SkipSubtree* at each opening and closing events. The analysis continues following the same principle and leads to deliver the elements underlined in Figure 31.c.

4 Management of pending predicates

An element in the input document is said to be pending if its delivery depends on a pending rule, that is a rule for which the navigational path final state has been reached but at least one predicate path final state remains to be reached. This unfavorable case is unfortunately frequent. Indeed, any rule of the form */…/e[P]* lead invariably to a pending situation. Any rule of the form */../e[P]/../* generates also a pending situation until *P* has been evaluated to true. Indeed, a false evaluation of *P* does not stop the pending situation because another instance of *P* may be true elsewhere in the document.

This situation is made even more difficult since pending parts cannot be buffered inside the

SOE considering the assumption made on its storage capacity. Moreover, when multiple pending predicates are considered, their management can become particularly complex. In the following, we present two different ways to tackle this problem, each adapted to a specific context. Then, we provide a solution to cope with multiple pending predicates.

4.1 Pending delivery

By nature, pending predicates are incompatible with applications consuming documents in a strict streaming fashion (note that a predicate may remain pending until the document end). When considering pending predicates, we make the assumption that the terminal has enough memory to buffer the pending parts of the document. We first propose a simple solution adapted to documents delivered in a strict streaming fashion (e.g., push-based access, broadcast). Then, we describe a more accurate solution adapted to contexts where backward and forward accesses are allowed in the document (e.g., pull-based access, VCR).

Strict streaming: When receiving the document in a strict streaming fashion, the outcome of pending predicates cannot be known beforehand. Pending parts cannot be buffered inside the SOE considering the assumption made on its storage capacity. To tackle this problem pending parts are externalized to the terminal in an encrypted form using a temporary encryption key. If later in the parsing, the pending parts are found to be authorized, the temporary key is delivered and discarded otherwise. A different temporary key is generated for every pending part which depends on different predicates. We refer in the following to *output block* as a contiguous output encrypted with the same key or a contiguous clear-text output.

Each issued output block B_i (encrypted or not) may include additional information to integrate it consistently into the result document when some ancestors or left sibling elements are in a pending situation. This information is called the *pathlist* of B_i and contains the list of tags in the path between the last authorized issued element and B_i's root. Indeed, if B_i's ancestors are finally found to be prohibited, this list is necessary to enforce the structural rule stating that the result document must keep the same structure as the input one (cf. Section 2.1 of Chapter 3)[5]. In order to avoid confusion between elements sharing the same tag during B_i's integration, every delivered pending element is marked with an identifier (e.g., a random value). These values are kept in the SOE and are associated to the tag of B_i's pathlist[6].

Backward/Forward access: In that case, we make the assumption that the pending parts can be read back (e.g., from the server) when pending predicates are solved. The objective pursued is therefore to detect pending subtrees and to leave them aside thanks to the *Skip Index* until the pending situation is solved. The goal is to never read and analyze the same

[5] An alternative is to tag forbidden ancestors with a dummy value to comply with [53][59].

[6] Note that the marking overhead can be restricted to ancestors of pending subtrees thanks to the *Skip Index* (marking is deactivated as soon as no pending rule may apply in the subtree).

data twice. The skipping strategy and the associated reassembling strategy proposed below meet this goal.

Pending subtrees are externalized at the time the logical expression conditioning their delivery is evaluated to true (e.g., *//a[d=6]/b[c=5]* requires that an element *d=6* and an element *c=5* are found to be true). Therefore, pending subtrees can be delivered in an order different from their initial order in the input document. The benefit of this asynchrony is to reduce the latency of the access control management and to free the SOE internal memory, at the cost of a more complex reassembling of the final result. Indeed, the initial parent, descendant and sibling relationships have to be preserved at reassembling time. This forces to register, at parsing time, the information <*offset*, *level*, *anchor*, *condition*> for each pending subtree (Figure 33 illustrates a pending situation involving four subtrees S_0, S_1, S_2, S_3 respectively associated with pending predicates P_0, P_1, P_2, P_3). *Offset* and *level* are respectively the subtree offset and subtree depth in the initial document (we use the term level to avoid any confusion with the depth attached to tokens); *anchor* references the target position of the subtree in the result (see below); *condition* is the logical expression conditioning the subtree delivery. This information is kept for each pending subtree in a list named *Pending List* and denoted by *PL*. The reassembling process is as follows:

- *Anchor assignation:* Let assume that each element *e* in the result document is labeled by a unique number *Ne* (representing for example the ordering in which elements are delivered). The future position of a pending subtree *e'* in the result can be uniquely identified by a single number using the following convention: *Ne* if *e'* is a potential right sibling of *e* or *-Ne* if *e'* is the potential leftmost child of *e*. No anchor needs to be memorized for pending right siblings and embedded subtrees of a pending subtree *e'* (in Figure 33: N_{S0} = -3 while S_1, S_2, S_3 have no anchor). The reason for this is twofold: (1) these subtrees share *e'* anchor until one of their left sibling (see *subtree delivery*) or ancestor (see *embedded pending subtrees*) is delivered and (2) parent and sibling relationships among pending elements can be recovered from the Pending List as follows:

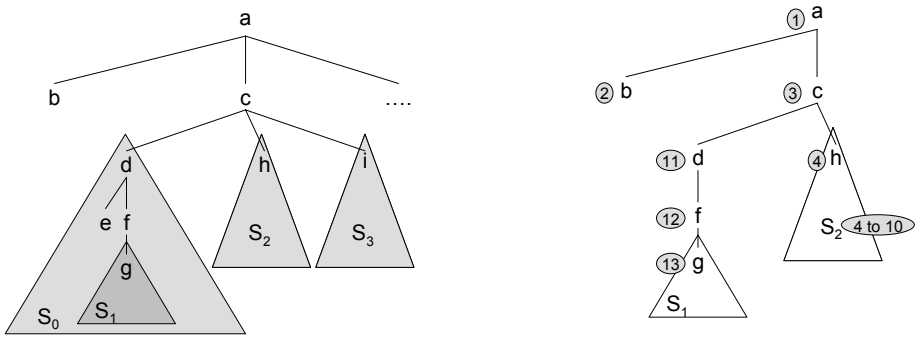
  π denotes the precedence relation in the Pending List

  $A \text{ child\_of } B \Leftrightarrow B\pi A \wedge Level_A = Level_B + 1 \wedge \neg (\exists C / B\pi C\pi A \wedge Level_C = Level_B)$

  $A \text{ right\_sibling\_of } B \Leftrightarrow B\pi A \wedge Level_A = Level_B \wedge \neg (\exists C / B\pi C\pi A \wedge Level_C \leq Level_A)$

- *Embedded pending subtrees*: a pending subtree may in turn embed other pending subtrees leading to tricky situations depending on the pending predicate issues (i.e., affecting the delivery order). Let assume the innermost subtree S_{inner} (e.g., S_1) is found authorized while the outermost subtree S_{outer} (e.g., S_0) is still pending. All the tags in the path from S_{inner} to its last authorized ancestor *Anc* (e.g., *d* and *f* connect S_1 to element *c*) must be delivered in their hierarchical order, along with S_{inner} to enforce the *Structural rule*. Let assume that S_{outer} is delivered afterward. S_{inner} and the path connecting it to *Anc* must not be delivered twice. To cope with this situation and produce a consistent result document, the parts of S_{outer} previously delivered must be recorded. This information is stored in a *skip list* integrated in the *PL* structure (see S_0 skip list in Figure 33).

- *Subtree delivery*: At the time a pending subtree *e'* is delivered, its place in the result document is determined by its anchor. In turn, *Ne'* (resp. *-Ne'*) becomes the anchor of the pending right sibling (resp. pending leftmost child) of *e'*, if any (e.g., S_3 anchor is updated when S_2 is delivered). To deliver the subtree *e'*, the whole subtree is read back from the input document, decrypted and delivered taking care to skip elements that may have been already delivered (i.e., authorized embedded subtrees, tags delivered to enforce the structural rule) and those that may not be delivered (still embedded pending subtrees and denied embedded subtrees).



(a) Initial document with pending subtrees

(b) Delivered document (with anchors)

PL structure: *for each subtree: (offset, level, anchor, condition, skiplist)*

Initial State: PL = ((@$_{S0}$, 3, -3, P_0, (@$_{S1}$)), d, f, (@$_{S1}$, 5, Null, P_1, Null), (@$_{S2}$, 3, Null, P_2, Null), (@$_{S3}$, 3, Null, P_3, Null))
P_2 = true: PL = ((@$_{S0}$, 3, -3, P_0, (@$_{S1}$)), d, f, (@$_{S1}$, 5, Null, P_1, Null), (@$_{S3}$, 3, 4, P_3, Null))
P_1 = true: PL = ((@$_{S0}$, 3, -3, P_0, ((@$_d$,11), (@$_f$, 12), (@$_{S1}$, 13)), (@$_{S3}$, 3, 4, P_3, Null))
P_0 = false: PL = ((@$_{S3}$, 3, 4, P_3, Null))
P_3 = false: PL = ()

(c) Snapshots of the pending list

Figure 33. Pending predicate management

Figure 33 illustrates the problem of embedded pending subtrees, showing the state of the *Pending List* at different steps during the execution. The figure is self-explanatory.

4.2 Coping with multiple pending predicates

The conditions recorded in the Pending List may be complex logical expressions. Each time a pending predicate is resolved, these conditions must be evaluated to determine whether some pending subtrees can be delivered. When coping with multiple pending predicates, there is a clear need to organize them in a way that reduces the storage overhead of Pending List conditions and the time required by their evaluation.

Let us call a *Pending Class* the set of pending subtrees whose delivery depends on the same logical expression. As the number of pending predicates is likely to be small, the logical expressions can be modeled using a bit vector representing the truth table. To minimize the memory consumption, two vectors *V* and *B* are attached to each *Pending Class*. *V={(p, d)}* is a list of predicates identified by a predicate id *p* and the depth *d* at which it occurs; *B* is the truth table results representing the logical expression (in Figure 34, only the gray parts are stored in memory, the rest is implicit). Using binary operations on bit vectors (e.g., bitwise

AND, bit shift), *V* and *B* may be expanded incrementally as new pending predicates are considered or, shrunk when pending predicates are resolved. This truth table evolves until it becomes either a true function (the associated data are delivered) or a false function (the associated data are prohibited).

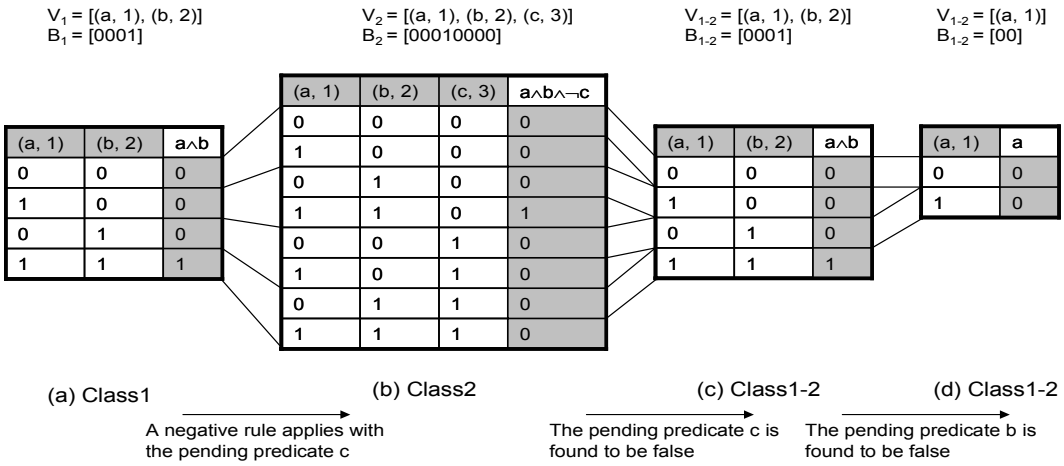


Figure 34. Multiple pending predicate management

Figure 34 illustrates how the logical expressions are incrementally built and evaluated. Let assume an initial situation with a unique *Pending Class Class1* containing the subtree S_1 rooted at *n1* with a delivery condition $a \wedge b$, where predicates *a* and *b* occur respectively at depth 1 and 2. Let us now consider an element *f*, descendant of n_1 on which a new pending rule *R* applies with a predicate *c*. If R is a negative (resp. positive) rule the logical expression conditioning the delivery of element *f* is $a \wedge b \wedge \neg c$ (resp. $a \wedge b \wedge c$). To reflect this situation, a new class *Class2* is created with a vector V_2 derived from V_1 (predicate *c* is inserted in the list in the lexical order) and a bit vector B_2 built as follows (we assume that B_2 must represent the expression $a \wedge b \wedge \neg c$). A bit vector *B'* is created from B_1 by duplicating every segment of $2^{index(c)-1}$ bits, *index(c)* being the position of predicate *c* in V_2 (e.g. *index((c, 3)) = 3*). Predicate *c* is represented by a bit vector B_c made as an alternation of $2^{index(c)-1}$ bit segments of 0 and 1. Finally, B_2 is computed thanks to a bitwise AND NOT between *B'* and B_c (Figure 34.b). Let assume predicate *c* is found to be false the rows representing a true value for *c* in B_2 are removed and the rest is shifted backward (Figure 34.c), that is all the rows in the intervals $[k*2^{index(c)} + 2^{index(c)-1}, (k+1)*2^{index(c)}[$, *k* varying from *0* to $|V| - 1$ where $|V|$ is the cardinality of *V*. The resulting truth table of *Class2* being the same as the one of C*lass1*, the two classes are merged into the *Class1-2*. Let assume predicate *b* is evaluated to false, the table is shrunk as pictured in Figure 34.d. The resulting truth table contains only bit values equal to zero, hence the logical expression is evaluated to false and all the elements of this class are discarded.

This solution is both time and space efficient considering that bitwise logical operations are applied on bit arrays that are likely to fit in 32-bit numbers.

5 Guaranteeing the confidentiality of data

5.1 Random integrity checking

Encryption and hashing are required to guarantee respectively the confidentiality and the integrity of the input document. Unfortunately, standard integrity checking methods are badly adapted to our context for two important reasons. First, the memory limitation of the SOE imposes a streaming integrity checking. Second, the integrity checking must tackle the forward and backward random accesses to the document incurred by the *Skip Index* and by the reassembling of pending document fragments. In this section, we provide a solution to face potential attacks on an input document.

In a client-based context, the attacker is the user himself. For instance, a user being granted access to a medical folder *X* may try to extract unauthorized information from a medical folder *Y*. Let assume that the document is encrypted with a classic block cipher algorithm (e.g., DES or triple-DES) and that blocks are encrypted independently (e.g., following the ECB mode [77][103]), identical plaintext blocks will generate identical ciphered values. In that case, the attacker can conduct different attacks: substituting some blocks of folders *X* and *Y* to mislead the access control manager and decrypt part of *Y*; building a dictionary of known plaintext/ciphertext pairs from authorized information (e.g., folder *X*) and using it to derive unauthorized information from ciphertext (e.g., folder *Y*); making statistical inference on ciphertext. Additionally, if no integrity checking occurs, the attacker can randomly modify some blocks, inducing a malfunction of the rule processor (e.g., Bob is authorized to access folders of patients older than 80 and he randomly alters the ciphertext storing the age).

To face these attacks, we exploit two techniques. Regarding encryption, the objective is to generate different ciphertexts for different instances of a same value. This property could be obtained by using a Cipher Bloc Chaining (*CBC*) mode in place of *ECB*, meaning that the encryption of a block depends on the preceding block [77][103]. This however would introduce an important overhead at decryption time if random accesses are performed in the document. As an alternative, we merge the position of a value with the value itself at encryption time, i.e., we perform an XOR (denoted $\oplus$) between each 8 byte block and the position of this block in the document, before encrypting the result using a simple *ECB* mode. The encryption itself is performed with a *Triple-DES* algorithm but other algorithms (e.g., *AES*) could be used for this purpose. Thus, a plaintext block *b* at position *p* in the document is encrypted by $E_k(b \oplus p)$, where *k* is a secret key attached to the document and stored in the SOE. Key *k* can be permanently stored in the SOE or can be downloaded securely along with the access control rules attached to a given (document, user) pair.

Encryption alone is not sufficient to guarantee the document integrity since the attacker can perform random modifications and substitutions in the ciphertext. We propose in the

following a solution where the integrity is checked by the SOE in cooperation with the untrusted terminal. A basic solution would be to split the document in chunks (e.g., 2KB) and use a collision resistant hash function (e.g., SHA-1) to compute a digest of each chunk, called *ChunkDigest* that prevents any tampering to occur without impacting this digest. Each chunk contains an identifier reflecting its position in the document, so that block substitutions can be easily detected. When the SOE accesses *n* bytes at position *pos* in a chunk, the terminal computes the hash of the *pos-1* preceding bytes and transmits its intermediate result to the SOE. Since hashing is computed incrementally, the SOE can continue the hashing computation on encrypted data and checks the integrity of the received data by comparing the final hashed value to *ChunkDigest*. Remark that *ChunkDigest* must be encrypted to prevent the terminal to compute by itself a new digest corresponding to tampered data. This solution incurs to communicate *sizeof(ChunkDigest) + sizeof(Chunk) – (pos-1)* bytes to the SOE and to decrypt *sizeof(ChunkDigest) + n* bytes in the SOE.

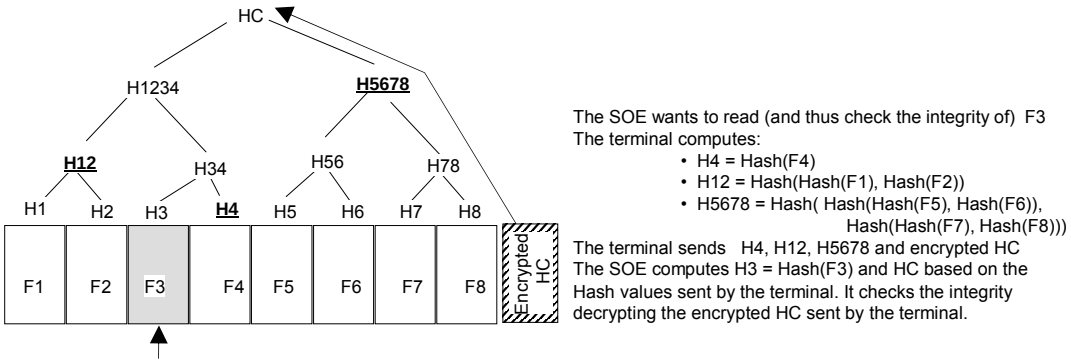


Figure 35. Random integrity checking

Although correct, the previous solution reduces the benefit of small skips in the document since the target chunk must always be read by the SOE from the position *pos* of interest until its end. Thus, *sizeof(Chunk) – (pos-1) – n* irrelevant bytes have to be transferred to the SOE. To alleviate this drawback, we adapt the *Merkle hash tree* principle introduced in [78] as follows. Each chunk is divided into *m* fragments (e.g., of 256 bytes), where *m* is a power of 2, and these *m* fragments are organized in a binary tree. A hash value is computed for each fragment and then attached to each leaf of the binary tree. Each intermediate node of the tree contains a hash value computed on the concatenation of its children hash value. The *ChunkDigest* corresponds to the hash value attached to the binary tree root. When the SOE accesses *n* bytes at position *pos* in a fragment *f* of a given chunk, the terminal sends: 1) the bytes from *pos* up to the end of fragment *f*, that is *sizeof(fragment) – (pos–1)* bytes including the *n* bytes of interest; 2) the intermediate hash computation of the *pos-1* first bytes of fragment *f*; 3) the hash information computed on the other fragments following the *Merkle hash tree* strategy; and 4) the encrypted *ChunkDigest*. Thanks to this information, the SOE can recompute the root of the Merkle hash tree and compare it to *ChunkDigest* as pictured in Figure 35.

To conclude, the document is protected against tampering and confidentiality attacks while remaining agnostic regarding the encryption algorithm used to cipher the elementary data. Unlike the work of Bouganim and Pucheral [20] and Hacigumus et al. [62], we make no assumption on any particular way of encrypting data that could facilitate the query execution at the price of a weaker robustness against cryptanalysis attacks.

5.2 Secure access right management

As stated in the introduction, dynamicity of the access control policies is a mandatory feature for a number of applications. This led us to design a secure mechanism to refresh the access control rules on the SOE. Just like the data, the definition of the access control rules can be stored encrypted on the server. However, the server may be untrustworthy and give an incorrect answer to the SOE when it asks for an update of the access control rules or for a document. Confidentiality, integrity and authenticity are enforced thanks to encryption and hashing mechanisms presented in Section 5.1. In this section, we focus on replay attacks. Replay attacks consist in sending to the SOE inconsistent versions of the access control rules and the document in order to mislead the access control rules evaluator. Replay attacks do not hurt confidentiality as long as no unauthorized access is gained by a user. For instance, if both the document and the access control rules are delivered in an old and consistent version, the user will get exactly the same result that he got in the past. Even if these data are no longer authorized at the present time, the user does not gain access to new information.

The problem arises when access control rules are defined over parts of a document which may be updated. Applying a new access control policy on an old document may reveal unauthorized outdated data. Similarly, applying an outdated access control policy on a recent version of the document may reveal unauthorized up to date data.

Both problems may be solved using a cross reference versioning between the data and the access rights. The access rights of every user[7] are stored on the server in a form of a 4-uple (*ts, right, docts, sig*) where *ts* is a timestamp incremented for every new access control rule, *right* is the access control rule definition in its encrypted form, *docts* the timestamp of the document at the time the access control rule was defined and *sig* is a signature of the tuple. Conversely, the document contains in a signed header its timestamp *docts* as well as the last access right timestamp for each user at the time the document was updated.

When the SOE asks for access right updates, it gets all the access control rules of the associated user with a *ts* greater than the one he got from its last connection. Then it checks that the access rights are properly chained thanks to their timestamp (no rules are missing). When it gets the document, it checks that the document *docts* (resp. *ts*) is greater (resp. smaller) than the *docts* (resp. *ts*) contained in the last access right, thus ensuring that the document and the access rights are consistent even though they may not be up to date.

[7] If many users are involved, a table can be shared by a group of users taking into account common access rights and individual access rights.

6 Conclusion

In this chapter, we provided an integrated solution to secure the access control of XML data in an insecure infrastructure. Compared to existing solutions, we clearly separated the access rights management from encryption in order to provide personalized and dynamic access control rights. The drastic constraints of the SOE led us to devise new streaming algorithms and indexing techniques to minimize the consumption of RAM and of the bandwidth. Therefore, to reduce the RAM consumption we designed a streaming evaluator able to manage propagations and conflicts of access control rules, and detect situations where some of them become irrelevant. To reduce the communication cost, we proposed a compact *Skip Index* embedded into the input document which takes advantage of these situations to skip irrelevant or forbidden data. We also proposed a graceful management of pending predicates, compliant with the memory capacity of the SOE which accommodates two kind of streams. Due to the untrusted natures of the infrastructure, we proposed a technique to make the integrity of the data verifiable despite random accesses and a secure mechanism to refresh the access rights. In the next chapter we show the feasibility and the effectiveness of our solution thanks to prototypes and performance evaluation, and describe how it can be beneficial in different application contexts.

Chapter 5 – Evaluation and experience

1 Introduction

Our solution described in Chapter 4 can be instantiated in several SOE (e.g., secure co-processor, secure chip, secure tokens, smart cards, etc.). Because smart cards have proven to be versatile and cost effective components, we used them to conduct our experiments.

This chapter is organized as follows. First, we recall in Section 2 the internal architecture described in the previous chapter and instantiate it on smart cards. Second, we measure in Section 3 the efficiency of our solution on an advanced smart card platform thanks to a hardware simulator. Then, in Section 4, we show the feasibility of our approach on current smart cards. In Section 5, we describe different application contexts that can benefit from our solution. Section 6 provides a description of a previous work to secure the sharing of relational data. Finally, Section 7 concludes.

2 Internal architecture

As said in Chapter 2, the smart card communicates with the external world through APDU (256 byte messages) in a half-duplex mode. The architecture of the engine running in the smart card is depicted in Figure 36. The different modules involved in the evaluation are as follows:

- *Integrity module:* This module receives data from the terminal which ensures the connectivity between the server and the smart card. As said in Section 5.1 of Chapter 4, the role of the terminal is also to contribute in the integrity checking by computing partial hashes. To this end, the terminal creates APDU messages (256 bytes) which contain encrypted XML data from the server (including signatures embedded in the data) and partial hashes. The *Integrity module*, then checks the integrity of these data using techniques developed in Section 5.1 of Chapter 4.
- *Decryption module:* The encrypted XML data are then decrypted thanks to a set of decryption keys. These keys can be, for instance, retrieved from the untrusted server, encrypted using the public key of the user.

- *Skip Index decoder:* A parser analyzes the XML data, decodes the *Skip Index* data and raises SAX events (*startElement*, *endElement* and *characters*) thanks to a stack which records the tags present in the current subtree, and its size (see Section 3 of Chapter 4).

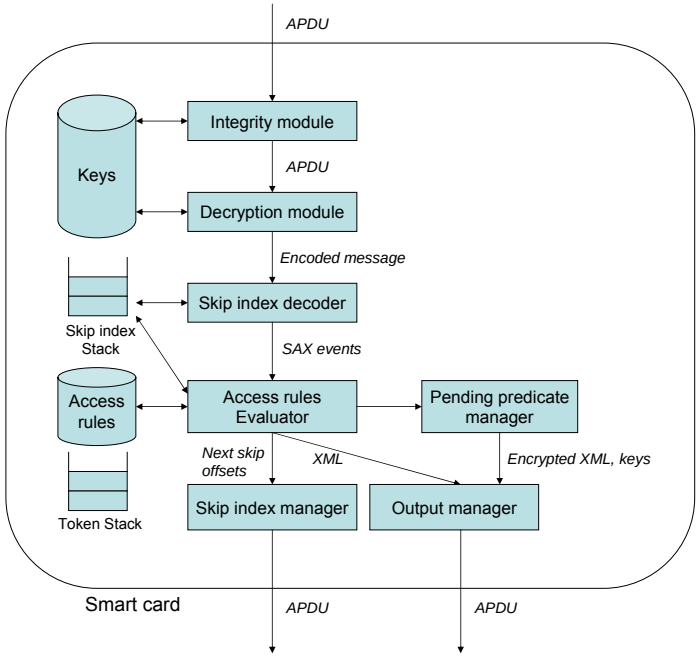


Figure 36. Internal architectures

- *Access rules evaluator:* Thanks to the SAX events, the access rules are evaluated as described in Section 2 of Chapter 4. It also takes into account the *Skip Index Stack* to determine beforehand if a rule may apply or not in the current subtree. Parts to be delivered are then sent to the *Output Manager*. The set of access rules can be updated, for instance, whenever the user connects to the server.

- *Pending Predicate Manager:* This module is in charge of managing pending predicates and the associated pending subparts of data. In the case of strict streaming, it manages the different temporary encryption keys associated to pending parts. These keys are then delivered or discarded when the issues of the pending parts are known. In the case of backward/forward streaming access, it manages a pending list to skip pending subparts and gets back to them when their issues are known. See Section 4 of Chapter 4 for more details.

- *Skip index manager:* Based on the *Token Stack*, the access rules evaluator may find a subtree to be irrelevant and decide to skip it. In the case of strict streaming access, it can tell the terminal to ignore some parts of the stream to skip a subtree. In the case of backward/forward streaming access, it may tell the server via APDU to the terminal to do a backward or forward access in the stream.

- *Output manager:* It formats the message into an APDU to be sent back to the terminal. It

is in charge of encrypting the pending subparts of XML data and to deliver the keys to the terminal when considering strict streaming. In the case of XML fragments, they are formatted in a way that they can be integrated consistently in the document view by the terminal.

3 Performance evaluation

While existing smart cards are already powerful (32-bit CPU running at 30 MHz, 4 KB of RAM, 128KB of EEPROM), they are still too limited to support our architecture, especially in terms of communication bandwidth (9.6Kbps). Our industrial partner, Axalto, announces by the end of this year a more powerful smart card equipped with a 32-bit CPU running at 40Mhz, 8KB of RAM, 1MB of Flash and supporting a USB protocol at 1MBps (operating in a half-duplex mode). Axalto provided us with a hardware cycle-accurate simulator for this forthcoming smart card. Our prototype has been developed in C and has been measured using this simulator. Cycle-accuracy guarantees an exact prediction of the performance that will be obtained with the target hardware platform.

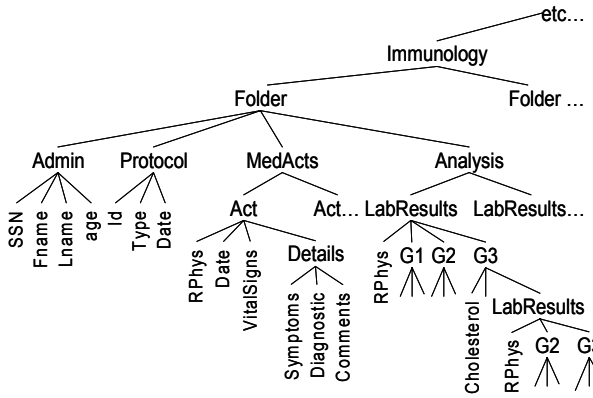
The measures are obtained from both synthetic and real datasets. We first define a Hospital document containing medical folders which is representative of data requiring a high level of confidentiality, and personalized and dynamic rights. Then, we give details about the experimentation platform. Then, we analyze the storage overhead incurred by the *Skip Index* and compare it with possible variants. Next, we study the performance of access control evaluation, query evaluation and integrity checking. Finally, the global performance of the proposed solution is assessed on four datasets that exhibit different characteristics.

3.1 Hospital document

For the purpose of our experiment, we defined a Hospital document composed of medical folders as well as three rather different profiles (selective vs. non-selective, complex vs. simple, recursive vs. non recursive rules). The choice of a medical document appeared to be relevant in our context since it requires a high level of confidentiality as well as dynamic and personalized access control rules. Indeed, medical data are protected by the need-to-know principle which is to give only the necessary information to a user, thus requiring personalized and fine-grained access. Regarding the dynamicity of data, a researcher may be granted an exceptional and time-limited access to some particular fragments of medical folders where the rate of *Cholesterol* exceeds 300mg/dL (a rather rare situation). Moreover, a patient having subscribed to a protocol to test the effectiveness of a new treatment may revoke this protocol at any time due to a degradation of her state of health or for any other personal reasons.

However, real datasets are very difficult to obtain in this area. Therefore, we built the Hospital document (pictured in Figure 37) with the help of Dr. Anaenza Maresca, physician

at the Tenon hospital (Paris) according to her own experience.



Doctor access control policy
D1: ⊕, //Folder/Admin D2: ⊕, //MedActs[//RPhys = USER] D3: ⊖, //Act[RPhys != USER]/Details D4: ⊕, //Folder[MedActs//RPhys = USER]/Analysis
Researcher access control policy
R1: ⊕, //Folder[Protocol]//Age R2: ⊕, //Folder[Protocol/Type='G3']//LabResults//G3[Cholesterol < 250] Rules 2 occur for each of the 10 groups {G1, …, G10}
Secretary access control policy
S1: ⊕, //Admin

Figure 37. Hospital XML document

The medical folders and different profiles are pictured in Figure 37, along with the access control policies associated to three profiles of users: secretaries, doctors and medical researchers. A secretary is granted access only to the patient's administrative subfolders. A doctor is granted access to the patient's administrative subfolders and to all medical acts and analysis of her patients, except the details for acts she did not carry out herself. Finally, a researcher is granted access only to the laboratory results and the age of patients who have subscribed to a protocol test of type *G3*, provided the measurement for the element *Cholesterol* does not exceed 250mg/dL.

We generated the document thanks to the ToXgene generator [110].

3.2 Experimentation platform

As this section will make clear, our solution is strongly bounded by the decryption and the communication costs. The numbers given in Table 1 allow projecting the performance results given in this section on different target architectures. The number given for the smart card communication bandwidth corresponds to a worst case where each data entering the SOE takes part in the result. The decryption cost corresponds to the 3DES algorithm, hardwired in the smart card (line 1) and measured on a PC at 1 GHz (lines 2 and 3).

Context	Communication	Decryption
Hardware based (e.g., future smartcards)	0.5 MB/s	**0.15 MB/s**
Software based - Internet connection	**0.1 MB/s**	1.2 MB/s
Software based - LAN connection	10 MB/s	**1.2 MB/s**

Table 1. Communication and decryption costs

In the experiment, we consider three real datasets: *WSU* corresponding to university courses, *Sigmod records* containing index of articles and *Tree Bank* containing English sentences tagged with parts of speech [114]. In addition, we considered the Hospital document described in Section 3.1. The characteristics of interest of these documents are summarized in Table 2.

	WSU	Sigmod	Treebank	Hospital
Size	1.3 MB	350KB	59MB	3.6 MB
Text size	210KB	146KB	33MB	2,1 MB
Maximum depth	4	6	36	8
Average depth	3.1	5.1	7.8	6.8
# distinct tags	20	11	250	89
# text nodes	48820	8383	1391845	98310
# elements	74557	11526	2437666	117795

Table 2. Documents characteristics

3.3 Index storage overhead

The *Skip Index* is an aggregation of three techniques for encoding respectively tags, lists of descendant tags and subtree sizes. Variants of the *Skip Index* could be devised by combining these techniques differently (e.g., encoding the tags and the subtree sizes without encoding the lists of descendant tags makes sense). Thus, to evaluate the overhead ascribed to each of these metadata, we compare the following techniques. NC corresponds to the original *Non Compressed* document. TC is a rather classic *Tag Compression* method and will serve as reference. In TC, each tag is encoded by a number expressed with log_2*(#distinct tags)* bits. We denote by TCS (*Tag Compressed and Subtree size*) the method storing the subtree size to allow subtrees to be skipped. The subtree size is encoded with log_2*(compressed document size)* bits. In TCS, the closing tag is useless and can be removed. TCSB complements TCS with a bitmap of descendant tags encoded with *#dictinct tags* bits for each element. Finally, TCSBR is the recursive variant of TCSB and corresponds actually to the *Skip Index* detailed in Section 3 of Chapter 4. In all these methods, the metadata need be aligned on a byte frontier. Figure 38 compares these five methods on the datasets introduced formerly. These datasets having different characteristics, the Y-axis is expressed in terms of the ratio *structure/(text length)*.

Clearly, TC drastically reduces the size of the structure in all datasets. Adding the subtree size to nodes (TCS) increases the structure size by 50%, up to 150% (big documents require

an encoding of about 5 bytes for both the subtree size and the tag element while smaller documents need only 3 bytes). The bitmap of descendant tags (TCSB) is even more expensive, especially in the case of the Bank document which contains 250 distinct tags. TCSBR drastically reduces this overhead and brings back the size of the structure near the TC one. The reason is that the subtree size generally decreases rapidly, as well as the number of distinct tags inside each subtree. For the Sigmod document, TCSBR becomes even more compact than TC.

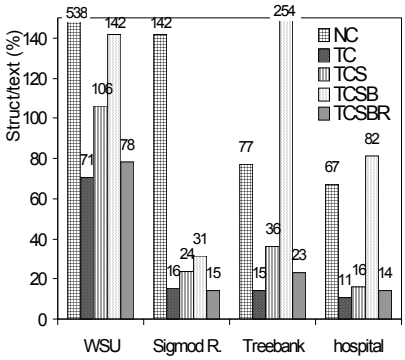


Figure 38. Index storage overhead

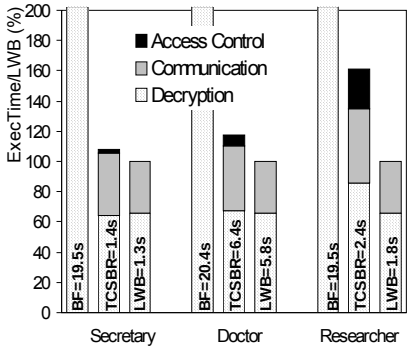


Figure 39. Access control overhead

3.4 Access control overhead

To assess the efficiency of our strategy (based on TCSBR), we compare it with: (i) a *Brute-Force* strategy (BF) filtering the document without any index and (ii) a time lower bound LWB. LWB cannot be reached by any practical strategy. It corresponds to the time required by an oracle to read only the authorized fragments of a document and decrypt it. Obviously, a genuine oracle will be able to predict the outcome of all predicates – pending or not – without checking them and to guess where the relevant data are in the document. Figure 39 shows the execution time required to evaluate the authorized view of the three profiles (Secretary, Doctor and Researcher) on the Hospital document. Integrity checking is not taken into account here. The size of the compressed document is 2.5MB and the evaluation of the authorized view returns 135KB for the Secretary, 575KB for the Doctor and 95 KB for the Researcher. In order to compare the three profiles despite this size discrepancy, the Y-axis represents the ratio between each execution time and its respective LWB. The real execution time in seconds is mentioned on each histogram. To measure the impact of a rather complex access control policy, we consider that the Researcher is granted access to 10 medical protocols instead of a single one, each expressed by one positive and one negative rule as pictured in Figure 37 for the Hospital document

The conclusions that can be drawn from this figure are threefold. First, the *Brute-Force* strategy exhibits dramatic performance, explained by the fact that the smart card has to read and decrypt the whole document in order to analyze it. Second, the performance of our

TCSBR strategy is generally very close to the LWB (let us recall that LWB is a theoretical and unreachable lower bound), exemplifying the importance of minimizing the input flow entering the SOE. The more important overhead noticed for the Researcher profile compared to LWB is due to the predicate expressed on the protocol element that can remain pending until the end of each folder. Indeed, if this predicate is evaluated to false, the access rule evaluator will continue – needlessly in the current case – to look at another instance of this predicate (see Section 4 of Chapter 4). Third, the cost of access control (from 2% to 15%) is largely dominated by the decryption cost (from 53% to 60%) and by the communication cost (from 30% to 38%). The cost of access control is determined by the number of active tokens that are to be managed at the same time. This number depends on the number of ARA in the access control policy and the number of descendant transitions (//) and predicates inside each ARA. This explains the larger cost of evaluating the Researcher access control policy.

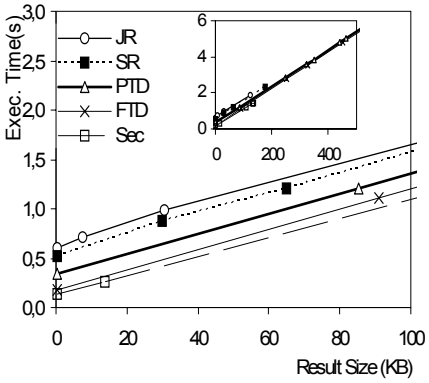


Figure 40. Impact of queries

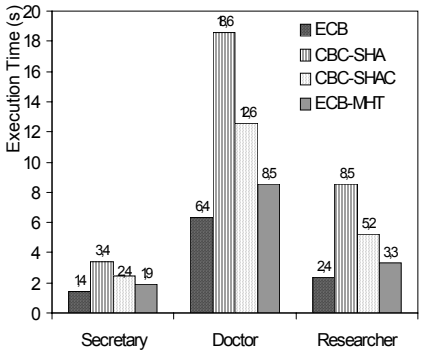


Figure 41. Impact of integrity control

3.5 Impact of queries

To measure accurately the impact of a query in the global performance, we consider the query *//Folder[//Age>v]* (v allows us to vary the query selectivity), executed over five different views built from the preceding profiles and corresponding to: a secretary (S), a part-time doctor (PTD) having in charge few patients, a full-time doctor (FTD) having in charge many patients, a junior researcher (JR) being granted access to few analysis results and a senior researcher (SR) being granted access to several analysis results. Figure 40 plots the query execution time (including the access control) as a function of the query result size. The execution time decreases linearly as the query and view selectivity's increase, showing the accuracy of TCSBR. Even if the query result is empty, the execution time is not null since parts of the document have to be analyzed before being skipped. The parts of the document that need be analyzed depend on the view and on the query. The embedded figure shows the same linearity for larger values of the query result size.

3.6 Evaluation of the integrity control

Figure 41 depicts the execution time required to build the authorized view of the Secretary, Doctor and Researcher profiles, including integrity checking. Comparing these results with Figure 39 shows that the cost ascribed to integrity checking remains quite acceptable when using the technique proposed in Section 5.1 of Chapter 4 (from 32% to 38%). To better capture the benefit of this technique, based on ECB and Merkle Hash Tree (ECB-MHT), we compare it with: (ECB) a basic ECB encryption scheme without hashing that enforces confidentiality but not tamper-resistance; (CBC-SHA) a CBC encryption scheme complemented by a SHA-1 hashing applied to the clear text version of complete chunks (this solution represents the most direct application of state-of-the-art techniques); (CBC-SHAC) that is similar to CBC-SHA except that the hashing applies to ciphered chunks, thereby allowing the SOE to check the chunk digest without decrypting the chunk itself. The results plotted in the figure are self-explanatory.

3.7 Performance on real datasets

To assess the robustness of our approach when different document structures are faced, we measured the performance of our prototype on the three real datasets WSU, Sigmod and Bank. For these documents we generated random access rules (including // and predicates). Each document exhibits interesting characteristics. The Sigmod document is well structured, non-recursive, of medium depth and the generated access control policy was simple and not much selective (50% of the document was returned). The WSU document is rather flat and contains a large amount of very small elements (its structure represents 78% of the document size after TCSBR indexation).

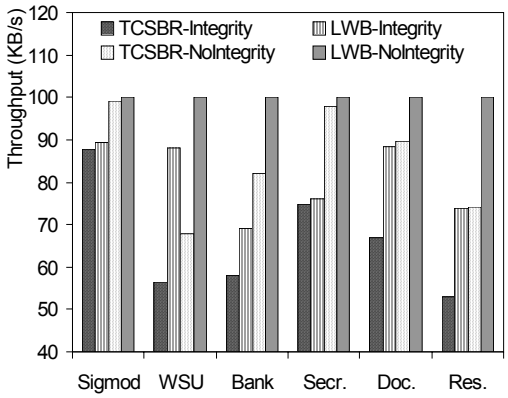


Figure 42. Performance on real datasets

The Bank document is very large, contains a large amount of tags that appear recursively in the document and the generated access control policy was complex (8 rules). Figure 42 reports the results. We added in the figure the measures obtained with the Hospital

document to serve as a basis for comparisons. The figure plots the execution time in terms of throughput for our method and for LWB, both with and without integrity checking. We show that our method tackles well very different situations and produces a throughput ranging from 55KBps to 85KBps depending on the document and the access control policy. These preliminary results as encouraging when compared with xDSL Internet bandwidth available nowadays (ranging from 16KBps to 128KBps).

We provided in this section a thorough evaluation of our solution on advanced smart cards. From these experiments, we showed (1) that the *Skip Index* space overhead is negligible and contributes to some extent in the compression of the whole document. (2) that the access control evaluation time overhead is very small compared to the decryption and communication time. (3) that the integration of the query in the evaluation can be beneficial. (4) that the integrity checking proposed in our solution has a reasonable overhead. Finally, (5) that our solution accommodates well various kind of data taken from real applications.

4 Feasibility of the approach

To demonstrate the feasibility of our approach on current secure devices, we implemented a prototype in Java Card running on e-gate smart cards that we presented at the Sigmod Conference 2005. The choice of the Java Card languages ensures full compatibility with many smart cards on the market.

As said in the previous section, current smart cards have limited performance. This is emphasized by the use of Java Card. Indeed, the Java Card Virtual Machine consumes most of the RAM, so only a few amount of RAM is left to the applications. Consequently, the EEPROM memory is used to palliate the lack of volatile memory. Recall that EEPROM write accesses are very slow. Moreover, the execution is slower since Java Card is interpreted by a virtual machine. According to our experience, in the PicoDBMS project developed by our team, the execution time obtained from a prototype in native C and running in a smart card with an optimized Operating System (dedicated to the prototype) was about a hundred times faster than its Java Card counterpart running on a normal smart card.

Our objective was to demonstrate the functionalities of our evaluator in current smart card environments and not to achieve good performance. For the purpose of the demo, we proposed two execution modes:

- real execution running on the smart card. This execution displays the input document, the access rights and the resulting view. Depending on the smart card, the authorized view is different. Moreover, we show the effectiveness of our solution regarding access right updates.
- detailed execution to display internal structures and data processing taking place inside the smart card. Because the smart card is a black box, there is no way to visualize the

internal structures easily. To tackle this problem, we developed a smart card simulator in Java to run Java Card programs on the PC. This simulator is then connected to a graphical interface which displays internal information.

The interface pictured in Figure 43 allows to execute the program step by step and see data access and how internal structures evolve. We describe below the different graphic components:

1. *Encoding of the XML document*. It is represented by an array distinguishing structural data (elements and skip index) and values (text nodes). This panel shows which parts of the data are accessed in order to exhibit the benefits of the *Skip Index*.

2. *Tree representation of the XML document*. This panel shows the outcome of the elements: discarded, delivered and pending.

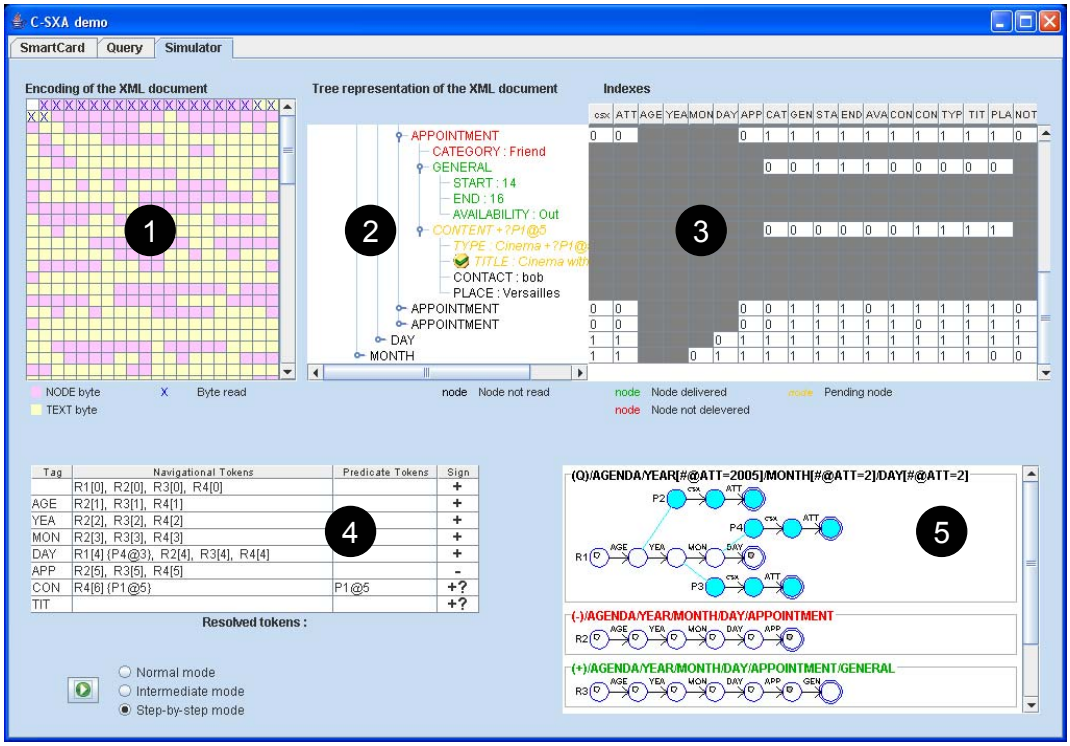


Figure 43. Detailed execution screenshot

3. *The Skip Index set of descendant tags*. Each row gives the set of descendant tags of the element of the subtree on the left. The list is given in its bit array representation where each bit corresponds to a tag. In this representation, only the cell in white are actually encoded; gray cells means that the tag is implicitly not present according to ancestors tag list. A cell is set to one if the corresponding tag is present. This panel allows to see the recursive compression of the descendant tag bit array.

4. *Token stack*. It displays the tokens referenced in the token stack. In addition, it

displays the current pending predicates, the tag stack and the sign after conflict resolution. This allows seeing the propagation of rules and conflicts.

5. Access rules automata. For each of them, we can visualize tokens and token proxies at every step of the execution. This allows to see when rules are applying (all the final states of both the navigational and predicate path have been reached) and when they are pending (the final states of the navigational path has been reached but not all of its predicate path).

The execution was illustrated on a collaborative work application (an electronic calendar) and a DRM application. We describe more in details these two applications in the next section.

5 Application contexts

The technology developed in this thesis can be exploited in various application contexts. We present in this section how our solution can accommodate data sharing applications, secure portable folder, and DRM. For each of them, we show how our technology can bring new benefits, then we explain how they can be integrated in the applications. The application contexts which are addressed in the following can be indifferently instantiated on computers and mobile devices infrastructure. For this reason, we developed some applications running on computers and some others on cell phones. The purpose of this section being to give an overview of the different contexts, we give a lesser importance to the technical aspects. In this section, we rely on the Java Card prototype described in the previous section.

5.1 Data sharing applications

The development of new devices, digital camera, cell phones, webcam generates a vast amount of information that can be shared within a community of users. That led many companies to design shared photo album systems and shared briefcases to facilitate the sharing of information. In this context, the data are traditionally stored in a central server in charge of enforcing the access control on the data. However, as discussed in the previous chapters, the server can be compromised by internal or external attacks.

We illustrate how our technology can tackle this problem on a shared electronic calendar application (as pictured in Figure 44) which is representative of applications where the access rights are dynamic (e.g, they evolve as the users make new relations) and personalized (e.g., grant access to my personal notes only to colleagues who are present at an appointment). This demonstration has been presented at the e-gate open contest 2004 and was rewarded with a Silver award [26].

We designed the electronic calendar application in Java and we provided an interface to

add/modify/remove appointments and setup the sharing policies for other users. The access rights can be set for a group (e.g., family, friend, and colleague) or a particular user using a ruler which can be positioned on five levels of confidentiality ranging from unauthorized access to complete access as shown in Figure 45. The program then translates these access levels into XPath rules.

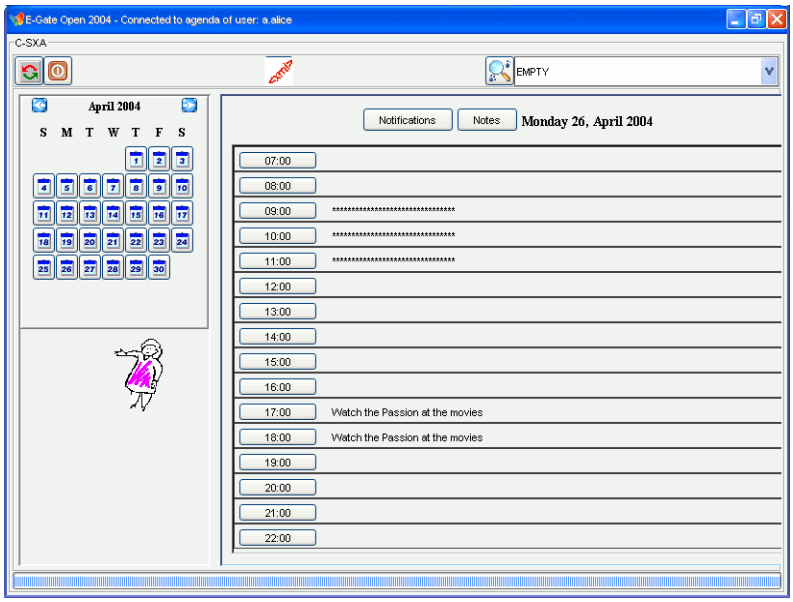


Figure 44. Electronic calendar remote access

To demonstrate the effectiveness of our solution, we showed that the authorized view of a remote calendar of another user changes as soon as the access rights evolve. In Figure 44, we show that Bob has a restricted access on Alice agenda for some appointments (the subject of the appointment between 9:00 a.m. and 11:00 a.m. is hidden).

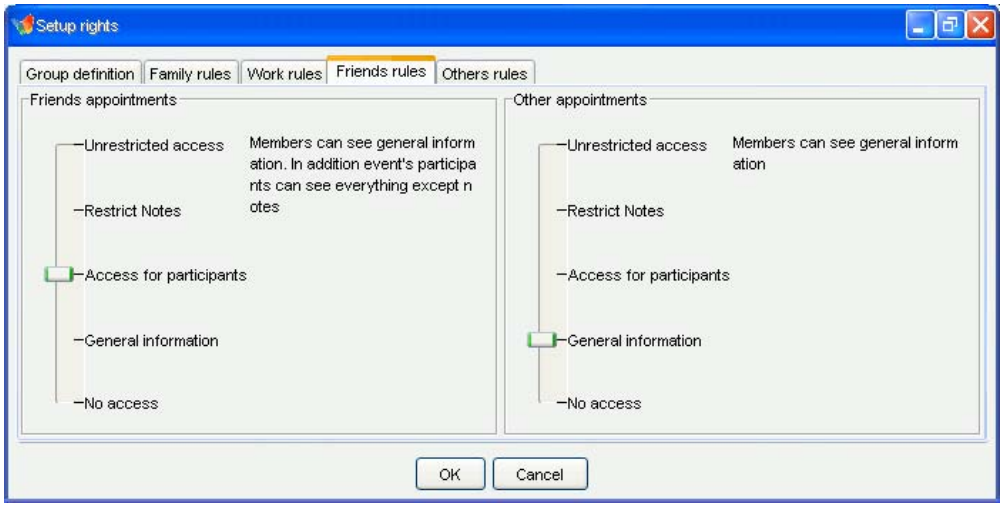


Figure 45. Electronic calendar access right management

5.2 Secure Portable Folder

The rapid growth of smart card stable storage capacity (from kilobytes to megabytes soon) makes the management of on-board data realistic and more and more attractive for a wide range of applications. The concept of secure portable folder has emerged with the idea of carrying a patient's medical history in a smart card [105]. Since then, the value of smart cards to secure and share in a controlled way personal information has been recognized in several domains like education (scholastic folders), commerce (loyalties), telecommunication (address book) or mobile computing (user's profiles containing licenses, passwords, bookmarks, etc).

Providing access control management on on-board data allows defining different views of the same on-board data for different users (secretary, colleagues, family) or software (web browser, address book manager, …) willing to access these data. This functionality is required by applications like secure portable folders (different users may query, modify and create data in the holder's folder) or virtual home environment (different softwares have a partial access to the holder's environment).

While the need for on-board data management and sharing facilities is clearly established, few technical solutions have been proposed yet. Existing solutions are generally application specific, store data sequentially, allow basic searches and protect data thanks to passwords. The Structured Card Query Language (SCQL) standard [71] is a first attempt to define database-style storage techniques and privileges. PicoDBMS [96][6], designed by our team, was the first full-fledged relational database system embedded in a smart card, supporting a robust subset of the SQL standard (and then encompassing SCQL). PicoDBMS is, however, a complex technology primarily designed to manage efficiently huge and well-structured embedded folders. The versatility and wide acceptance of XML [119] makes this standard the best candidate today to describe, organize, store and share the variety of data that appear in the above applications.

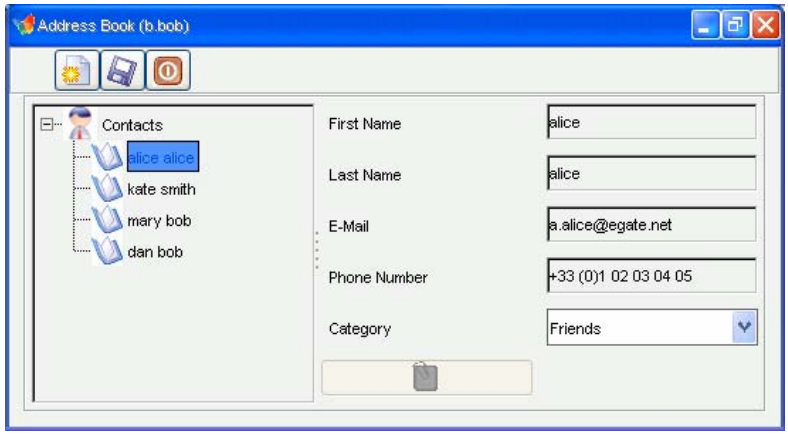


Figure 46. Address book application

Our solution can easily suit this need, considering a few changes in our architecture. Indeed, data as well as access rights need not be retrieved from an external server and are stored inside the smart card instead. To this end, we implemented a data store manager in charge of storing XML data inside the smart card. More precisely, it performs basic tasks such as storing, updating and erasing a block of data considering issues related to data fragmentation. These issues are outside of the scope of this thesis and are not discussed here. In this solution, instead of storing the data in an encrypted form on an untrusted server, the data are stored in a clear-text compressed form in EEPROM. The access rights are set up by the owner through a simple interface and when a user retrieves data, the access rule evaluator takes as input the data stored in the smart card, evaluate the access control rules and the query, and delivers the authorized view to the user. The smart card can be used by several users and/or applications, and different access can be granted depending on the user connected to it (e.g., members of the family have access to names and phone numbers while the secretary has a complete access of all her colleagues). We evaluated this approach on an address book (see Figure 46) which is representative of applications that needs be accessible at any time. This application was part of our solution demonstrated at e-gate 2004 [26].

5.3 DRM applications

The development of decentralized ways of sharing such as P2P systems made the distribution of media files out of control for the media distributors and became a serious threat for the global multimedia content industry. The first attempt to combat piracy was to publish media on uncopyable CDs/DVDs. However, it was too restrictive, considering that people could not copy their music for their personal usages (e.g., mp3 players, compilations). Consequently, it turned against its initial objective and was rather an incentive to download illegal copies from the Internet. The second attempt was based on a fairer model for the consumers. The idea is to provide a way to pay and download from the Internet only the media the consumer is interested in (e.g., pay for only 2 tracks of a full album) and to allow a limited number of copies (e.g., allow to copy a track 4 times). While this solution seems attractive, it is still too restrictive since data cannot be copied on all consumer's personal devices (mp3 players, phones, computers, …) and cannot be shared as it was the case with classical physical media.

5.3.1 Fair use

Based on these observations, we proposed MobiDiQ, a DRM engine for mobile phone based on our solution which implements fair use rules. This solution was presented at the SIMagine 2005 contest [24] and received the Gold Award. Fair use aims at reconciling the content providers' and mobile consumers' point of views by giving the ability to develop fair business models (i.e., that preserve the interest of both parties). The different interests can be stated as follows. First media distributors must have strong guarantees against piracy. Second, consumers are expecting to pay for the exact content they are interested in rather than for complete commercial packages. Third,

once the content has been legally acquired it can be made available on any device owned by the consumer without any restriction. Fourth, as members of different communities (family, friends, colleagues, clubs), users are expecting a reasonable way to exchange (loaning, gifting) digital assets as they would with classical physical media while preventing large-scale distribution (e.g., P2P). Fifth, some categories of citizen (e.g., students, needy persons, artists) may have the opportunity to access valuable content at a special rate, assuming commercial agreements took place between institutions (government, universities, associations) and content providers. Sixth, the consumer may have the opportunity to control the contents and the usage of these contents made by her children. Finally, the privacy attached to all the aforementioned practices must be strictly preserved (services used, videos watched, etc).

In the following, we present the architecture and various scenarios to show how we implemented fair use to preserve the interests of both consumers and media industries and finally illustrate them on a parental control application.

5.3.2 Architecture

The overall infrastructure of MobiDiQ is depicted in Figure 47. We suppose that each user has a multimedia cell phone including a SIM card. The SIM card embeds an access controller and the user's profile (detailed in the next subsections). Basically, the users acquire encrypted multimedia content and encrypted licenses from content providers. The licenses are decrypted by the SIM card and checked by the MobiDiQ engine, taking into account the metadata describing the multimedia content, the user's profile and potential historical data (all expressed in XML). Depending on the outcome of the license verification, MobiDiQ may allow "playing" (part of) the multimedia content. To this end, the decryption key(s) required to decrypt the multimedia content is extracted from the metadata.

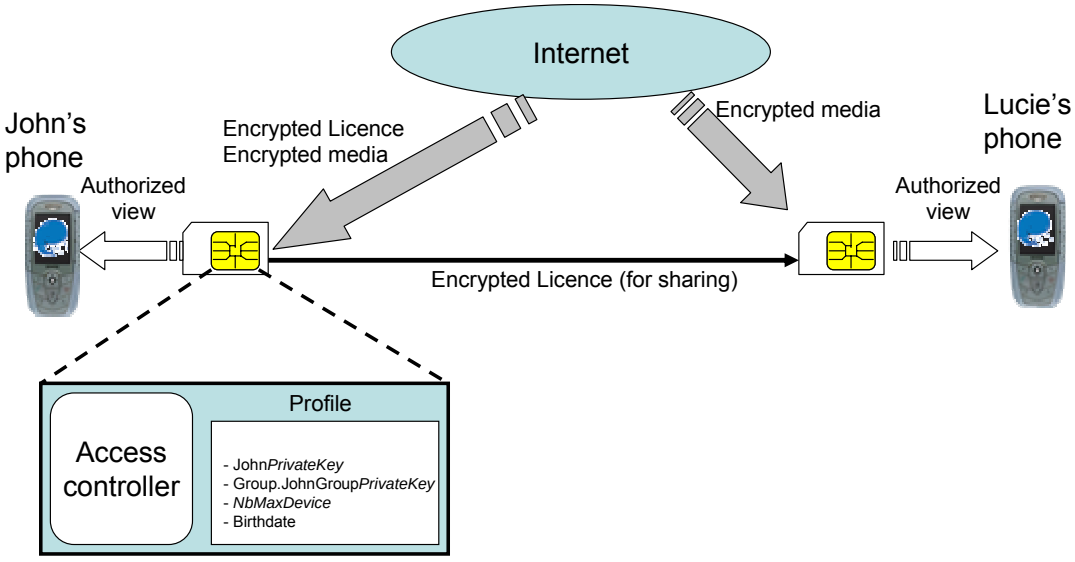


Figure 47. MobidiQ license management

User Friendliness scenarios

Private licenses: Let assume that John Doe wants to listen a given song on his cell phone. A classic way to ensure that John can acquire a private license is to encrypt it with John's SIM card public key (assuming a PKI infrastructure is used). Therefore, the unique way to play the song is to decrypt the license with the corresponding private key, owned and protected by John's SIM card. While this solution is satisfactory for private licenses used on a single device, it must be extended to enable multi-device usage (e.g., cell phone, car radio, mp3 player, etc).

Multi-device usage: to handle that case, MobiDiQ allows creating a group of devices identified by a *GroupPrivateKey*, and acquiring licenses encrypted with the corresponding *GroupPublicKey*. The maximum number of devices belonging to the group, say *NbMaxDevice*, must be limited to avoid illegal mass sharing. John asks his phone SIM card to create the group *JohnGroup*. MobiDiQ then adds two fields to John's profile: (1) the Group.JohnGroup*PrivateKey* (obtained through a PKI infrastructure); (2) the Group.JohnGroup*Count* recording the current number of devices in the group. Adding the PDA to John's group is possible if the condition (Group.JohnGroup*Count* < *NbMaxDevice*) holds. In that case the Group.JohnGroup*PrivateKey* is transmitted from the phone card to the PDA secure component (either smart card or secure chip).

Familial usage: The group mechanism explained above can be used as well to allow media sharing between family members. To prevent illegal mass sharing, it is however important to restrict the number of groups a device can belong to. Each device can join a rather reduced number of groups, typically a single one corresponding to the direct family circle (parents and their children). Consequently, the *NbMaxDevice* limit can be set to a much larger value (e.g., 10 to 15) to encompass all the family's devices and the number of transfers can be unlimited.

License Loaning: Let's assume that Lucie wants to loan for a week a given license to her friend Julie. MobiDiQ handles this case in three steps. First, a specific record is added to Lucie's profile (on her SIM card) to disable a personal use of that license for one week. Second, the license is downloaded on Lucie's SIM card, decrypted, updated to include a one week validity limit, re-encrypted using Julie's public key, and finally uploaded on Julie's cell phone. Julie can now retrieve her private license and listen freely the music during one week.

5.3.3 The demonstrator

In order to support the fair DRM features, we enhanced our initial prototype adding the management of licenses and profiles. Our engine is embedded into a SIM card which is plugged into a cell phone. Our demonstration shows how licenses are expressed, exchanged and evaluated in a MobiDiQ enabled cell phone when playing a video sequence under strict

parental control (violent scenes are withdrawn from the video stream). To this end, we developed a video player in C++ on the cell phone.

We assume that the media file sequences are rated with different levels of violence, sex and language as pictured in Figure 48. This can be done by a public national institute in charge of rating movies. An interface is provided for the parents to define easily the profile of their children as pictured in Figure 49.

When a child wants to see a movie on her cell phone, her SIM cards will first get the parental control license that gives her the authorization to watch the movie. The license describes the condition to access a media. In this case, it tells that sequences of a movie may be played if the sequence ratings are compliant with the child profile. Then, the movies as well as metadata describing the movies are sent to the SIM card. The MobiDiQ engine then processes the license on the metadata, using some information from the profile and delivers only the authorized parts of the movies.

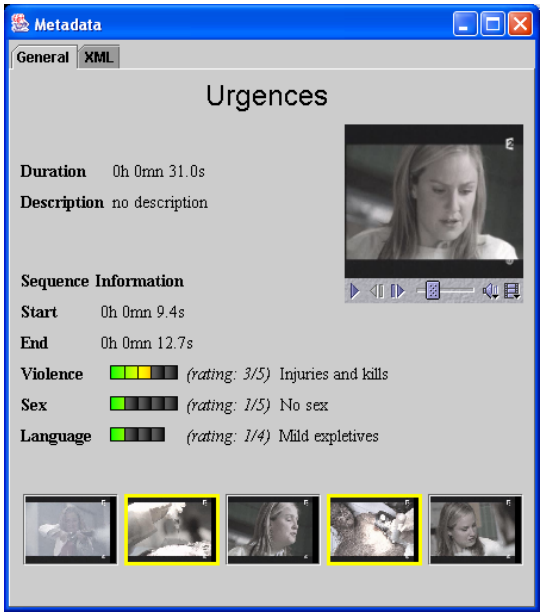


Figure 48. Sequence rating

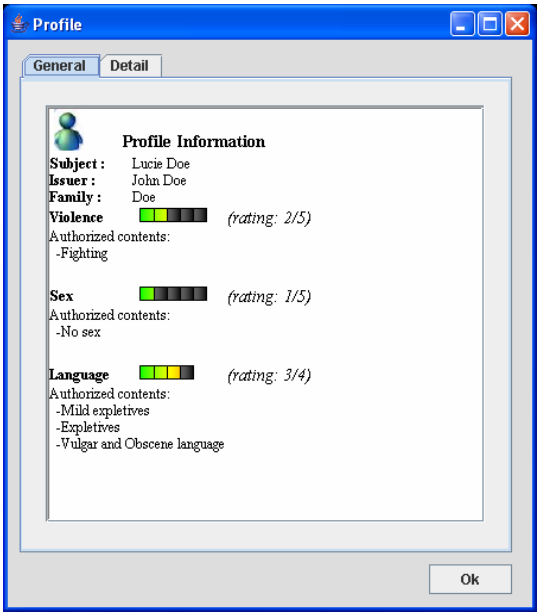


Figure 49. Child profile

6 Earlier project

When I started my PhD thesis at INRIA, I worked on the implementation of a prototype of C-SDA, a solution proposed by Bouganim and Pucheral [22][20]. This was a preliminary work of ours and it relied on an SOE to secure the access control for relational data stored on an insecure Database Service Provider (DSP). It has been demonstrated at the demo session of VLDB'03 [21].

6.1 C-SDA overview

C-SDA is a secure access controller for relational data. Access right in relational databases are based on the use of SQL views. A view is a virtual table built from other physical tables and computed thanks to an SQL statement. In this model, a user is granted access to a set of views and tables.

To accommodate the limited resources of the SOE, the query processing is delegated to the server and optionally to the terminal. The basic idea is to let the server executes part of the processing that can be done directly on encrypted data. The decryption and the remaining part of the processing are then performed in the SOE in a pipeline fashion to minimize the consumption of RAM. Finally, the terminal can sort the result (*Order by* clause).

The proposed solutions rely on a specific encryption scheme allowing delegating projections, equi-selections, equi-joins and grouping operations to the server. To this end, each column value is encrypted (e.g., using Triple DES) separately using an ECB mode. Therefore, the equality property is preserved among encrypted data. Bouganim and Pucheral [20] shows that, thanks to this property, any SQL query can be split in three parts. The first part (Q_S) is executed by the server and deal with equality predicates on the encrypted data. The second part (Q_C) is calculated in a pure pipeline fashion by the SOE from the data returned from the server, after decryption. Finally, the last part (Q_T) of the query for which the evaluation cannot hurt confidentiality, namely the result presentation (*Order by* clause) is computed by the terminal from the result returned by the SOE.

Optimizations based on a multi-stage cooperation between the SOE and the server have been proposed to minimize the amount of processed data and the transmission, decryption and CPU costs

The limitations of this solution are the following. First, by preserving the equality among the encrypted data, the encryption process provides a limited opacity. Thus, statistical analysis attacks can be done on the encrypted data. Second, this solution assumes no collusion between the client and the server. Indeed, to gain access to unauthorized data, a malicious client could try to modify the data on the server, even if encrypted, using substitution, random modification, etc. Thus, the system is not protected against altering, substituting and replay attacks.

This approach is however promising. The volume of storable data is unbounded, the solution provides data sharing and a reasonable level of performance and functionality can be obtained. New techniques have to be devised in order to ensure a higher security level while delegating part of the processing to a potentially untrusted server. These techniques are currently being investigated in our team by Anciaux et al. [5].

6.2 Demonstration

To show the effectiveness of C-SDA, we developed a full prototype in Java Card embedded in an e-gate smart card provided by Axalto. In order to provide a transparent access to the data, we implemented a JDBC driver on the terminal to give a transparent communication between the user application, the smart card and the server and hide the complexity of the underlying protocol (APDU).

To illustrate the internal of this solution, we developed a graphical interface which displays the data that are processed in each module of our architecture as shown in Figure 50. More precisely, we showed how the processing is delegated to the server and the terminal, and the communications which occur between the smart card, the terminal and the server.

We illustrated it on a business oriented shared database based on TCP-H and considered corporate users sharing some data with two different partners. More precisely, we showed in an example that a user can have access to a view computed from an aggregation (e.g., sum) of tuples from a physical table but not to the individual tuples.

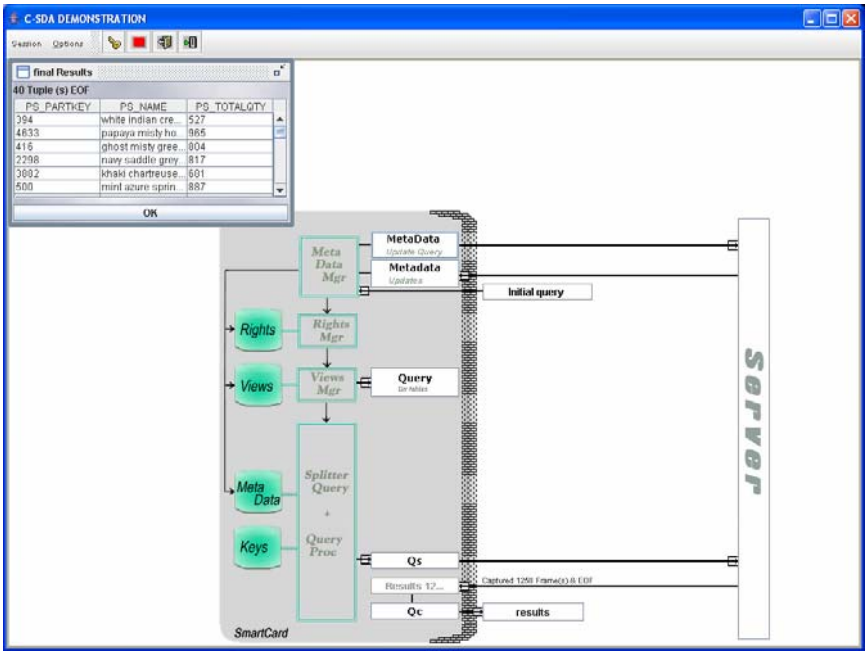


Figure 50. C-SDA demonstration

7 Conclusion

In this chapter, we assessed our solution on a hardware smart card simulator to have an exact prediction of its performance on advanced smart cards as announced by Axalto. The global throughput measured is around 70KBps and the relative cost of the access control is less than

20% of the total cost. These measurements are promising and demonstrate the applicability of the solution. Then, we showed the feasibility of our solution on current smart card which presented the functionality of the internals of our solution. Then, we showed that our approach can bring many benefits in various application context ranging from secure portable folders to DRM applications. Our work, rewarded by two international software contests demonstrates the growing interests of the mobile and DRM industries for secure client-based solutions. Finally, we described a previous preliminary work based on SOE to secure relational data.

Chapter 6 – Preliminary optimizations

1 Introduction

In chapter 4, we provided an efficient solution to enforce the access control on streaming XML data and demonstrated its effectiveness in chapter 5. Although our solution achieves good performance, we study in this chapter different cases which are unfavorable and then propose different solutions to improve our approach. These preliminary solutions have not been assessed yet and can serve as future prospects.

2 Case studies

From the experience conducted in the previous chapter, we found that our solution do not perform well for some sort of documents or queries. In this section, we characterize and explain the different factors which reduces the performances. In the following, we rely on Figure 51 to describe the different case studies.

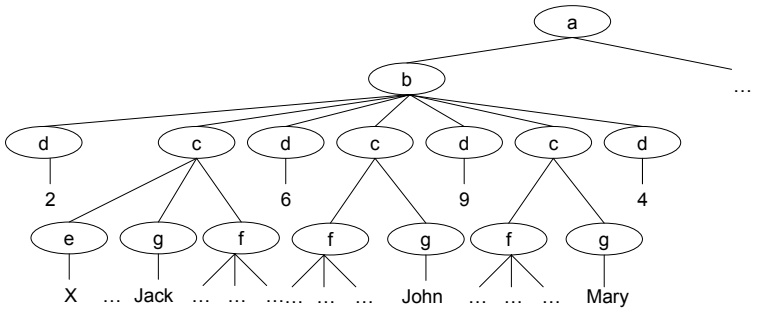


Figure 51. Case studies illustration

Case 1 – Cardinality of elements. In the current approach, we chose not to exploit the XML schema of the documents in order to have a generic evaluation. In this case, we do not have any information about the cardinality of elements. Therefore, if a user issues the query *//a/b/c/e*, then even after parsing the unique element *e* in the document, the parser will keep looking for other potential elements *e* among the siblings of element *e*. More generally, the parser will read all the sibling element metadata that are found along the path leading to the requested subtrees. The metadata (*Skip Index* encoding) are generally small but the overhead

may become not negligible if the XML document is wide (e.g., the document WSU is large and shallow so the execution of a selective query can have a high overhead).

Case 2 – Locality of elements. In our solution, the decryption is based on block cipher algorithms which operates on 64 (3DES) or 128 bit (AES) blocks. Most importantly, the integrity check is performed using the SHA-1 hash function operating on 64-byte blocks. Therefore, when a user accesses a single byte of data he will get along at least 64 extra bytes to check the integrity. Finally, an extra signature (24 bytes) is to be read to guarantee authenticity and integrity. Having these considerations in mind, it is easy to see that the sparser are the accesses, the bigger is the overhead.

Case 3 - Value-based queries. As we made the choice not to rely on any XML schema, the domain of values (which can be integers or strings) are not known and cannot be indexed. For instance, if we want to get all the subtree */a/b/c[g='Jack']/f*, all the subtrees */a/b/c[g]/f* are explored in the current solution. A partial knowledge of the schema (domain of values) may improve queries based on the selection of values.

3 Preliminary proposals

We provide in this section, some sketches of solutions to improve our approach based on the use of the XML schema of the documents. However, because of the drastic memory constraints of the SOE, it obviously cannot store the whole schema. In the following, we propose various techniques to integrate only relevant information of the schema on automata but also in the form of an index embedded in the data.

3.1 Exploiting the schema

Exploiting the knowledge of the schema can be done at different stages, that is: when defining access rules, when issuing the query and when parsing the document. We put aside the use of the knowledge of the schema at parsing time since it would mean that the schema would have to be stored on the SOE. For this reason, we investigate in the following different methods to integrate the schema in the access rule and query definitions.

The schema stores information about the cardinality of elements as well as referential constraints. We see in the following how they can be exploited.

3.1.1 Cardinality constraints

Cardinality of elements can be captured thanks to the XML schema which provides for each element, the possible number of occurrences of its child elements (e.g., element *a* has between 2 and 3 child elements *b*). Based on these information we can compute a list *CC* of 3-uple defined as <*SO*, *SI*, *count*> which tells that all the subtrees matching the XPath

expression *SI* can be found at most *count* times within the subtrees matching the XPath expression *SO*. For the sake of simplicity, we consider that *SI* is an XPath expression which is defined relatively to *SO* and that it does not exploit *//*; we will analyze how to handle // in future works. In the following, we refer to subtree *SI* or *SO*, subtrees matching *SI* or *SO*. For instance, we can devise for the document depicted in Figure 51 the following cardinality constraints: *CC={</a/b, c/e, 1>, </a, b, 10>}* which says, that the subtree */a/b* can have only one occurrence of element *c/e* and that element *a* can have at most 10 occurrences of element *b*.

The list *CC* can bring many benefits for the evaluation of the access rules. First, as soon as all the occurrences of a subtree *SI* have been found in a subtree *SO*, the evaluator knows that it is useless to evaluate the rules interested in a subpart of *SI* in the subtree *SO*. Second, this reduces the number of pending predicates. Indeed, the issue of a predicate may be known as soon as all the occurrences of the predicate elements are found. For instance, if we consider the rule *//b[c/e=Y]//g* and *CC = {<//b, c/e, 1>}*, then when the first element *c/e* is parsed (which is equal to X), the evaluator may know that no other element *c/e* can be found and thus concludes that the predicate is false. By doing so, a pending situation has been avoided. Because the list *CC* can be space consuming, it obviously cannot be stored into the SOE. To tackle this problem, the program in charge of encoding the access rules can be modified to integrate relevant information about the list *CC* into the access rules automata. When the owner defines a new rule, the program will append for some states of the automaton the following information:

- Depth *Depth* of the current subtree relatively to subtree *SO*.
- Maximum number of occurrences *NumOccurs* of the subtree *SI* in the subtree *SO*.

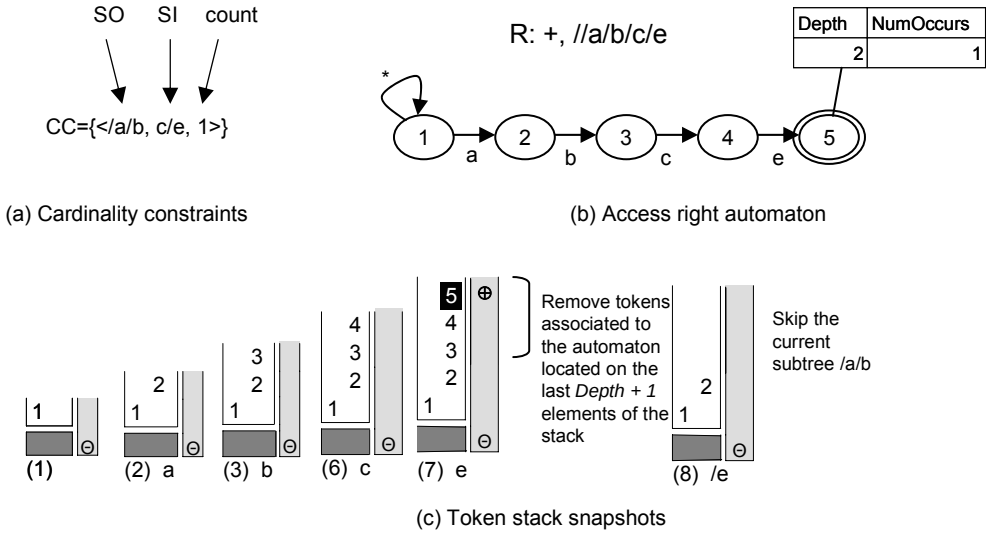


Figure 52. Exploiting the cardinality of elements

The intuition is as follows. When a token reaches a state corresponding to the matching of

an element which may occur *NumOccurs* times in *SO*, a counter (initially set to *NumOccurs*) is decremented. When the counter becomes null, it means that no other occurrences may occur in *SO*, thus all the tokens considering elements of the subtree *SO* can be discarded.

In the running example described in Figure 52, we consider only one cardinality constraint which tells that the subtree */a/b* can only have one instance of a subtree *c/e*. Based on this information, we mark the state 5 of the access right automaton representing *//a/b/c/e* with *NumOccurs=1* and *Depth=2*. Whenever this state is reached, the counter is decremented and reaches 0. Therefore, all the token proxy found in the subtree *SI* can be discarded since they cannot lead to a final state. Consequently, all the token proxies of rule *R* located in the last *(Depth + 1)* elements of the stack are removed, enabling thus to skip the whole subtree. Finally, the counter is reset to its initial value as soon as the outermost subtree */a/b* is left.

3.1.2 Referential constraints

In a schema, it is possible to define referential constraints thanks to primary keys. A primary key is an element value which identifies a subtree and makes it unique within an outermost subtree. These referential constraints can be defined by 3-uples defined as: *RC = {<SO, SI, SK>}* where the element matching *SK* (the key) identifies the subtree matching *SI*; the key value being unique within the outermost subtree *SO* (and therefore in *SI*). As in the case of cardinality constraints, we consider that *SI* is an XPath expression which is defined relatively to *SO*, that *SK* is defined relatively to *SI*, and that we do not exploit *//* in *SI* and *SK*; once again, we will analyze how to handle // in future works. For instance, the document pictured in Figure 51 can be subject to the referential constraint *RC = {</a/b, b/c, g>}* which tells that the element *g* value (e.g., Jack, John) identifies the subtree *b/c* within the subtree */a/b*.

The immediate benefit of primary keys is for the evaluation to skip more subtrees. Indeed, in the case of a rule with an equality predicate, once the subtree with the proper primary key satisfying the predicate has been found, there is no need to continue its evaluation on the outermost subtree in search for other matchings. Conversely, if the primary key value found in a subtree *SI* is not the requested one, then there is no need to evaluate the rule in this subtree since the key element is unique in *SI*. Once again, the program in charge of encoding the access rules can be modified to integrate relevant information about the list *RC* into the access rule automata. When the owner sets up a new rule, the final state of the predicate automaton operating on a key are augmented using the two following information:

- *dO* which is the depth of the primary key relatively to the outermost subtree *SO*.
- *dI* which is the depth of the primary key relatively to the innermost subtree *SI*.

The intuition is as follows. When a token reaches the final state of a predicate automaton corresponding to the matching of a primary key element, the value is examined and two cases may occur. First, it is the requested value, thus the current subtree is to be explored

but the others in the subtree *SO* can be skipped considering that the value is unique. In this case, all the tokens considering elements in *SO* but not in *SI* can be removed. Second, it is different from the requested value. In this case, the current subtree *SI* can be skipped and so tokens considering elements in *SI* can be removed.

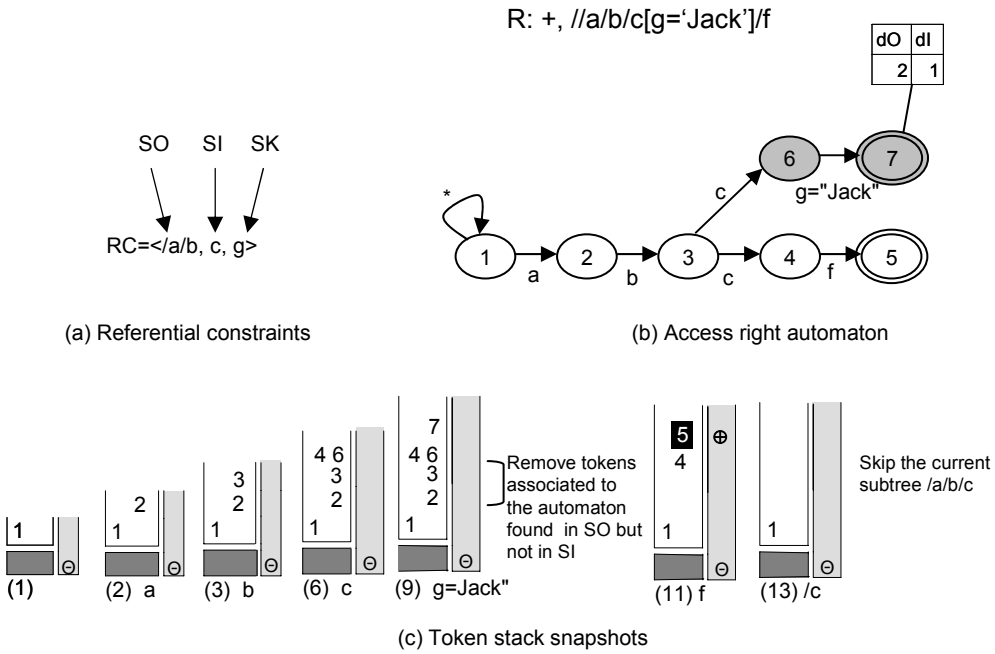


Figure 53. Exploiting referential constraints

We depict in Figure 53 a running example which exploits referential constraints. In the running example, we consider a single referential constraint which says that in the subtree */a/b*, only one subtree rooted by element *c* and identified by *g* can be found in the subtree. Based on this information, we mark the state 7 of the access right automaton representing *//a/b/c[g='Jack']/e* with *dO* and *dI*. Whenever this state is reached, the evaluator checks if the proper value is found (here, "Jack"). In the positive case, all the token proxies considering elements within *SO* but not in *SI* cannot lead to a final state and must be discarded. To this end, all the token proxies which are in the elements of the token stack corresponding to a depth in the range *[top – dO - 1, top – dI - 1]* are removed (step 9), where *top* denotes the current depth. Conversely, if the value was not the one requested, then the token proxies considering the evaluation of the subtree *SI* could not have lead to a final state and would have been discarded. To this end, all the token proxies located in the last *(dI + 1)* elements of the stack would have been removed.

3.1.3 Analysis

At first glance, when dealing with cardinality constraints, the overhead can be at most: *numStates * 2 * sizeOf(Integer)* since each state may be marked by a depth and a counter representing the number of occurrences. Note that each integer is likely to fit in a single byte

since they encode the depth and the number of occurrences which are small numbers. Also, it is more likely that only a single state of an automaton is to be affected. What can be done also, is to limit to one cardinality constraint per automata, considering the most relevant one. In the overall, only *numAutomata * 2* bytes are incurred by the cardinality.

Regarding referential constraints, they may apply only on predicate automaton final states. In this case, the overhead is equals to *2 * sizeOf(Integer).* Once again, the integer is likely to be small and fits in a single byte. Therefore, a predicate automaton will need *2* extra bytes.

If we consider that a rule automaton integrates one cardinality and one referential constraint, then the overhead is in overall equals to: *numAutomata * 4* bytes which is negligible.

These optimizations have the advantage to be easy to integrate in our current architecture, by adding a few modifications to the access rules evaluator and the program in charge of encoding access rule automata. In addition, they may also apply to the user query. The only difference here is that the query has to be pre-processed either by a third party (e.g., the terminal) having the knowledge of the schema of the requested document or the SOE itself. In the first case, this does not hurt the confidentiality of the data provided that the knowledge of the query is not confidential. In such situation, the only attacks that can be conducted by the third party is to provide an erroneous query automaton which may return a result with missing matches (e.g., the third party can tell that an element is unique while it is not). However, the user will not gain access to forbidden data. In the second case, the SOE can get in a streaming fashion the (encrypted) *CC* and *RC* and build the query automaton accordingly. The solutions described in this section solve also the problem raised by case 1.

3.2 Improving access locality

To improve access locality of accesses, two main solutions may be envisioned. First, storing XML data in relational databases. This allows, to take advantage of research conducted in this domain to group data which are likely to be accessed together. However, this solution does not accommodate streaming data. Indeed, the indexes are separated from the data (e.g., in B-Tree), meaning that there would be repeated accesses between indexes and data, thus making its use not compatible with streams. Second, multi-streaming data can be considered. The idea is to generate a separated stream for each different tags (e.g., Hospital, Folder, Diagnosis). For each of these streams, we consider a set of tuples defined as *(offset, size, value)*, where *offset* denotes the position of the element in the document, *size* is the size of the underlying subtree and *value* is the text node contained in the element if any. Based on this, the different streams are received at different speed and the evaluator basically performs containment comparison between the offsets and size of the different elements of different tags to process the query. However, this can become complex for documents having a large number of distinct tags; in this case there would be a large number of streams to consider. Moreover, this solution does not accommodate common streaming accesses such as broadcast and backward/forward access streams.

In the following, we sketch a solution embedding data and index, and compatible with a single stream of data.

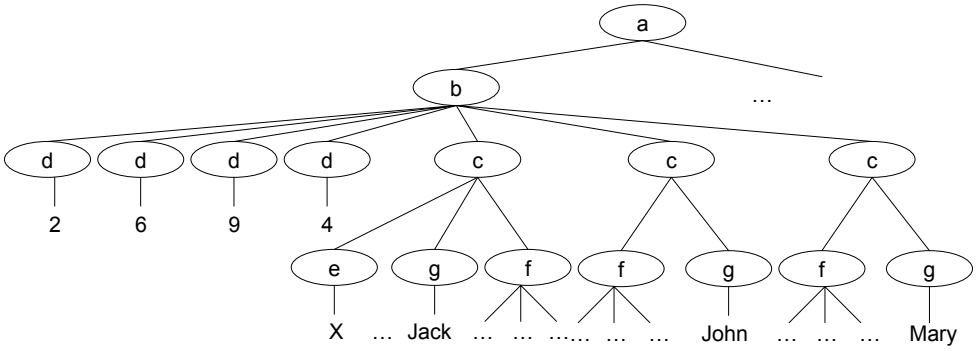


Figure 54. Grouping elements

Design rules

In this section, we propose different design rules (DR) to reduce the fragmentation of the read accesses on the encoded XML document and provide an encoding which integrates them all.

DR1 - Grouping elements of the same type This strategy aims at directing the evaluation towards the requested and relevant elements. For instance, to evaluate the query */a/b/c*, a good solution would be to access all the elements *a*, then all the elements *b* and finally all the elements *c* but not other elements such as *d*. By grouping the elements of the same type together as pictured in Figure 54, we can access all the relevant elements without having the fragmentation incurred by other irrelevant elements interleaved in the relevant ones. This obviously changes the global orders of the elements but not within a same group where the relative order is preserved (e.g., if we compare all the elements *d* their relative order remains the same). Therefore, for every node, we encode extra information which tells the global order *Gorder* of its child elements. For instance, for the element *b*, we encode the sequence of elements *d, c, d, c, d, c, d*; the orders between elements of the same type being preserved. This requires for the order an encoding of 2 bits (either *c* or *d*) per element. A more general formula is: $Log_2(distinct(tags_{children})) * numChildren$ bits. In this case, when evaluating the XPath */a/b/c*, all the elements *a* will be read, then all the elements *b* and then all the elements *c*. In this case, the traversal is made more efficient since no irrelevant elements are read (e.g., elements *d*). In some cases, the order is implicit (e.g., when there is only elements of the same type or when the elements of the same type are already grouped) or not important (e.g., data-centric applications) and needs not be encoded.

DR2 - Grouping metadata When navigating in the document, the decision to traverse an element depends on its metadata (more precisely on *TagArray* the descendant bit array). However, in the case of a selective query, only a few elements will actually need be explored. In this case, the fragmentation will be high considering that an element metadata is

always followed by its content, meaning that accesses will mostly consist of skipping element contents to go to the next metadata. We propose here to reduce this fragmentation by grouping all the metadata prior to their content. In this case all the metadata can be read in a contiguous block access.

DR3 - Factorizing the descendant bit array. We propose here to further compress the *Skip Index* considering that elements of the same type are likely to share the same structure and so the same descendant bit array. For each group of element *e*, we consider a factorized bit array *Obmp* which is computed as the ORed value of all the elements *e* descendant bit array. In this case, the descendant bit array of each element *e* can be encoded relatively to *Obmp*. Obviously, further optimization techniques can be devised for better factoring (e.g., consider groups of elements sharing the same descendant bit array and/or use a bit which tells if an element has descendant array equals to the factorized bit array) but are not discussed here and left for future prospects.

Putting them all together To integrate DR1, we consider for an element *e* that its child elements are grouped in different sets; each set considering elements having the same tag is identified by a *tagId*. The number of sets *#sets* is bounded by the number of descendant tag. For each of these set, we consider the *tag id, Obmp* (DR3) *and Gorder*. All these different sets are grouped together to allow a contiguous access on them.

Then we consider the metadata of elements having the same tag to implement DR2. These metadata contains only the list of descendant *bmp* which is a bit array computed relatively to *Obmp*. Once again, all these different sets are grouped together to allow a contiguous access on them.

Finally, we consider the contents of the elements (which contains in turn either text or other child elements). These contents are also grouped together.

In order to allow navigation between the different sets, we defined pointers encoded as offset which tells their relative positions. Note that in this model, the size of subtrees needs not be encoded since it can be obtained from the difference between the offset of an element and the element next to it. The offset can be compressed recursively using the same compression used in the current solution for the subtree size. This solution solves case 2 and also case 1.

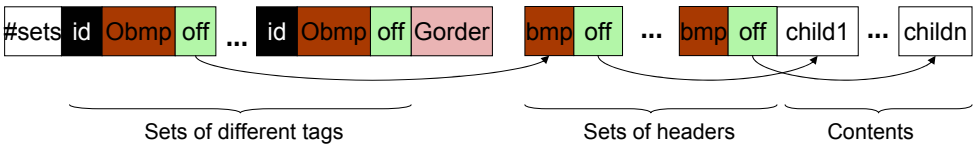
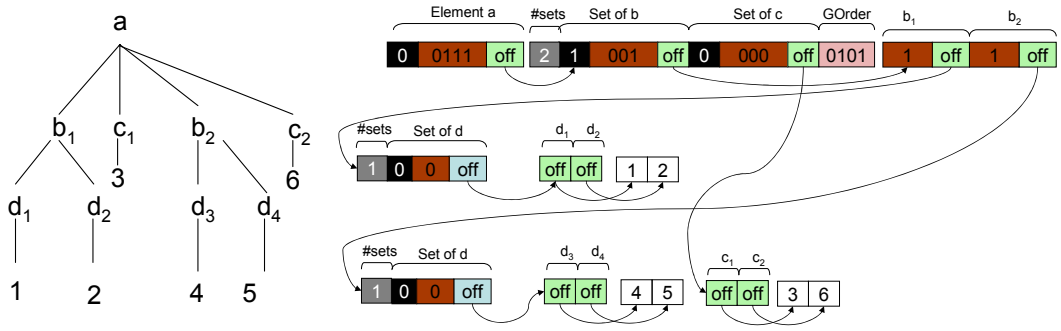


Figure 55. Node encoding



(a) XML Document (b) Encoded document

Figure 56. Example of encoding

We represent in Figure 55 the encoding of a node content and represent a whole encoding of a simple document in Figure 56. For the root element *a*, we encode first the list of tags present in the subtree. As, there is 4 tags in the document, we use 4 bits referring respectively to *a*, *b*, *c* and *d*. The associated bit array 0111 means that only 3 tags *b*, *c* and *d* can be found in the subtree. The subtree *a* is composed of a set of elements *b* and another of elements *c*. Their *tagId* computed from the tag array of *a* are respectively 0 and 1 and the order in which elements *b* and *c* appears are *b*, *c*, *b*, *c* which is described by the *GOrder* 0101. The two elements *b* contains both tags *d*, thus the *Obmp* of the set of elements *b* is 001 (no *b*, no *c*, *d*). As its two instances contain both elements *d*, their bit array *bmp* computed relatively to *Obmp* is equal to 1. Since there are only elements *d*, the *GOrder* field is useless and is not encoded. The element *d* contents are stored as is and their size is determined by the difference between offsets. The rest of the encoding are based on the same idea.

Analysis

Our solution is compliant with streaming data for the following reasons. First, the index which is embedded in the data always refer to elements which occurs afterwards. Second, information to be memorized in the SOE remains small. At first glance, we may think that we would need to buffer for every elements, the set of tags (i.e., tag id, factorized bitmap and offset) and the relative orders between siblings. However, only those related to ancestors of the current element (that is, the elements in the tag stack) need be memorized.

Regarding the space overhead incurred in this new model, we do not need to encode the *tagId* for each element but have to encode the orders between siblings. Moreover, the tag array is further compressed thanks to factoring. However, we have to consider additional offsets which can be compressed as it was the case for the subtree size. As a result, comparing the overhead with the current solution is not so easy and may depend on the nature of the document (shallow vs. deep, many vs. few distinct tags, big vs. small).

Regarding the execution time, we may expect a real improvement since this new encoding will reduce fragmentation and thus the cost ascribed to integrity checking. In our example, if a user issues the query *//c*, the evaluator can read them with 2 contiguous block accesses instead of 4 using the current *Skip Index*.

3.3 Indexing values

As addressed in case 3, indexing values can be used to detect more irrelevant subtrees wrt. access rules or a query interested in values. At the present, for the best of our knowledge, there is no work which consider indexing values on an XML stream.

A simple idea would be to split the document in blocks (for instance, the same partitioning as the one considered for the random integrity checking) and add extra information about the domain of values of the indexed elements with the benefit of skipping more subtrees. When getting to a subtree, the parser may know for a certain element (e.g., predicate) the domain of values it covers and may decide to skip it if these values are not relevant with a query or access rules.

If no care is taken, this information can be space consuming. Therefore, we append to each block, a bit array which gives either information about the numerical or textual values contained in the block. The purpose of this bit array is to make a first coarse filtering by the SOE so that when it receives the data block, it first decrypt and checks if the index is compliant with the value selection (either text or numeric). In the positive case, the current subtree is parsed and discarded otherwise.

For numerical value, each bit in the bit-array corresponds to an interval of values and are set to true if there is at least an indexed element whose value belongs to the interval in the block. For textual values, we use n-grams [62]. We consider a set of n-grams denoted by *N={n1, n2, …}* where *n1*, *n2* are strings of at most *n* characters (e.g., *n1* is “se” and *n2* is “ho”). Each bit of the bit-array corresponds then to an n-gram and is set to true if the n-gram is present in the block.

Let us illustrate how it works. If the user is interested in getting all the patients between 20 and 30 years old, the SOE will first look in the bit-array if the bit corresponding to intervals [20, 30] is set to true. In the same way, if a user is interested in retrieving all the patients with the name “Martin’, then the SOE will first look in the bit array if it contains for instance, the n-grams “ma” and “in”. In the positive case, the current subtree is explored and skipped otherwise.

Analysis

Regarding security, the set of n-gram and the definition of intervals can be made public since the bit array is stored encrypted embedded in the block. The computation of the bit array by the SOE can be either done by a third party (e.g., the terminal) or by the SOE itself.

In the first case, this does not hurt the confidentiality of data as long as the knowledge of the query is not confidential. Indeed, the only attack that can be conducted by the third party is to provide an erroneous bit array. In this case, false blocks will be considered but the evaluation still check the values, so in the global result, some parts may be missing. However, no forbidden information can be disclosed. In the latter case, the SOE can receive the set of n-gram and the definition of intervals in a streaming fashion and compute the proper bit array. This is secure since no information on the query is disclosed.

Regarding the space overhead, the size of the global index for an element depends on the precision (number of bits used) and on the number of blocks but not on the number of occurrences of the element.

This index can be further improved considering additional level of indexation (e.g., provide another index in the block which references subblocks). Moreover, instead of considering the block used for integrity checking, we may use another scheme and append the index only to some crucial nodes. This can once again serve as a basis for future research.

4 Conclusion

We sketched in this chapter different methods to improve the performance of our solution. First, by considering cardinality and referential constraints on the XML schema, we devised simple enhancements of the automata to allow skipping more irrelevant data. Second, by considering the locality of elements to avoid sparse access to data, we proposed an XML encoding to group data which are likely to be read together, therefore reducing the overhead incurs by integrity checking. Finally, we provided a sketch of solution to index values. In the future, we planned to investigate these preliminary solutions, to assess and to compare them with our existing solution.

Chapter 7 – Conclusion and research perspectives

1 Summary

Nowadays, many important factors motivate to move the access control from servers to client devices. However, by compiling the access control policies into the data encryption, existing client-based access control solutions minimize the trust required on the client at the price of a rather static way of sharing data.

Unlike existing solutions, we separated the access control policies from encryption by taking advantage of *Secure Operating Environments.* This clear distinction allows to provide a dynamic and personalized access control on a ciphered XML document. While SOE provide a high tamper resistance, they have drastic constraints: few memory, low throughput and relatively low cryptographic processing.

We introduced in chapter 2, the different technologies which can be used to build our solution. First, we presented XML and the XPath language used to define access control policies. More precisely, we described solutions to evaluate a single XPath query on streaming XML data. While they offer interesting feature, they cannot be used in our context since access rules are not independent and may generate conflicts. Second, we presented different cryptographic techniques to enforce the confidentiality of data. Third, we provided an overview of Secure Operating Environments. Finally, we provided a bried overview of different access control models.

In Chapter 3, we provided a state of the art on XML access control. First, we described the basic XML access control model that is considered in this thesis and gave an overview of the different variations of this model. Then we presented different methods to evaluate the access control on XML documents using DOM labeling and query rewriting in a secure server. Then we described more in details existing client-based solutions which consider untrusted servers. These solutions compile access rights in the encryption and are concerned by the distribution of keys. While they provide interesting techniques to tackle this issue, the encryption scheme still considers a large number of keys (or a combination of keys), and thus failed in providing a dynamic and personalized access control rights.

In order to alleviate these limitations, we proposed in Chapter 4 to make a clear separation between the encryption scheme and access control. To this end, we took advantage of new

elements of trust (SOE) in client devices to propose a client-based access control manager capable of evaluating securely dynamic and personalized access rules on a ciphered XML document. In order to accommodate the limited RAM capacity of the SOE, we proposed a streaming evaluator of access control rules supporting a rather robust fragment of the XPath language, enabling thus to provide fine-grained access. To the best of our knowledge, this is the first approach dealing with XML access control in a streaming fashion. Then, to accommodate the low throughput and expensive cryptographic functions, we designed the *Skip Index* structure embedded into the data that gives to the evaluator extra information on the structure of the document and allows to skip data irrelevant with a query as well as forbidden data. While streaming data is beneficial to minimize the consumption of RAM, it may generate pending situations. A pending situation occurs when the delivery of a subpart of the document is conditioned by a predicate depending on some data which appears later in the stream. To tackle this situation, we proposed two graceful methods depending on the nature of the stream (strict streaming and backward/forward access streaming). Finally, we provided security techniques to guarantee the confidentiality of data. To this end, we designed a secure encryption scheme and integrity mechanism compliant with the backward/forward accesses generated by the *Skip Index*. Moreover, we proposed a secure mechanism to refresh the access rights. This mechanism resists against replay attacks aiming at creating an inconsistency between access rights and data to gain unauthorized access to data.

In Chapter 5, we have validated these contributions. First, because current smart cards are too limited (especially in terms of bandwidth), we measured the performance obtained from a C prototype running on a hardware cycle-accurate smart card simulator provided by Axalto to predict the performances on advanced smart cards. From these measures, we concluded that the *Skip Index* space overhead is negligible, that the access control evaluation time overhead is small compared to the communication and decryption overhead, that the integration of the query in the evaluation is beneficial, that the integrity checking has a reasonable overhead and finally that our solution accommodates well different kind of data taken from real datasets. The global throughput measured is around 70KBps and the relative cost of the access control is less than 20% of the total cost. These measurements are promising and demonstrate the applicability of the solution. Second, we showed the feasibility of our approach on current smart cards (Java Cards) even though they do not achieve good performance. Finally, we integrated our Java Card prototype in three different application contexts to show the versatility of the solution from the application's viewpoint. In these contexts, we showed that our approach can bring many benefits in terms of security and also in terms of business models. These experimental works were awarded by two international software contests.

In Chapter 6, we discussed different optimizations. We identified some cases where the *Skip Index* can be improved. By taking into account some elements of the schema of the document, we augmented the access rules to devise further optimizations in order to skip

more irrelevant data. Moreover, we proposed a different encoding scheme to improve the locality of the read access, thus reducing the overhead incurred by the integrity checking and hence the communication overhead. Finally, we provided a way to index values on the XML documents. We plan to assess these optimizations in future works.

2 Research perspectives

In this thesis, we expect demonstrating that client-based security solutions deserve a special attention for the new research perspectives they broaden and for their foreseeable impact on a growing scale of applications. We describe below different research perspectives that can be drawn:

- *Extend our solution to take into account new variations of XML access control models*
  New access control models [36][54][84] are now evolving towards the use of security views which consider restructuring the original document to prevent inference-based attacks. In these models, an element may be moved to another place in the result document, more specifically: (1) its sibling order may be changed, (2) it can be attached to one of its ancestor (path-reduction), (3) one of its ancestors may be anonymized. In this case, we would have to devise new buffering techniques to make their evaluation in a streaming fashion compliant with the constraints of the SOE. When delivering elements in the result, elements would have to come with extra information to be integrated properly in the final result (in a way similar to Pool Encryption [60]), yet without disclosing any information that could be used in inference attacks. In this case, when an element is delivered, its position in the result can be made fuzzy, omitting the sibling order and/or anonymizing or omitting ancestor information to integrate properly the element in the result. Another possible extension is to consider provisional accesses [72]. In this case we would have to augment the rule definitions so that a user is granted access provided he fulfilled a condition. This condition may be an emergency case, an obligation to provide the purpose for the access which is made available to the owner of data, a ticket provided by another user, etc. For instance, in the case of emergency, a nurse may bypass the access control to have a full access to a patient's medical folder provided that all her accesses are logged permanently in her SOE.
- *Consider new generations of SOE*
  SSMSC (Smart Secure Mass Storage Cards) is a new secure component which combines in the same hardware platform a secure chip and a Flash mass storage memory with a capacity of gigabytes. From a technical point of view, it is similar to our context in which a SOE has access to a large unprotected amount of data on a remote untrusted server. In the case of SSMSC, the access from the secure chip to the data is fast since they are on the same physical support. However, the Flash technology has a complex management of writing access (incremental writing, block erasure prior to updates). In this model, as the

read accesses are fast, there is no need to have the streaming requirement. Therefore, we can design a new access control evaluation which can takes advantage of indexes to have direct access to the data. Obviously, we would have to check both the integrity of the index and the data. In this model, updating data is very costly, but adding new data has a reasonable cost. That precludes the use of normal indexing methods such as B-Tree which need be balanced and thus updated. These paradigms are quite similar to problems addressed in WORM (Write Once Read Many) storage but provide more flexibility. We can investigate new techniques based on incremental indexing, based on non-balanced trees and hash indexing to accommodate the constraints on the write accesses.

- *Delegating processing*
  In our solution, the server has a limited role in the processing of data and consists only in sending segments of data. The role of the server was reduced to the minimum in order to keep the encryption scheme opaque. As explained in Chapter 5, in the C-SDA approach the server has an active role since it evaluates parts of the query on encrypted data. These bring two major issues. First, using a particular encryption scheme or indexes on encrypted data may reveal some information. When giving the server possibilities to make some processing, it implicitly gives the pirate the possibility to do the same and to exploit them to get unauthorized information. Second, as the server is untrusted, it means that it can return incorrect answers which have then to be verified by the SOE. Some techniques exist (e.g., Merkle Hash Tree [78]) but they have a non-negligible overhead. Therefore, we can investigate further in this direction to propose new methods to reduce this overhead. Moreover, an alternative would be to use a secure component at the server-side. While the secure component may simplify to some extents the two problems aforementioned (in the extreme case, it evaluate the query as in the current solution and sends the result to the client), it is likely to become a bottleneck on the server since it would have to answer to many queries from different clients. Therefore, we would have to provide new techniques to reduce the amount of data to be read and processed by the secure component. To this end, we may investigate new compact indexes (not necessarily streaming indexes) that can be analyzed quickly by the SOE to tell the server the supersets of data to be returned and refined by the client.

Résumé en Français – French Summary

1 Introduction

La gestion du contrôle d'accès, un des éléments fondamentaux des systèmes de bases de données, est traditionnellement effectuée sur les serveurs, à qui les utilisateurs accordent leur confiance. Cependant, cette situation évolue rapidement à cause de plusieurs facteurs : la méfiance vis à vis des hébergeurs de données ou *Database Service Providers* (DSP) concernant la préservation de la confidentialité des données [20][62], la vulnérabilité croissante des serveurs de bases de données face aux attaques externes et internes [35], l'émergence de moyens décentralisés pour partager les données grâce aux bases de données pair à pair [86] et aux systèmes de distribution basé sur des licences [127], et l'inquiétude croissante des parents et des enseignants vis à vis des contenus digitaux téléchargés sur Internet par des mineurs [120].

La conséquence commune de ces facteurs orthogonaux est la nécessité de faire migrer le contrôle d'accès des serveurs vers les clients. Du fait de l'absence intrinsèque de sécurité des terminaux clients, toutes les solutions de gestion du contrôle d'accès sur le client reposent sur le chiffrement de données. Les données sont stockées chiffrées sur le serveur et un client ne peut accéder qu'aux parties pour lesquelles il possède la clé de déchiffrement. Plusieurs améliorations de ce modèle ont été proposées dans des contextes très variés comme l'hébergement de données [62], la sécurité des serveurs de bases de données [63], la publication de données lucrative et non-lucrative [18][80][81] et les bases de données multi-niveaux [3][100]. Ces modèles diffèrent sur plusieurs points : le modèle d'accès aux données (interrogation vs. diffusion), le modèle de droit d'accès (DAC, RBAC, MAC), le schéma de chiffrement, le mécanisme de distribution des clés et la granularité de partage. Cependant, ces modèles ont en commun de minimiser la confiance requise sur le terminal client au prix d'un partage statique des données. En effet, quelle que soit la granularité du partage considéré, l'ensemble des données est découpé en sous-parties qui suivent le schéma de partage et chacune d'elle est chiffrée avec une clé différente ou une composition de clés différentes. Ainsi les intersections des règles de contrôle d'accès sont précompilées par le chiffrement. Une fois l'ensemble des données chiffré, toute modification de la définition des règles de contrôle d'accès peut entraîner une modification de la frontière entre les différentes sous-parties et aboutir à un rechiffrement partiel de l'ensemble des données et éventuellement à une redistribution des clés.

Cependant, il y a de nombreuses situations où les règles de contrôle d'accès sont spécifiques à chaque utilisateur, dynamiques et donc difficile à prédire. Considérons par exemple une communauté d'utilisateurs (famille, amis, équipe de recherche) partageant des données via un DSP ou d'une manière pair à pair (agendas, carnets d'adresses, profils d'utilisateurs, travaux de recherche, etc.). Il est très probable que les politiques de partage changeront au fur et à mesure que la situation initiale évolue (changement des relations entre les utilisateurs, nouveaux partenaires, nouveaux projets avec des intérêts divergents, etc.). Traditionnellement, l'échange d'information médicale est dicté par des politiques strictes de partage pour protéger la vie privée des patients mais ces règles peuvent subir des exceptions dans des situations particulières (e.g., en cas d'urgence) [48], peuvent évoluer avec le temps (e.g., en fonction du traitement en cours du patient) et peuvent être sujet à des autorisations provisionnelles [72]. De la même manière, rien ne justifie qu'une base de données hébergée ait des règles de contrôle d'accès plus statiques qu'une base de données gérées localement [20]. En ce qui concerne le contrôle parental, ni les gestionnaires de sites Web ni les fournisseurs d'accès Internet ne peuvent prédire la diversité des règles de contrôle d'accès que les parents, avec leurs sensibilités différentes, veulent voir appliquer à leurs enfants. Finalement, la diversité des modèles de publication (lucratif et non-lucratif) amène à définir des langages de contrôle d'accès complexes comme XrML, XACML ou ODRL [91][127][93]. Les règles de contrôle d'accès étant plus complexes, le contenu chiffré et les licences sont gérés par des canaux différents, permettant à différents utilisateurs de jouir de différents privilèges sur le même contenu chiffré.

Parallèlement, les architectures matérielles et logicielles évoluent rapidement et intègrent des éléments de confiance dans les terminaux clients. Windows Media 9 [80] est un exemple de solution logicielle qui sécurise du contenu numérique publié sur le PC et d'autres terminaux électroniques. Les tokens sécurisés et les cartes à puce reliés ou intégrés dans divers appareils (e.g., PC, PDA, téléphone portable, set-top-box) sont des solutions matérielles utilisées dans un nombre croissant d'applications (certification, authentification, vote électronique, paiement électronique, santé, gestion de droits digitaux, etc.). Finalement, TCG [112] est une solution hybride où une puce sécurisée est utilisée pour certifier le logiciel installé sur une plate-forme donnée, l'empêchant ainsi d'être piraté[8]. Ainsi, les *Secure Operating Environments* (SOE) deviennent aujourd'hui une réalité sur les terminaux clients [115]. Les SOE matérielles garantissent une grande résistance aux attaques, généralement sur des ressources limitées (e.g., une petite portion de mémoire stable et de RAM sont protégées pour garder secret des données comme les clés de chiffrement et des structures de données sensibles).

L'objectif de cette thèse est d'exploiter ces nouveaux éléments de confiance pour établir de

[8] Les architectures comme TCG font aujourd'hui l'objet d'une controverse. Notre objectif n'est pas d'alimenter ce débat. Cependant on peut constater que les architectures basées sur un client sécurisé se développent de plus en plus et les considérer pour concevoir des nouveaux modèles de sécurité, de nouveaux moyens pour protéger la confidentialité et la privacité des données est indéniablement un challenge important. Le vrai danger serait de laisser un seul acteur ou consortium décider d'un modèle unique de sécurité qui s'imposerait à tous.

meilleures solutions de gestion du contrôle d'accès sur le client. Le but poursuivi est d'évaluer des règles de contrôle d'accès personnalisées et dynamiques sur un document chiffré passé en entrée, avec comme bénéfice de dissocier les droits d'accès du schéma de chiffrement. Les documents considérés sont des documents XML, le standard de fait pour l'échange de données. Les modèles d'autorisation proposés pour réguler l'accès aux documents XML utilisent des expressions XPath pour définir la portée de chacune des règles de contrôle d'accès [18][39][58]. Dans ce contexte, nous adressons dans cette thèse le problème de la manière suivante :

- *Proposer une évaluation efficace de règles de contrôle d'accès compatible avec les contraintes matérielles du SOE*
  Le SOE étant le seul élément sûr, l'évaluateur de contrôle d'accès de règles doit être embarqué dans le SOE. Premièrement, la faible quantité de mémoire sécurisé du SOE exclue les techniques basées sur la matérialisation (e.g., construire une représentation DOM [118] du document). Deuxièmement, la puissance limitée du processeur et de la bande passante de communication nous amène à minimiser le volume de données reçu et déchiffré par le SOE. L'efficacité est quant à elle une préoccupation majeure et récurrente.
- *Garantir que les informations non autorisées ne sont jamais révélées*
  Le contrôle d'accès étant réalisé sur le terminal client, seules les parties autorisées doivent être rendues accessibles aux éléments non sûrs du terminal client.
- *Protéger le document d'entrée contre toute forme de modification illicite*
  Sous l'hypothèse que le SOE est sûr, la seule manière de tromper l'évaluateur de contrôle d'accès est de modifier illégalement le document d'entrée, par exemple en substituant ou en modifiant des blocs chiffrés de ce document.

Contributions

Pour résoudre ce problème, cette thèse apporte les contributions suivantes :

1. *Evaluation en flux des règles de contrôle d'accès*
   Nous proposons un évaluateur en flux de règles de contrôle d'accès XML supportant un sous-ensemble conséquent du langage XPath. A première vue, évaluer un ensemble de règles de contrôle d'accès basé sur XPath et évaluer des requêtes XPath sur un document arrivant en flux semblent être des problèmes équivalents [43][30][61]. Cependant, les règles de contrôle d'accès ne sont pas indépendantes entre elles et peuvent générer des conflits ou devenir redondantes sur certaines parties du document. L'évaluateur proposé détecte de manière précise ces situations et en tire profit pour stopper activement les règles qui deviennent non pertinentes.
2. *Index de Saut*
   Nous proposons une structure compacte d'indexation pour les données en flux qui permet (i) de converger rapidement vers les parties autorisées du document d'entrée en sautant

les parties non autorisées, et (ii) de calculer les intersections avec une requête qui pourrait s'appliquer sur le document (dans un contexte pull). Indexer est particulièrement important compte tenu du fait que les deux facteurs limitants de l'architecture cible sont le coût du déchiffrement dans le SOE et le coût de communication entre le SOE, le client et le serveur. Cette seconde contribution est complémentaire de la première pour atteindre l'objectif de performance.

3. *Gestion des prédicats*
   Les prédicats en attente (i.e., un prédicat *P* qui conditionne la délivrance d'un sous-arbre *S* mais qui est rencontré après *S* lors de l'analyse du document) sont difficiles à gérer en flux. Nous proposons une stratégie permettant de détecter les parties en attente du document, de les sauter lors du parsing puis de réassembler au bon endroit celles qui sont pertinentes vis à vis du résultat final. La manière dont les prédicats en attente sont gérés garantit que les données non autorisées ne sont jamais révélées sur le terminal client.

4. *Gestion des droits d'accès*
   La dynamicité des politiques de contrôles d'accès impose au SOE de rafraichir la définition des droits d'accès à partir d'un serveur non-sécurisé. Nous proposons de protéger ce mécanisme de rafraichissement contre des attaques de rejeu conduit par un hacker pour gagner accès à des parties interdites.

5. *Vérification de l'intégrité sur des accès aléatoires*
   Nous combinons des techniques de hachage (arbre de hachage de Merkle [78]) et de chiffrement (*Cipher Block Chaining* ou *CBC* [77][103]) pour vérifier l'intégrité du document en flux, malgré les accès aléatoires en avant et en arrière générés par l'utilisation de l'*Index de Saut* et par la gestion des prédicats en attente.

Cette thèse est organisée comme suit. La Section 2 donne un aperçu global des approches existantes pour sécuriser le contrôle d'accès sur des documents XML. La Section 3 introduit le modèle de contrôle d'accès XML considéré ici et l'illustre sur un exemple motivant l'intérêt de l'approche. Les Sections de 4 à 8 détaillent les contributions mentionnées ci-dessus. La Section 9 présente les résultats expérimentaux basés sur des jeux de données synthétiques et réels. La Section 10 reporte deux expériences effectuées avec un prototype de notre approche développé sur une plate-forme de carte à puce agissant comme un SOE. La Section 11 conclut.

2 Travaux connexes

Malgré l'intérêt croissant pour le chiffrement de documents XML d'un côté [32][121] et le contrôle d'accès XML de l'autre [1][18][29][39][53][54][57][58][72][81][127], peu de travaux combinent réellement ces deux aspects. Comme mentionné dans l'introduction, il y a pour l'instant un nombre important de situations où le contrôle d'accès doit être effectué sur des terminaux clients et le chiffrement est vu comme un prérequis dans ce contexte. Cette

section décrit différentes approches pour traduire des politiques de contrôle d'accès en un schéma de chiffrement et met l'accent sur leurs limites par rapport au problème addressé dans cette thèse. Des éléments plus spécifiques de travaux connexes sont référencés dans chaque section de cette thèse.

Chiffrement direct Le chiffrement direct se réfère aux méthodes qui traduisent une politique de contrôle d'accès en un ensemble de fragments XML et de clés de chiffrement tel que : (i) une partition du document est définie selon l'ensemble des autorisations (i.e., règles de contrôle d'accès positive et négative) formant la politique, (ii) chaque fragment résultant de cette partition est chiffrée avec une clé différente, (iii) chaque sujet reçoit les clés nécessaires pour déchiffrer les fragments pour lesquels il est autorisé. Le framework Author-X [18] est représentatif de cette approche. Il considère un modèle de publication/souscription où les données sont émises aux souscripteurs. Le schéma de chiffrement suit un chiffrement élément par élément (i.e., les tags, attributs et valeurs sont chiffrés à leur place dans le document avec une granularité d'un élément). Les clés de déchiffrement peuvent être fournies aux souscripteurs de différentes manières (e.g., par un répertoire LDAP ou à l'intérieur du document lui-même, chiffrées avec la clé publique de chaque souscripteur). Le facteur limitant dans cette approche est le nombre de clés à considérer puisque ce nombre peut croitre exponentiellement avec le nombre d'utilisateurs. Ce problème peut être résolu en utilisant des clés compatibles [99] qui fournit une manière de déchiffrer des données chiffrées en utilisant des clés différentes. Cependant, les clés compatibles s'appuient sur un chiffrement asymmétrique coûteux, qui est environ trois fois plus lent que du chiffrement symmétrique.

Chiffrement imbriqué Miklau et Suciu [1][81] proposent un autre schéma de chiffrement basé sur du chiffrement imbriqué (i.e., chiffrement récursif d'une même donnée avec des clés différentes). Des clés internes sont utilisées pour chiffrer des sous-parties du document et sont elles-mêmes intégrées dans le document. Les clés internes sont chiffrées avec des clés des utilisateurs ou des informations provisionnelles (e.g., date de naissance, numéro de sécurité sociale) et peuvent être combinées ensemble (e.g., en utilisant l'opérateur XOR) pour former une nouvelle clé correspondante à une expression logique potentiellement complexe. Par ce moyen, les conditions logiques pour accéder aux données peuvent être directement compilées dans le processus de chiffrement. Quand un utilisateur reçoit un document, il déchiffre les sous-parties auxquels il a été initialement autorisé puis continue le déchiffrement des sous-parties récursivement tant qu'il possède les clés de déchiffrement appropriées.

Cette solution fournit une solution élégante pour implémenter des accès basés sur conditions logiques complexes et provisionnelles en reposant sur une distribution simple de clés. Cependant, il souffre de limitations importantes dans notre contexte. Premièrement, le coût occasionné par le chiffrement imbriqué et par l'initialisation des fonctions cryptographiques pour les clés internes rend cette solution innapropriée pour les terminaux ayant des capacités

de traitement limités. Deuxièmement, comme aucune compression n'est considérée, le surcoût en terme de place induit par le format XML encryption et les clés internes peuvent être importants.

Chiffrement supportant les requêtes Les solutions précédentes ne sont pas efficaces à partir du moment où un utilisateur est intéressé par (ou autorisé à accéder) un petit sous-ensemble du document. En effet, il n'y a pas de structure d'indexation pour converger vers les parties pertinentes du document par rapport à une requête potentielle et/ou des règles de contrôle d'accès. L'idée développée par Carminati et al. [29] est de déléguer une partie de l'évaluation de la requête au serveur non-sûre hébergeant les données chiffrées. Pour cela, le chiffrement élément par élément est utilisé. Une requête sur la structure XML peut être exécutée relativement facilement par le serveur, en chiffrant en place les tags et les attributs de l'expression XPath (e.g., */a/b* peut être évalué sur des données chiffrées comme étant */E(a)/E(b)* où *E* dénote la fonction de chiffrement). Les sélections sur les valeurs sont résolues en utilisant une partition d'index de manière similaire à Hacigumus et al. [62], en associant à chaque valeur chiffrée, une valeur d'index en claire qui indique l'intervalle de valeurs à laquelle la valeur appartient (les limites des intervalles restent cachés du serveur). Cela permet un filtrage gros-grain sur le serveur qui est ensuite raffiné par le client après déchiffrement.

Déléguer le traitement sur un serveur non sûr nécessite de vérifier l'intégrité du résultat ; le problème le plus difficile à traiter étant celui de la complétude du résultat. Ce problème est résolu par Devanbu et al. [42] à l'aide d'arbre de hachage de Merkle [78] créé sur le document. Cependant Devanbu et al. ne considèrent le problème de complétude que sur des sous-arbres complets, excluant ainsi l'utilisation de règles négatives d'autorisations dans les politiques de contrôle d'accès. Carminati et al. ont étendu l'arbre de hachage de Merkle vers un arbre de hachage XML en considérant pour chaque nœud interne XML, une valeur de hachage calculée à partir de son tag, de son contenu et du hachage des ses nœuds fils.

Dans notre contexte à ressource limitée, cette solution souffre de deux faiblesses. Premièrement, le format d'encodage peut induire un surcoût important en terme d'espace, sachant que les tags et les valeurs doivent être paddés au moment du chiffrement (e.g., 3DES et AES produisent respectivement des blocs de 64 et 128 bits). Les index et les informations sur le schéma contribuent aussi à ce surcoût d'espace. Deuxièmement, l'extension de l'arbre de hachage de Merkle qui à l'origine est prévu pour opérer sur des arbres binaires induit un surcout important : quand un élément possédant *n* éléments frères est retourné par le serveur, les *n* hachages des éléments frères doivent aussi être transmis avec la réponse (SHA-1 produit des hachages de 20 octets).

Comme conclusion, bien que ces méthodes offrent des caractéristiques intéressantes, elles sont mal adaptées dans le contexte considéré dans cette thèse. Les faiblesses communes de ces approches sont relatives à la gestion des droits d'accès dynamiques. En effet, les sous-parties du document doivent être rechiffrées à chaque fois que les politiques de droits

d'accès changent.

3 Modèle de contrôle d'accès

Sémantique du modèle de contrôle d'accès

Plusieurs modèles d'autorisation ont été récemment proposés pour réguler l'accès à des documents XML. La plupart de ces modèles suivent le modèle discrétionnaire (DAC) [18] [39][58], même si les modèles RBAC et MAC ont aussi été considérés [31][34]. Nous introduisons ci-dessous un modèle simplifié de contrôle d'accès pour XML, inspiré du modèle de Bertino et al. [18] et de celui de Damiani et al. [39] qui partagent globalement les mêmes principes. Les subtilités de ces modèles sont ignorées pour des raisons de simplicité.

Dans ce modèle simplifié, les règles de contrôle d'accès (ou *règles d'accès*), prennent la forme d'un triplet <*signe*, *sujet*, *objet*>. *Signe* désigne soit une permission (règle positive), soit une interdiction (règle négative) pour l'opération de lecture. *Sujet* représente le destinataire de la règle. *Objet* correspond aux éléments ou sous-arbres du document XML, identifiés par une expression XPath. La puissance d'expression du modèle de contrôle d'accès, et donc la granularité du partage, est directement liée au sous-ensemble de XPath supporté.

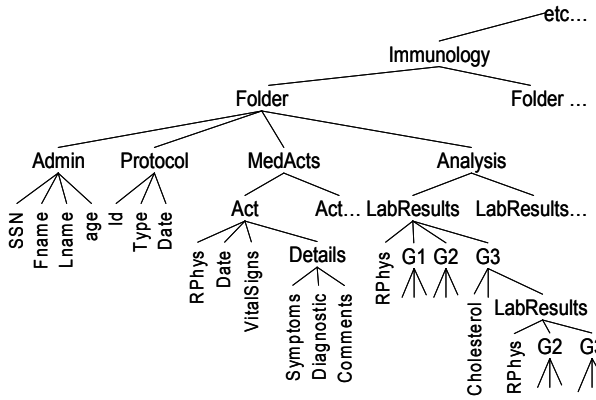
Dans cette thèse, nous considérons un sous-ensemble significatif de XPath désigné par $XP^{\{[],*,//\}}$ [83]. Ce sous-ensemble, largement utilisé en pratique, englobe l'utilisation de tests sur les nœuds, de l'axe parent-enfant (*/*), de l'axe de descendance (*//*), de jokers (*) et de prédicats *[…]*. Les attributs sont gérés dans ce modèle de manière similaire aux éléments et ne seront donc pas considérés explicitement dans la suite.

La propagation en cascade des règles est implicite dans ce modèle, ce qui signifie qu'une règle se propage d'un objet à tous ses descendants dans la hiérarchie XML. Etant donné ce mécanisme de propagation et le fait que plusieurs règles peuvent être définies pour un même utilisateur sur un même document, un principe de résolution de conflit est nécessaire. Les conflits sont résolus en utilisant deux politiques : *L'Interdiction-Est-Prioritaire* et *Le-Plus-Spécifique-Est-Prioritaire*. Considérons deux règles *R1* et *R2* de signe opposé. Ces règles peuvent être en conflit soit parce qu'elles sont définies sur le même objet, soit parce qu'elles sont définies respectivement sur deux objets différents *O1* et *O2*, reliés par une relation ancêtre-descendant (i.e., *O1* est l'ancêtre de *O2*). Dans le premier cas, la politique *L'Interdiction-Est-Prioritaire* donne priorité à la règle négative. Dans le second cas, la politique *Le-Plus-Spécifique-Est-Prioritaire* donne priorité à la règle qui s'applique directement sur l'objet par rapport à celle qui est héritée (i.e., *R2* est prioritaire sur *R1* pour l'objet *O2*). Finalement, un sujet se voyant accorder l'accès à un objet obtient aussi l'accès au chemin reliant la racine du document à cet objet (les noms des éléments interdits sur le chemin peuvent être remplacés par des valeurs factices). Cette règle *Structurelle* conserve ainsi la structure du document qui reste cohérente avec l'original.

L'ensemble des règles associé à un sujet et s'appliquant sur un document donné est appelé une *politique de contrôle d'accès*. Ces politiques définissent une vue autorisée de ce document, qui, suivant le contexte applicatif, peut être interrogée. Nous considérons que les requêtes sont exprimées avec le même sous-ensemble XPath que celui considéré pour les règles d'accès, c'est à dire $XP^{\{[],*,//\}}$ [83] Du point de vue sémantique, le résultat d'une requête est calculé à partir de la vue autorisée du document considéré (e.g., les prédicats ne peuvent s'appliquer sur des éléments non-autorisés même si ceux-ci n'apparaissent pas dans le résultat de la requête). Cependant, les prédicats de règles d'accès peuvent s'appliquer sur n'importe quelle partie du document initial.

Exemple

Nous utilisons un document XML représentant des dossiers médicaux pour illustrer la sémantique du modèle de contrôle d'accès. Cet exemple sera utilisé tout au long de cette thèse. Une partie de ce document est représenté dans la Figure 57, avec les politiques de contrôle d'accès associées à trois profils d'utilisateurs : les secrétaires, les docteurs et les chercheurs du domaine médical. La secrétaire a seulement accès aux sous-dossiers administratifs des patients. Le docteur peut accéder à tous les sous-dossiers administratifs ainsi qu'à tous les actes médicaux à l'exception des détails des actes des patients dont il n'a pas la charge. Finalement, le chercheur ne peut accéder qu'aux résultats de laboratoire et à l'âge des patients qui ont souscrit à un test de protocole de type *G3*, à condition que la mesure de l'élément *Cholestérol* n'excède pas 250mg/dL.



Politique de contrôle d'accès du docteur
D1: ⊕, //Folder/Admin D2: ⊕, //MedActs[//RPhys = USER] D3: ⊖, //Act[RPhys != USER]/Details D4: ⊕, //Folder[MedActs//RPhys = USER]/Analysis
Politique de contrôle d'accès du chercheur
R1: ⊕, //Folder[Protocol]//Age R2: ⊕, //Folder[Protocol/Type=G3//LabResults//G3 R3: ⊖, //G3[Cholesterol > 250] Les règles 2 & 3 s'appliquent de manière similaire pour chacun des 10 groupes {G1, ..., G10}
1.1 Politique de contrôle d'accès de la secrétaire
S1: ⊕, //Admin

Figure 57. Document XML Hôpital

Les applications médicales illustrent bien le besoin d'avoir des règles d'accès dynamiques. Par exemple, un chercheur peut être autorisé à avoir accès à titre exceptionnel et cela pendant une durée limitée à une partie des dossiers médicaux dans le cas où le taux de *cholestérol* dépasse 300mg/dL (une situation plutôt rare). Un patient ayant souscrit à un protocole pour tester l'efficacité d'un nouveau traitement peut sortir du protocole à n'importe quel moment à cause par exemple d'une dégradation de son état de santé ou pour toute autre raison personnelle. Les modèles qui compilent les politiques de contrôle d'accès dans le schéma de chiffrement ne peuvent gérer cette dynamicité. Il est pourtant souvent nécessaire de chiffrer les données et de déléguer le contrôle d'accès sur le client : échange de données entre plusieurs équipes de recherche médicale d'une manière sécurisée en pair à pair, protection des données aussi bien contre les attaques externes que les attaques internes. Le dernier aspect est particulièrement important dans le domaine médical à cause du haut niveau de confidentialité requis par les données et du très haut niveau de décentralisation du système d'information (e.g., les petites cliniques et les médecins généralistes peuvent être amenés à déléguer la gestion de leur système d'information).

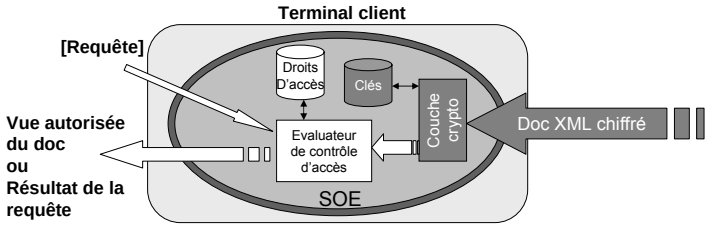


Figure 58. Architecture cible abstraite

Architecture cible

La Figure 58 donne une représentation abstraite de l'architecture cible pour les applications mentionnées dans l'introduction. Le contrôle d'accès étant évalué sur le client, le terminal client doit être rendu résistant à toute attaque grâce à un *Secure Operating Environment* (SOE). Comme mentionné dans l'introduction, le SOE peut reposer sur une solution logicielle ou matérielle ou un mélange des deux. Cependant, seuls les SOE matériels fournissent des garanties fortes de sécurité et donc seulement eux seuls seront considérés dans notre étude. Dans la suite de cette thèse, et cela jusqu'à la section évaluation, nous ne faisons aucune hypothèse sur le SOE, à l'exception des hypothèses traditionnelles : 1) le code exécuté par le SOE ne peut être corrompu, 2) le SOE possède au moins une petite quantité de stockage stable et sécurisée (pour stocker les données secrètes comme les clés de chiffrement), 3) Le SOE a au moins une petite quantité de mémoire de travail sécurisé (pour protéger les structures de données sensibles au moment du traitement).

Dans notre contexte, le SOE est en charge de déchiffrer le document d'entrée, de vérifier son intégrité et d'évaluer la politique de contrôle d'accès correspondant à un couple (document, sujet). La politique de contrôle d'accès ainsi que les clés requises pour déchiffrer le document peuvent être stockées de manière permanente dans le SOE, mises à jour ou

téléchargées via un canal sécurisé provenant de sources diverses (tiers de confiance, serveur de sécurité, parent ou enseignant, etc.).

4 Gestion en flux du contrôle d'accès

Bien que plusieurs modèles de contrôle d'accès XML aient été récemment proposés, peu de travaux portent sur la mise en pratique de ces modèles et à notre connaissance, aucun ne considère l'évaluation du contrôle d'accès en flux. A première vue, la gestion en flux du contrôle d'accès ressemble au problème bien connu du traitement de requêtes XPath sur des documents en flux. Beaucoup de travaux ont été réalisés sur ce problème dans le contexte de filtrage XML [43][30][61]. Ces études considèrent un grand nombre d'expressions XPath (généralement des dizaines de milliers). L'objectif principal est alors de trouver le sous-ensemble de requêtes XPath qui produisent une réponse pour un document donné (on ne s'intéresse pas aux parties du sous-document satisfaisant ces expressions XPath) et l'accent est mis sur l'indexation ou la manière de combiner l'ensemble de ces requêtes. L'un des premiers travaux adressant le problème précis de l'évaluation d'expressions XPath complexes sur des flux XML a été réalisé par Peng et Chawathe [95] qui propose une solution pour délivrer les parties du document qui satisfont une requête XPath unique. Bien que les règles d'accès soient aussi exprimées en XPath, la nature de notre problème diffère largement des travaux précédents. En effet, le principe de propagation des règles et les politiques de résolution de conflits associées (voir Section 3) rendent les règles d'accès dépendantes entre elles. L'interférence entre les règles introduit deux problèmes importants :

- *Evaluation de règles d'accès :* pour chaque nœud du document d'entrée, l'évaluateur doit être capable de déterminer l'ensemble des règles qui s'y applique et pour chaque règle déterminer si elle s'applique directement ou si elle est héritée. L'imbrication de la portée des règles d'accès détermine si ce nœud sera ou non autorisé. Ce problème est rendu plus complexe par le fait que des règles sont évaluées de manière paresseuse à cause de la présence de prédicats en attente.
- *Optimisation du contrôle d'accès :* L'imbrication de la portée des règles associée à la résolution de conflits peut inhiber l'effet de certaines règles. L'évaluateur de règles doit pouvoir tirer avantage de cette inhibition pour suspendre l'évaluation de ces règles et éventuellement de toutes les règles si une décision globale peut être établie pour un sous-arbre donné.

4.1 Evaluation des règles d'accès

Comme nous considérons des documents arrivant en flux, nous supposons que l'évaluateur de règles est alimenté par un parseur basé sur des évènements (e.g., SAX [102]) qui déclenche les évènements *ouverture*, *valeur* et *fermeture* respectivement pour chaque *ouverture*, *texte* et *fermeture* de tag du document d'entrée.

Nous représentons chaque règle d'accès (i.e., expression XPath) par un automate non-déterministe à états finis (NFA) [64]. La Figure 59.b représente les *Automates de Règles d'Accès* (*ARA*) correspondant à deux règles d'accès relativement simples exprimées sur le document XML abstrait. Cet exemple abstrait, utilisé ici à la place de l'exemple introduit à la Section 3, nous donne l'opportunité d'étudier plusieurs situations (dont les plus complexes) sur un document simple. Dans la représentation de nos ARA, un cercle désigne un état et un double cercle un état final, les deux étant identifiés par un identifiant unique d'état *IdEtat*. Les arcs orientés représentent des transitions, qui sont franchies par les évènements *ouverture* satisfaisant l'étiquette de l'arc (nom d'élément ou *). Par conséquent, les arcs orientés représentent l'axe XPath parent-enfant (/) ou un joker, cela dépendant de l'étiquette. Pour modéliser l'axe de descendance (//), nous ajoutons une transition réflexive étiquetée avec * qui est satisfaite par n'importe quel évènement d'*ouverture*. Un ARA se décompose en un *chemin de navigation* et de manière optionnelle d'un ou de plusieurs *chemins de prédicats* (en gris sur la figure). Pour gérer l'ensemble des ARA représentant une politique de contrôle d'accès, nous introduisons les structures de données suivantes :

- *Jetons et Piles de Jetons :* Nous faisons la distinction entre les *Jetons de navigation* (JN) et les *jetons de prédicats* (*JP*) suivant le chemin de l'ARA dans lequel ils sont impliqués. Pour modéliser la traversée d'un ARA par un jeton donné, nous créons en fait une copie du jeton à chaque fois qu'une transition est franchie et nous l'étiquetons avec l'identifiant *IdEtat* de l'état de destination. Les termes jeton et copie de jeton seront utilisés indifféremment dans la suite de la thèse. La progression de la navigation dans tous les *ARA* est mémorisée grâce à une structure de pile appelée *Pile de Jetons*. Le sommet de la pile contient tous les jetons actifs *JN* et *JP*, i.e., les jetons qui peuvent franchir une nouvelle transition au prochain évènement d'*ouverture*. Les jetons créés par une transition qui a été franchie sont empilés. La pile est dépilée à chaque évènement de *fermeture*. Le but de la *Pile de Jetons* est double : permettre de faire un retour en arrière de manière direct dans tous les *ARA* et réduire le nombre de jetons à vérifier à chaque événement (seuls sont considérés les jetons actifs, au sommet de la pile).
- *Statut des règles et la Pile d'Autorisation:* Supposons pour le moment que les expressions de règles d'accès n'exploitent pas l'axe de descendance (pas de //). Dans ce cas, une règle est dite *active* - signifiant que sa portée couvre le nœud courant ainsi que son sous-arbre – si tous les états finaux de son *ARA* contiennent un jeton. Une règle est dite *en attente* si l'état final du chemin de navigation contient un jeton alors que l'état final d'un chemin de prédicat n'a pas encore été atteint. La *Pile d'Autorisation* enregistre les jetons *JN* qui ont atteint l'état final de leur chemin de navigation, à une certaine profondeur dans le document. La portée d'une règle est limitée par la durée pendant laquelle son jeton *JN* reste dans la pile. Cette pile est utilisée pour résoudre les conflits entre les règles. Le statut d'une règle présent dans la pile peut prendre quatre valeurs différentes : *positive-active* (noté $\oplus$), *positive-en-attente* (noté $\oplus^?$), *négative-active* (noté $\ominus$), *négative-en attente* (noté $\ominus^?$). Par convention, l'élément en bas de la pile contient implicitement une

règle *négative-active* qui matérialise ainsi une politique de contrôle d'accès fermée (i.e., par défaut, l'ensemble des objets auxquels l'utilisateur a accès est vide).

- *Matérialisation des instances de règles* : La prise en compte de l'axe de descendance (*//*) dans les expressions des règles d'accès complexifie le problème. En effet, les mêmes noms d'éléments peuvent être rencontrés à différentes profondeurs dans le même document, amenant plusieurs jetons à atteindre leur état final du chemin de navigation et des chemins de prédicats du même *ARA* sans toutefois avoir de liens entre eux[9]. Pour résoudre cette situation, nous annotons les copies de jetons de navigation et de prédicat avec la *profondeur* à laquelle le jeton de prédicat a été créé, matérialisant ainsi sa participation à la même *instance de règle*[10]. Ainsi, un jeton doit garder les informations suivantes : *IdRegle* (noté *R*, *S*, ...), le statut du jeton (de navigation ou de prédicat), *IdEtat* et *Profondeur*[11]. Par exemple $Rn2_2$ et $Rp4_2$ (aussi noté 2_2, 4_2 pour simplifier les figures) désignent les jetons de navigation et de prédicat créés dans *l'ARA* de la règle *R* au moment où l'élément b est rencontré à la profondeur 2 dans le document. Quand la transition entre les états 4 et 5 de *l'ARA* est déclenchée, une copie de jeton $Rp5_2$ est créée et représentera alors la progression du jeton original $Rp4_2$ dans *l'ARA*. Tous ces jetons sont relatifs à la même instance de règle puisqu'ils sont annotés avec la même profondeur. Une instance de règle est dite *active* ou *en-attente* selon les mêmes critères qu'énoncés précédemment en prenant en compte seulement les jetons relatifs à cette instance.
- *Ensemble de Prédicats :* Cet ensemble mémorise les jetons *JP* qui ont atteint leur état final dans le chemin de prédicat. Un jeton *JP*, représentant une instance de prédicat, est supprimé de cet ensemble au moment où la profondeur courante dans le document devient inférieure à sa propre profondeur.

Les structures de données basées sur une pile sont bien adaptées à la traversée d'un document hiérarchique. Cependant, nous avons besoin d'un accès direct à tous les niveaux de la pile pour mettre à jour les informations en attente et pour permettre de faire certaines optimisations décrites dans la suite de la thèse. La Figure 59.c représente une exécution pas à pas basée sur ces structures de données. Cette exécution s'expliquant pratiquement d'elle-même, nous ne détaillons qu'un petit ensemble d'étapes.

[9] La complexité du problème a été soulignée par Peng et al. [95].

[10] Pour illustrer cela, considérons la règle *R* et le sous-arbre droit du document présenté à la Figure 59. L'état final 5 du chemin de prédicat (exprimant *//b[c]*) peut être atteint par deux instances différentes de b, une localisée à la profondeur 2 et l'autre à la profondeur 3 dans le document alors que l'état final 3 du chemin de navigation (exprimant *//b/d*) peut être atteint seulement par le b se trouvant à la profondeur 3. Ainsi, une seule instance de règle est valide ici, matérialisé grâce aux copies de jetons de navigation et de prédicat annotés avec la même profondeur 3.

[11] Si un même *ARA* contient des chemins de prédicats différents commençant à des niveaux différents du chemin de navigation, un jeton *JN* devra en plus enregistrer tous les jetons *JP* qui lui sont associés.

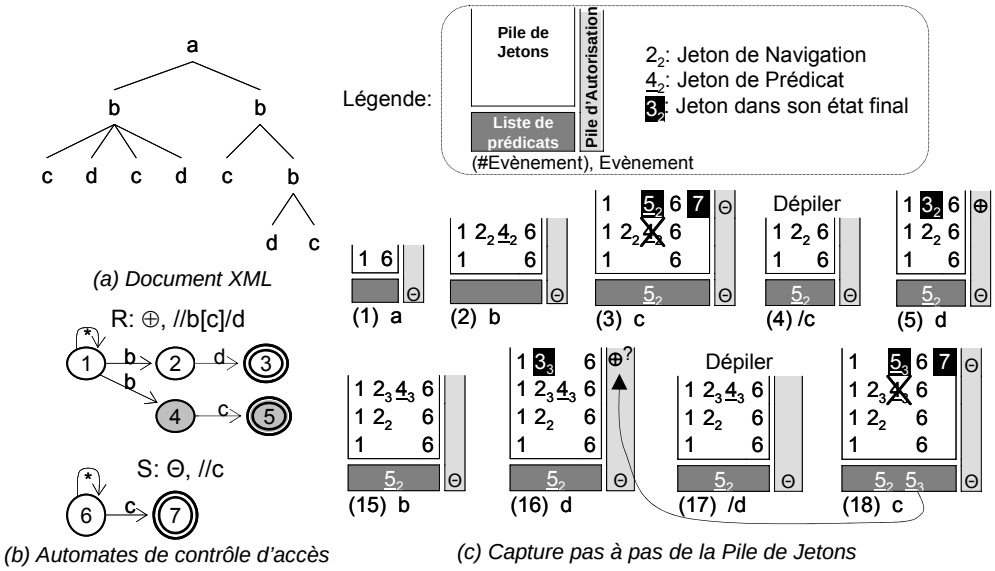


Figure 59. Capture pas à pas d'une exécution

- *Etape 2 :* L'évènement d'ouverture *b* génère deux jetons $Rn2_2$ et $Rp4_2$, participant à la même instance de règle.
- *Etape 3 :* L'*ARA* de la règle négative *S* a atteint son état final et une instance active de *S* est empilée dans la *Pile d'Autorisation*. L'autorisation courante reste négative. Le jeton $Rp5_2$ est inséré dans l'*Ensemble de Prédicats*. Le prédicat correspondant sera considéré comme vrai jusqu'à ce que la profondeur 2 de la *Pile de Jetons* soit dépilée (i.e., jusqu'à ce que l'événement */b* soit déclenché à l'étape 9). Par conséquent, il n'y a pas besoin de continuer à évaluer ce prédicat dans ce sous-arbre et le jeton $Rp4_2$ peut être supprimé de la *Pile de Jetons*.
- *Etape 5 :* Une instance active de la règle positive *R* est empilée dans la *Pile d'Autorisation*. L'autorisation courante devient positive, permettant de délivrer l'élément *d*.
- *Etape 16 :* Une nouvelle instance de *R* représentée par le jeton $Rn3_3$ est empilée dans la *Pile d'Autorisation*. Cette instance est en attente car le jeton $Rp5_2$ présent dans l'*Ensemble de Prédicats* à l'étape 12 (évènement *c*) ne participe pas à la même instance de règle.
- *Etape 18 :* Le jeton $Rp5_3$ est inséré dans l'*Ensemble des Prédicats*, changeant ainsi le statut de l'instance de la règle qui lui est associé en *positive-active*.

4.2 Résolution de conflits

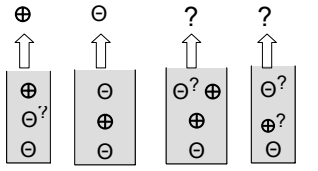
A partir des informations présentes dans la *Pile d'Autorisation*, l'issue du nœud courant peut être déterminée facilement. L'algorithme de résolution de conflits présenté dans la Figure 60 intègre la politique fermée de contrôle d'accès (ligne 1), les politiques *L'Interdiction-est-Prioritaire* (ligne 2) et *La-Plus-Spécifique-est-Prioritaire* (ligne 5 et 7) pour prendre une décision. Dans cet algorithme, *PA* désigne la *Pile d'Autorisation* et *PA[i].StatutsRegles*

désigne l'ensemble des statuts de toutes les règles qui se sont enregistrées au niveau *i* de la pile. Dans le premier appel à l'algorithme récursif, le niveau correspond à celui du sommet de *PA*. La récursion matérialise le fait qu'une décision peut être atteinte même si les règles au sommet de la pile sont en attente, et que celle-ci dépend du statut des règles qui se trouvent à un niveau plus bas de la pile. Il faut noter cependant qu'une décision peut rester en attente si une règle en attente se trouvant au sommet de la pile est en conflit avec d'autres règles. Dans ce cas, le nœud courant doit être bufferisé, en attendant de pouvoir être délivré par une condition. Ce problème est adressé et développé plus en détail dans la Section 6. Le reste de cet algorithme se comprend de lui-même et des exemples de résolutions de conflits sont donnés dans la figure.

L'algorithme *DecisionNoeud* présenté ci-dessous considère seulement les règles d'accès. Les choses deviennent plus complexes si des requêtes sont également considérées. Les requêtes sont exprimées en XPath et sont traduites en un automate non-déterministe à états finis de la même manière que les règles d'accès. Cependant, une requête ne peut être vue comme une règle d'accès au moment de la résolution des conflits. La condition pour délivrer le nœud courant d'un document devient double : (1) la décision de délivrance doit être vraie et (2) ce nœud doit être qualifié par la requête. La première condition est le résultat de l'algorithme *DecisionNoeud*. La seconde condition est satisfaite si la requête est *active*, c'est à dire si tous les états finaux de *l'ARA* de la requête contiennent un jeton, ce qui signifie que le nœud courant fait partie de la portée de la requête.

DecisionNoeud(profondeur) → Decision ∈ {⊕, ⊖,?}

```
1:  si profondeur = 0 alors retourne '⊖'
2:    sinon si '⊖'∈ PA[profondeur].StatutsRegles alors retourne '⊖'
3:      sinon si '⊕' ∈ PA[profondeur].StatutsRegles et
4:            '⊖?' ∉ PA[profondeur].StatutsRegles alors retourne '⊕'
5:        sinon si DecisionNoeud(profondeur -1) = '⊖' et
6:            ∀t∈{'⊕?','⊕'} t∉ PA[profondeur].StatutsRegles alors retourne '
7:          sinon si DecisionNoeud(profondeur -1) = '⊕' et
8:                '⊖?' ∉ PA[profondeur] StatutsRegles alors retourne '⊕'
9:            sinon retourne '?'
```



Exemples de résolution de conflits

Figure 60. Algorithme de résolution de conflits

4.3 Optimisations

La première optimisation qui peut être envisagée est de faire une analyse statique du système de règles qui composent la politique de contrôle d'accès. La propriété d'inclusion des règles peut être exploitée pour diminuer la complexité du système de règles. Notons $\subseteq$ la relation d'inclusion entre les règles *R, S, …T*. Si $S \subseteq R \wedge (R.Signe=S.Signe)$, l'élimination de *S* peut être envisagée. Mais cette élimination est à proscrire si par exemple, $\exists T \,/\, T \subseteq R \wedge (T.Signe \neq R.Signe) \wedge (S \subseteq T)$. De ce fait, les règles ne peuvent être examinées deux à deux et le problème revient à vérifier si un ordre partiel entre les règles peut être établi par rapport à la

relation d'inclusion, e.g., $\{T_i, \ldots T_k\} \subset \{S_i, \ldots S_k\} \subseteq \{R_i, \ldots R_k\} \wedge \forall i, (R_i.Signe=S_i.Signe \wedge S_i.Signe \neq T_i.Signe) \Rightarrow \{S_i, \ldots S_k\}$ peut être éliminé. Notons que cette condition forte d'élimination est suffisante mais pas nécessaire. Par exemple, soient *R* et *S* deux règles positives définies respectivement par */a* et */a/b[P1]*, et *T* une règle négative définie par */a/b[P2]/c*. *S* peut être éliminée alors que $T \not\subset S$, car l'inclusion est valable pour tous les sous-arbres où les deux règles sont actives en même temps. Ce problème est particulièrement complexe sachant que le problème d'inclusion de requête est à lui seul un problème co-NP complet pour la classe de XPath considérée, c'est à dire $XP^{\{[],//,*\}}$ [83]. Ce problème pourrait être étudié plus en détail sachant que des résultats favorables ont été trouvés pour des sous-classes de $XP^{\{[],//,*\}}$ [83], mais ce travail est hors du contexte de notre étude.

Une deuxième forme d'optimisation est de suspendre dynamiquement l'évaluation des ARA qui deviennent non-pertinents ou inutiles dans un sous-arbre. Nous pouvons pour cela exploiter les différentes informations contenues dans la *Pile de Jetons*, la *Pile d'Autorisation* et l'*Ensemble de Prédicats*. La première optimisation est alors de suspendre l'évaluation d'un prédicat d'un sous-arbre aussitôt qu'une instance de ce prédicat a été évaluée à vrai dans le sous-arbre. Cette optimisation a été illustrée par l'étape 3 de la Figure 59.c. La seconde optimisation est d'évaluer dynamiquement la relation d'inclusion entre les règles actives et en attente et de tirer profit de la condition d'élimination décrite précédemment. La *Pile d'Autorisation* permet de détecter les situations où la condition s'applique localement : $(T \subset S \subseteq R) \wedge (R.Signe=S.Signe \wedge S.Signe \neq T.Signe)$, les niveaux de la pile reflétant la relation d'inclusion dans le sous-arbre courant. Ainsi *S* peut être inhibé dans ce sous-arbre. Même s'il est profitable de suspendre l'évaluation d'une règle, il faut garder à l'esprit que les deux facteurs limitants de notre architecture sont le coût de déchiffrement et de communication. Par conséquent le vrai challenge est d'être capable de prendre une décision pour l'ensemble d'un sous-arbre, une condition nécessaire pour détecter et sauter les sous-arbres non-autorisés afin de réduire les coûts de déchiffrement et de communication

Sans information supplémentaire sur le document d'entrée, une décision peut être prise pour l'ensemble d'un sous-arbre de racine *n* ssi : (1) l'algorithme *DecisionNoeud* peut retourner une décision *D* ($\oplus$ ou $\ominus$) pour *n* lui-même et (2) aucune règle *R* dont le signe est contraire à *D* ne peut devenir active dans le sous-arbre (ce qui signifie que tous ses états finaux, de son chemin de navigation et des chemins de prédicat, ne peuvent être atteints simultanément). Ces deux conditions sont compilées dans l'algorithme présenté à la Figure 61. Dans cet algorithme, *PA* représente la *Pile d'Autorisation*, *PJ* la *Pile de Jetons*, *PJ[i].JN* (resp. *PJ[i].JP*) l'ensemble des jetons *JN* (resp. *JP*) enregistrés au niveau *i* de la pile et Sommet qui correspond au sommet de la pile. De plus, *j.InstRegle* désigne l'instance de la règle associée à un jeton donné, *Regle.Signe* le signe de cette règle et *Regle.Pred* un booléen indiquant si cette règle inclut des prédicats dans sa définition.

Le bénéfice immédiat de cet algorithme est de pouvoir arrêter l'évaluation de jetons *JN* actifs et le bénéfice principal attendu est de pouvoir sauter des sous-arbres complets si la décision est $\ominus$. Notons cependant que seuls les jetons *JN* sont supprimés de la pile à la ligne 6. La

raison est que les jetons JP actifs doivent être tout de même considérés, sinon des prédicats en attente pourraient rester en attente indéfiniment. En résumé, un sous-arbre de racine *n* peut être sauté ssi : (1) la décision pour *n* est ⊖, (2) L'algorithme *DecisionSousArbre* retourne ⊖ et (3) si aucun jeton *JP* ne se trouve au sommet de la *Pile de Jetons* (qui est alors vide).

```
DecisionSousArbre() → Decision ∈ {⊕, ⊖,?}
1:  D = DecisionNoeud(PA.sommet)
2:  si D = '?' alors retourne '?'
3:  si non (∃ jn ∈PJ[sommet].JN / jn.Regle.Signe ≠ D
4:              et (non jn.Regle.Pred
5:                      ou (∃ jp ∈PJ[sommet].JP / jp.InstRegle = jn.InstRegle))
6:  alors PJ[sommet].JN = ∅; retourne (D)
7:  sinon retourne '?'
```

Figure 61. Décision sur un sous-arbre complet

Malheureusement ces conditions sont rarement réunies simultanément, en particulier quand l'axe de descendance est utilisé dans les expressions de règles et de prédicat. La section suivante introduit une structure d'*Index de Saut* qui donne des informations utiles à propos du contenu à venir du document. Le but de cet index est de détecter à priori les règles et les prédicats qui deviendront non-pertinents, augmentant ainsi la probabilité de réunir les conditions précédemment mentionnées.

Quand les requêtes sont considérées, tout sous-arbre n'étant pas inclut dans la portée de la requête est candidat à un saut. Cette situation s'applique aussitôt que le jeton *JN* d'une requête (ou les jetons JN quand plusieurs instances d'une même requête peuvent co-exister) devient inactif (i.e., n'est plus un élément de *PJ[sommet].JN*). Ce jeton peut être supprimé de la *Pile de Jetons* mais les jetons *JP* doivent toutefois être considérés, encore une fois pour empêcher que les prédicats en attente restent en attente indéfiniment. Comme précédemment, le sous-arbre sera sauté si la *Pile de Jeton* devient vide.

5 Index de Saut

Cette section introduit une nouvelle forme de structure d'index, appelé *Index de Saut*, conçu pour détecter et sauter les fragments non-autorisés (selon la politique de contrôle d'accès) et les fragments non-pertinents (d'après une requête potentielle) d'un document XML, tout en satisfaisant les contraintes introduites par l'architecture cible (document chiffré en flux, peu de capacité mémoire du SOE).

La première contrainte que doit respecter l'index est la nécessité de le garder chiffré en dehors du SOE pour garantir qu'aucune information ne soit révélée. La deuxième contrainte (liée à la première et à la capacité du SOE) est que le SOE doit gérer l'index en flux, de la même manière que le document lui-même. Ces deux contraintes nous amènent à concevoir un index très compact (le surcoût induit par son déchiffrement et sa transmission ne doit pas excéder son propre bénéfice) et intégré dans le document d'une manière compatible avec le

flux. Pour ces raisons nous nous concentrons sur l'indexation de la structure du document, en mettant de côté les index sur son contenu. Des résumés structurels [8] ou un squelette XML 0[28] pourraient être considérés comme des candidats pour cet index. Cependant, en dehors du fait qu'ils peuvent poser un problème de taille ou de lecture en flux, ces approches ne tiennent pas compte des irrégularités des documents XML (e.g., il est très possible que les dossiers médicaux diffèrent d'une instance à l'autre alors qu'ils partagent la même structure générale).

Dans la suite nous proposons un index structurel très compact, encodé récursivement dans le document XML pour permettre une gestion en flux. Une conséquence intéressante du schéma d'indexation proposé est de permettre de compresser encore mieux les parties structurelles du document.

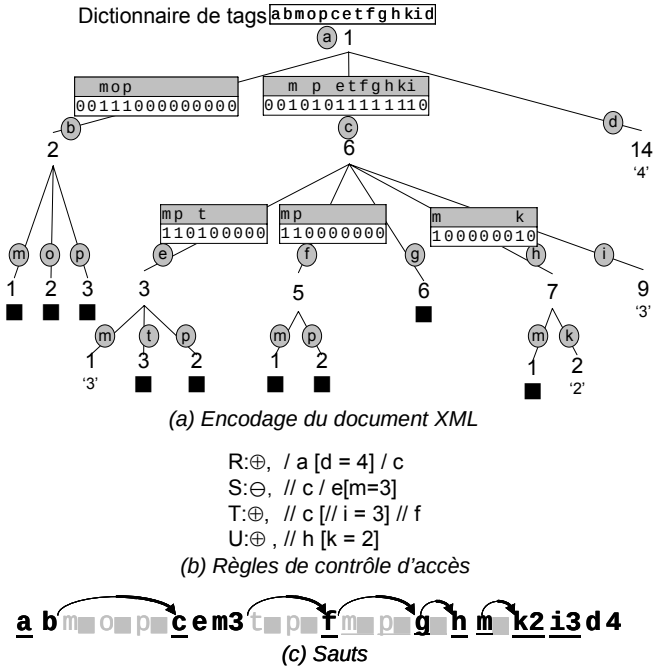
5.1 Format d'encodage de l'Index de Saut

L'objectif principal de cet index est de détecter les règles et les requêtes qui ne peuvent s'appliquer à l'intérieur d'un sous-arbre, avec comme bénéfice attendu de pouvoir sauter le sous-arbre si les conditions mentionnées dans la Section 4.3 sont réunies. En gardant à l'esprit la nécessité d'avoir un index compact, l'information structurelle minimale requise pour atteindre notre objectif est l'ensemble des tags d'élément (ou tags) qui sont présents dans chaque sous-arbre. Bien que ces méta-données ne prennent pas en compte l'imbrication des tags, elles se révèlent comme étant une manière très efficace de filtrer les expressions XPath non-pertinentes. Nous proposons ci-dessous les structures de données qui encodent ces méta-données d'une manière très compacte. Ces structures de données sont illustrées dans la Figure 62.a sur un exemple abstrait de document XML.

- *Encoder l'ensemble des tags des descendants* : La taille du document d'entrée étant un problème, nous faisons l'hypothèse assez classique que la structure du document est compressée grâce à un dictionnaire de tags [8][109][12]. L'ensemble des tags qui sont présents dans le sous-arbre de racine *e*, appelé $TagsDesc_e$, peut être encodé sur un tableau de bits, appelé $TabTags_e$, de longueur N_t, où N_t est le nombre d'entrées dans le dictionnaire de tags. Un encodage récursif peut réduire davantage la taille de ces méta-données. Notons *TagsDesc(e)* la fonction bijective qui associe $TabTags_e$ au dictionnaire de tags pour calculer $TagsDesc_e$. Nous pouvons diminuer la taille de l'index au prix d'une complexité de calcul plus importante en réduisant l'image de *TagsDesc(e)* en $TagsDesc_{parent(e)}$ à la place du dictionnaire de tags. La taille de la structure *TabTags* diminue ainsi au fur et à mesure que l'on descend dans la hiérarchie du document mais la fonction *TagsDesc()* devient alors récursive. Puisque le nombre d'éléments augmente généralement avec la taille du document, le gain est considérable. Pour faire la distinction entre les nœuds intermédiaires et les feuilles (qui n'ont pas besoin de la méta-donnée

[12] Considérer la compression du contenu du document lui-même est hors du contexte de notre étude. Cela étant, elle peut être utilisée conjointement avec notre proposition tant que le schéma de

TabTags), un bit supplémentaire est ajouté à chaque nœud.



(a) Encodage du document XML

R:⊕, / a [d = 4] / c
S:⊖, // c / e[m=3]
T:⊕, // c [// i = 3] // f
U:⊕ , // h [k = 2]

(b) Règles de contrôle d'accès

(c) Sauts

Figure 62. Exemple d'index de saut

- *Encoder les tags d'éléments* : Dans une compression basée sur un dictionnaire, le tag de chaque élément *e* du document est remplacé par une référence à l'entrée correspondante dans le dictionnaire. $Log_2(N_t)$ bits sont nécessaires pour encoder cette référence. L'encodage récursif de l'ensemble des tags peut être exploité également pour compresser encore mieux l'encodage des tags eux-mêmes. En utilisant ce schéma d'encodage, $Log_2(TagsDesc_{parent(e)})$ bits suffisent à encoder le tag d'un élément *e*.

- *Encoder la taille d'un sous-arbre:* Encoder la taille de chaque sous-arbre est nécessaire pour implémenter l'opération de saut. A première vue, $Log_2(taille(document))$ bits sont requis pour encoder $TailleSousArbre_e$, la taille du sous-arbre de racine *e*. De la même manière, un schéma récursif permet de réduire l'encodage de la taille à $Log_2(TailleSousArbre_{parent(e)})$ bits. De plus, en stockant la taille du sous-arbre pour chaque élément, les tags de fermeture deviennent inutiles.

- *Décoder la structure du document* : Le décodage de la structure du document doit être fait par le SOE de manière efficace, en flux et sans consommer trop de mémoire. Pour cela, le SOE stocke le dictionnaire de tags et utilise une pile interne *PileSaut* pour enregistrer le *TagsDesc* et le *TailleSousArbre* de l'élément courant. Quand on décode un élément *e*, $TagsDesc_{parent(e)}$ et $TailleSousArbre_{parent(e)}$ sont récupérés de la pile et utilisés pour décoder à leur tour $TabTags_e$, $TailleSousArbre_e$ et le tag encodé de *e*.

- *Mettre à jour le document :* Dans le pire des cas, mettre à jour un élément *e* résulte en la

compression reste compatible avec les ressources du SOE.

mise à jour de *TailleSousArbre*, de *TabTags* et de l'encodage du tag de chaque ancêtre de l'élément *e* et de leurs fils directs. Dans le meilleur des cas, seulement le *TailleSousArbre* des ancêtres de *e* ont besoin d'être mis à jour. Le pire cas arrive dans deux situations assez peu courante. L'information *TailleSousArbre* des fils des ancêtres de *e* doit être mis à jour si la taille du père de *e* augmente (resp. diminue) et saute une puissance de 2. L'information *TabTags* et l'encodage du tag des fils des ancêtres de *e* doivent être mis à jour si la mise à jour de *e* génère une insertion ou une suppression dans le dictionnaire de tags.

5.2 Utilisation de l'Index de Saut

Comme mentionné précédemment, l'objectif principal de l'*Index de Saut* est de détecter les règles et les requêtes qui ne peuvent s'appliquer à l'intérieur d'un sous-arbre. Cela signifie que tous les jetons actifs qui ne peuvent atteindre un état final dans leurs *ARA* peuvent être supprimés du sommet de la *Pile de Jetons*. Soit *LabelsRestants(j)* la fonction qui détermine l'ensemble des tags de transitions rencontrés sur le chemin séparant l'état courant d'un jeton *j* de son état final sur son *ARA*, et soit *e* l'élément courant dans le document. Un jeton *j* , de navigation ou de prédicat, ne pourra atteindre un état final de son *ARA* que si $LabelsRestants(j) \not\subset TagsDesc_e$. Remarquons que cette condition est suffisante mais pas nécessaire puisque l'*Index de Saut* ne prend pas en compte l'imbrication des tags.

SautSousArbre () → Decision ∈ {vrai, faux}

1: pour chaque jeton j ∈PJ[sommet].JN ∪ PJ[sommet].JP
2: Si LabelsRestants(j) ⊄ $TagsDesc_e$ alors supprimer j de PJ[sommet]
3: si DecisionSousArbre () ∈ {'⊖', '?'} et (PJ[sommet].JN = ∅) et
4: (PJ[sommet].JP = ∅) alors retourne vrai
5:sinon retourne faux

Figure 63. Décision de saut

Une fois que ce filtrage de jetons a été fait, la probabilité que l'algorithme *DecisionSousArbre* atteigne une décision pour le sous-arbre dont la racine est l'élément courant *e* est beaucoup plus grande puisque plusieurs règles non-pertinentes ont été filtrées. Si cette décision est négative (⊖) ou en attente (?), le saut du sous-arbre peut être envisagé. Ce saut est en fait possible s'il n'y a plus de jetons actifs, de navigation ou de prédicat au sommet de la *Pile de Jetons*. L'algorithme *SautSousArbre* donné en Figure 63 décide si le saut est possible ou non. Remarquons que cet algorithme pourra être déclenché lors des événements d'ouverture et de fermeture de tags. En effet, chaque élément peut changer la décision retournée par l'algorithme *DecisionNoeud*, et donc *DecisionSousArbre* et finalement *SautSousArbre* avec comme bénéfice de pouvoir sauter un plus grand sous-arbre à l'étape suivante. La Figure 62 montre un exemple illustratif de document XML, son encodage, un ensemble de règles d'accès et les sauts réalisés lors de l'analyse du document. Les informations en gris sont présentées pour simplifier la compréhension du schéma d'indexation mais ne sont pas stockées dans le document. Considérons l'analyse du

document (pour des raisons de clarté, nous utiliserons dans la suite le tag de l'élément réel au lieu de son encodage). Au moment où l'élément *b* (du sous-arbre le plus à gauche) est rencontré, toutes les règles actives sont stoppées grâce à $TabTags_b$ et le sous-arbre complet peut ainsi être sauté (la décision est $\ominus$ à cause de la politique de contrôle d'accès fermée). Quand l'élément *c* est atteint, la règle *R* devient en attente. Cependant, l'analyse du sous-arbre continue puisque $TabTags_c$ ne permet pas de filtrer davantage. Quand l'élément *e* est atteint, $TabTags_e$ filtre les règles *R*, *T* et *U*. La règle *R* devient négative-active quand la valeur '3' est rencontrée dans l'élément m.

A l'événement de fermeture de tag, l'algorithme *SautSousArbre* décide de sauter le sous-arbre de racine *e*. Cette situation illustre bien le bénéfice de déclencher *SautSousArbre* à chaque événement d'ouverture et de fermeture de tags. L'analyse continue en suivant le même principe et aboutit à délivrer les éléments soulignés dans la Figure 62.c

6 Gestion des prédicats en attente

Un élément du document en entrée est dit *en attente* si sa délivrance dépend d'une règle en attente, c'est-à-dire d'une règle pour laquelle l'état final de son chemin navigationnel a été atteint mais pour laquelle au moins un des états finaux de ses chemins de prédicats n'a pas encore été atteint. Ce cas défavorable est malheureusement fréquent. En effet, toute règle de la forme */…/e[P]/…/* génère une situation en attente tant que *P* n'a pas été évaluée à vrai. En effet, une évaluation à faux de *P* n'arrête pas la situation en attente parce qu'une autre instance de *P* pourrait devenir vraie quelque part d'autre dans le document.

Cette situation est rendu d'autant plus difficile car les parties en attente ne peuvent être bufferisées dans le SOE si on considère les hypothèses faites sur sa capacité mémoire. De plus, quand plusieurs prédicats en attente sont considérés, leurs gestions peuvent devenir particulièrement complexes. Dans la suite, nous présentons deux manières différentes de gérer le problème, chacun étant adapté à un contexte différent. Puis nous fournissons une solution pour gérer plusieurs prédicats en attente.

6.1 Délivrance de parties en attente

Par nature, les prédicats en attente sont incompatibles avec les applications consommant les documents en flux pur (on pourra remarquer qu'un prédicat peut rester en attente jusqu'à la fin du document). Quand on considère des prédicats en attente, nous faisons l'hypothèse que le terminal a assez de mémoire pour bufferiser les parties en attente du document. Nous proposons d'abord une solution simple adapté aux documents délivré en flux pur (e.g., accès push, broadcast). Puis, nous décrivons une solution plus précise adaptée aux contextes où les accès en arrière et en avant sont autorisés dans le document (e.g., accès pull).

Flux pur : Lorsque l'on reçoit le document en flux pur, l'issue des prédicats en attente ne

peut être connu à l'avance. Les parties en attente ne peuvent être bufferisées à l'intérieur du SOE si on considère les hypothèses faites sur sa capacité mémoire. Pour résoudre ce problème, les parties en attentes sont externalisées au terminal sous forme chiffrée en utilisant une clé temporaire de chiffrement. Si après, ultérieurement dans le parsing, les parties en attente sont déterminées comme étant autorisées alors la clé temporaire est délivrée ; autrement elle est détruite. Une clé temporaire de chiffrement différente est générée pour chaque partie en attente dépendante de prédicats différents. On appellera dans la suite *bloc sortant*, un bloc sortant contigu chiffré avec la même clé ou un bloc sortant contigu en clair.

Chaque bloc sortant délivré B_i (chiffré ou non) peut inclure des informations supplémentaires afin qu'il puisse être intégré de manière cohérente dans le résultat final quand des éléments ancêtres ou frères sont dans une situation en attente. Cette information est appelée *ListeTagsChemin* de B_i et contient la liste des tags dans le chemin entre le dernier élément délivré autorisé et la racine du sous-arbre B_i. En effet, si les ancêtres de B_i sont finalement déterminés comme étant interdits, cette liste est nécessaire pour appliquer la règle *structurale* qui dit que le document résultat doit garder la même structure que le document d'origine (cf., Section 3)[13]. Afin d'éviter la confusion entre les éléments partageant le même tag pendant l'intégration de B_i, tous les éléments en attente qui sont délivrés sont marqués par un identifiant (e.g., une valeur aléatoire). Ces valeurs sont gardés dans le SOE et sont associées aux tags de la *ListeTagsChemin* de B_i[14].

Accès en arrière/en avant : Dans ce cas, nous faisons l'hypothèse que les parties en attente peuvent être relues (e.g., du serveur) quand des prédicats en attente sont résolus. L'objectif est alors de détecter les sous-arbres en attente et de les laisser de côté grâce à *l'Index de Saut* jusqu'à ce que la situation en attente soit résolu. Le but est de ne jamais lire et analyser les mêmes données deux fois. La stratégie de saut et la stratégie de de reassemblage proposées ci-dessous permettent d'atteindre cet objectif.

Les sous-arbres en attente sont externalisés au moment où l'expression logique conditionnant leur délivrance est évaluée à vraie (e.g., *//a[d=6]/b[c=5]* nécessite qu'un élément *d=6* et un élément *c=5* soient déterminés comme vrai). Par conséquent, les sous-arbres en attente peuvent être délivrés dans un ordre différent de l'ordre initial de celui du document en entrée. Le bénéfice de cet asynchronie est de réduire la latence de la gestion du contrôle d'accès et de libérer la mémoire interne du SOE, au prix d'un réassemblage complexe du résultat final. En effet, les relations père, descendants et frères entre les éléments doivent être conservées au moment du reassemblage. Cela nous oblige à enregistrer au moment du parsing les informations *<adresse, niveau, ancre, condition>* pour chaque sous-arbre en

[13] Une solution alternative serait de marquer les ancêtres interdits avec des valeurs factices pour être en accord avec les travaux de Gabillon et al. [59] et Fan et al. [53].

[14] On pourra remarquer que le surcoût du marquage peut être restreint à celui des ancêtres grâce à l'*Index de Saut* (le marquage est désactivé dès qu'aucune règle en attente ne peut s'appliquer dans le sous-arbre).

attente (La Figure 64 illustre une situation en attente impliquant quatre sous-arbres S_0, S_1, S_2, S_3 associé respectivement aux prédicats en attente P_0, P_1, P_2, P_3). L'*adresse* et le *niveau* correspondent respectivement à l'adresse et à la profondeur du sous-arbre dans le document initial (on utilise le terme niveau pour éviter la confusion avec la profondeur attachée aux jetons) ; L'*ancre* référence la position dans le sous-arbre résultat (voir ci-dessous) ; *condition* est l'expression logique qui conditionne la délivrance du sous-arbre. Cette information est conservée pour chaque sous-arbre en attente dans une liste nommée *Liste d'Attente* et dénotée par *LA*. Le mécanisme de reassemblage est comme suit :

- *Assignation de l'ancre :* Supposong que chaque élémént *e* dans le document résultat soit marqué par un nombre unique *Ne* (représentant par exemple l'ordre dans lequel les éléments sont délivrés). La position future d'un sous-arbre en attente *e'* dans le résultat peut être identifiée de manière unique par un nombre en utilisant la convention suivante : *Ne* si *e'* est un frère droit potentiel de *e* ou *–Ne* si *e'* est le premier fils gauche potentiel de *e*. Aucune ancre n'a besoin d'être mémorisée pour des frères droits et des sous-arbres imbriquées d'un sous-arbre en attente *e'* (Dans la Figure 64, $N_{SO} = -3$ alors que *S1, S2, S3* n'ont pas d'ancre). Cela s'explique par deux raisons : (1) ces sous-arbres partagent la même ancre que *e'* jusqu'à ce que l'un de leurs frère gauche (voir *délivrance des sous-arbre*) ou ancêtres (voir *sous-arbres en attente imbriqués*) soit délivré et que (2) les relations de parent et de frère entre les éléments en attente peuvent être reconstituées à partir de la *Liste d'Attente* comme suit :

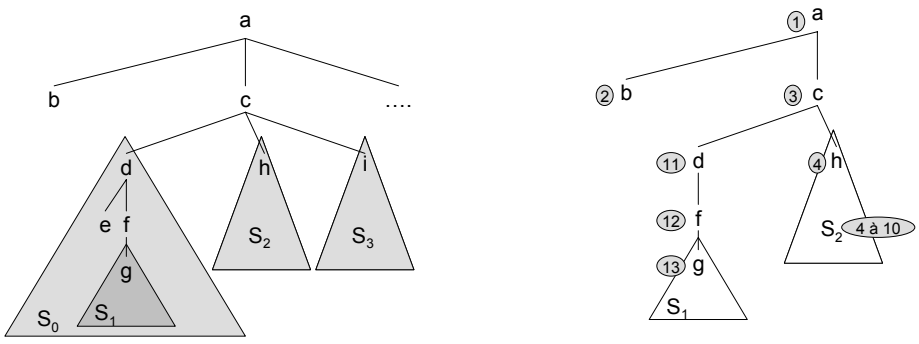
  π dénote la relation de précédence dans la *Liste d'Attente*
  A fils de B $\Leftrightarrow$ BπA $\wedge$ Niveau$_A$=Niveau$_B$+1 $\wedge$ $\neg$ ($\exists$ C / BπCπA $\wedge$ Niveau$_C$=Niveau$_B$)
  A frère droit de B $\Leftrightarrow$ BπA $\wedge$ Niveau$_A$=Niveau$_B$ $\wedge$ $\neg$ ($\exists$ C / BπCπA $\wedge$ Niveau$_C$$\leq$Niveau$_A$)

- *Sous-arbres en attente imbriqués* : un sous-arbre en attente peut à son tour imbriquer d'autres sous-arbres en attente, ce qui peut générer des situations complexes dépendant de l'issue des prédicats en attente (i.e., qui peut affecter l'ordre de délivrance). Supposons que le sous-arbre interne $S_{interne}$ (e.g. S_1 soit déterminé comme étant autorisé alors que le sous-arbre externe $S_{externe}$ (e.g., S_0) est encore en attente. Tous les tags sur le chemin de $S_{interne}$ à son dernier ancêtre autorisé *Anc* (e.g., *d* et *f* connectent *S1* à l'élément *c*) doivent être délivrés dans leur ordre hiérarchique, avec $S_{interne}$ pour appliquer la règle *Structurelle*. Supposons maintenant que $S_{externe}$ soit délivré après. $S_{interne}$ et le chemin qui le connecte à *Anc* ne doivent pas être délivrés deux fois. Pour résoudre cette situation et donner un document résultat cohérent, les parties de $S_{externe}$ délivrées précédemment doivent être enregistrées. Ces informations sont stockées dans une liste de parties à sauter intégrée dans la structure *LA* (voir la liste des parties à sauter de S_0 dans la Figure 64).

- *Délivrance des sous-arbres* : Au moment où un sous-arbre en attente *e'* est délivré, sa place dans le document résultat est déterminée par son ancre. A son tour, *Ne'* (resp. *–Ne'*) devient l'ancre du frère droit en attente (resp. le premier fils gauche en attente) de *e'*, si il existe (e.g., l'ancre S_3 est mise à jour quand l'ancre S_2 est délivrée). Pour délivrer

le sous-arbre *e'*, le sous-arbre complet est relu à partir du document en entrée, déchiffré et délivré en prenant garde à sauter les éléments qui ont déjà été délivrés (i.e., sous-arbres imbriqués autorisés, tags permettant de garantir la règle structurelle) et ceux qui ont pu ne pas être délivrés (sous-arbre encore en attente et sous-arbre interdits).

La Figure 64 illustre le problème des sous-arbres imbriqués en attente, montrant l'état de la *Liste d'Attente* a différents moments de son exécution. La figure se comprend d'elle-même.



(a) Document initial avec des sous-arbres en attente (b) Document délivré (avec des ancres)

Structure de la LA : *pour chaque sous-arbre : (adresse, niveau, ancre, condition, Liste des parties à sauter)*

Etat initial : PL = (($@_{S0}$, 3, -3, P_0, ($@_{S1}$)), d, f, ($@_{S1}$, 5, Nul, P_1, Nul), ($@_{S2}$, 3, Nul, P_2, Nul), ($@_{S3}$, 3, Nul, P_3, Nul))
P_2 = vrai : PL = (($@_{S0}$, 3, -3, P_0, ($@_{S1}$)), d, f, ($@_{S1}$, 5, Nul, P_1, Nul), ($@_{S3}$, 3, 4, P_3, Nul))
P_1 = vrai : PL = (($@_{S0}$, 3, -3, P_0, (($@_d$,11), ($@_f$, 12), ($@_{S1}$, 13)), ($@_{S3}$, 3, 4, P_3, Nul))
P_0 = faux : PL = (($@_{S3}$, 3, 4, P_3, Nul))
P_3 = faux : PL = ()

(c) Capture de la Liste d'Attente

Figure 64. Gestion des prédicats en attente

6.2 Gestion de plusieurs prédicats en attente

Les conditions enregistrées dans la *Liste d'Attente* peuvent être des expressions logiques complexes. A chaque fois qu'un prédicat en attente est résolu, ces conditions doivent être évaluées pour déterminer si certains sous-arbres en attente peuvent être délivrés. Quand on considère plusieurs prédicats en attente, il est nécessaire de les organiser de telle façon à minimiser le surcoût de stockage des conditions de la *Liste d'Attente* et le temps requis pour leur évaluation.

Appellons *Classe en Attente* l'ensemble des sous-arbres en attente dont la délivrance dépend de la même expression logique. Comme le nombre de prédicats en attente sera vraisemblablement petit, les expressions logiques peuvent être modélisées par un vecteur de bits représentant la table de vérité. Pour minimiser la consommation mémoire, deux vecteurs *V* et *B* sont associés à chaque *Classe en Attente*. *V={(p, d)}* est la liste des prédicats identifiés par un identifiant de prédicat *p* et sa profondeur *d* à laquelle il s'applique ; *B* est le résultat de la table de vérité représentant l'expression logique (dans la Figure 65, seulement les parties grisées sont stockées en mémoire, le reste étant implicite). En utilisant des opérations

binaires sur les vecteurs de bits (e.g., *ET* bit à bit, décalage de bit), *V* et *B* peuvent être étendus incrémentalement au fur et à mesure que des nouveaux prédicats en attente sont considérés, ou peuvent être réduits quand des prédicats en attente sont résolus. Cette table de vérité évolue jusqu'à devenir équivalent à une fonction qui retourne toujours vrai (toutes les données associées sont alors délivrées) ou à une fonction qui retourne toujours faux (toutes les données associées sont alors interdites).

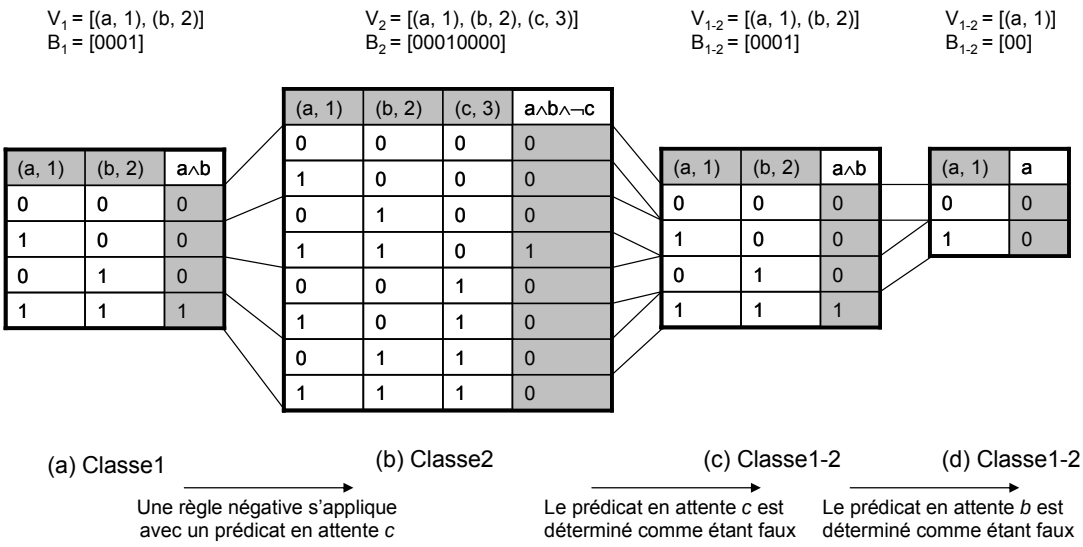


Figure 65. Gestion de plusieurs prédicats en attente

Figure 65 illustre comment les expressions logiques sont construites incrémentalement et évaluées. Supposons qu'initialement on ait une *Classe en Attente Classe1* contenant le sous-arbre S_1 de racine *n1* dont la condition de délivrance est $a \wedge b$, où les prédicats *a* et *b* s'appliquent respectivement à la profondeur 1 et 2. Supposons maintenant qu'un élément *f* soit un descendant de n_1 sur lequel une nouvelle règle en attente *R* s'applique avec un prédicat *c*. Si *R* est négative (resp. positive), l'expression logique conditionnant la délivrance de l'élement *f* est $a \wedge b \wedge \neg c$ (resp. $a \wedge b \wedge c$). Pour prendre en compte cette nouvelle situation, une nouvelle classe *Classe2* est créée avec un vecteur V_2 calculé à partir de V_1 (le prédicat *c* est inséré dans la liste dans l'ordre lexical) et d'un vecteur de bits B_2 construit comme suit (nous supposons que B_2 doit représenter l'expression $a \wedge b \wedge \neg c$). Un vecteur de bits *B'* est créé à partir de B_1 en duplicant chaque segment de $2^{index(c)-1}$ bits, *index(c)* étant la position du prédicat *c* dans V_2 (e.g. $index((c, 3)) = 3$). Le prédicat *c* est représenté par un vecteur de bits B_c constitué d'une alternance de segments de $2^{index(c)-1}$ bits à 0 et à 1. Finalement, B_2 est calculé grâce à l'opération ET NON entre *B'* et B_c (Figure 65.b). Supposons que le prédicat *c* soit déterminé comme faux, dans ce cas les lignes codant une valeur vraie pour *c* dans B_2 sont alors supprimées et le reste est décalé vers le haut (Figure 65.c), c'est-à-dire toutes les lignes des intervalles $[k*2^{index(c)} + 2^{index(c)-1}, (k+1)*2^{index(c)}[$, *k* variant de *0* à $|V| - 1$ où $|V|$ représente la cardinalité de *V*. Le résultat de la table de vérité de *Classe2* étant le même que celui de la C*lasse1*, les deux classes sont regroupées en une seule classe *Classe1-2*.

Supposons que le prédicat *b* soit évalué à faux, la table est alors réduite comme dans la Figure 65.d. Le résultat de la table de vérité ne contient que des bits égaux à zéro, par conséquent l'expression logique est toujours fausse et tous les éléments de la classe sont alors supprimés.

Cette solution est efficace du point de vue temps et en terme d'espace si on considère que les opérations logiques opèrent sur des tableaux de bits qui sont vraisemblablement susceptibles d'être encodés sur des nombres de 32 bits.

7 Gestion des droits d'accès

Comme mentionné dans l'introduction, la dynamicité des politiques de contrôle d'accès est une fonctionnalité obligatoire pour un grand nombre d'applications. Cela nous amène à concevoir un mécanisme sécurisé pour rafraichir les règles de contrôle d'accès sur le SOE. De la même manière que les données, les définitions des règles de contrôle d'accès peuvent être stockées sous forme chiffrées sur le serveur. Cependant, le serveur peut ne pas être de confiance et donner une réponse incorrecte au SOE quand celui-ci demande une mise à jour des règles de contrôle d'accès ou un document. La confidentialité, l'intégrité et l'authenticité sont garantis par le chiffrement et des mécanismes de hachage présentés dans la Section 8. Dans cette section, nous nous concentrons sur les attaques de rejeu. Les attaques de rejeu consistent à envoyer au SOE des versions incohérentes des règles de contrôle d'accès et du document afin de tromper l'évaluateur de contrôle de règles. Les attaques de rejeu ne violent pas la confidentialité des données tant qu'aucun accès non-autorisé n'est donné à un utilisateur. Par exemple, si le serveur délivre une ancienne version cohérente du document et des règles de contrôle d'accès, l'utilisateur aura exactement le même résultat que ce qu'il a obtenu dans le passé. Même si ces données ne sont plus autorisées au moment présent, l'utilisateur n'obtiendra pas d'accès supplémentaire à d'autres information.

Le problème se pose quand les règles de contrôle d'accès sont définis sur des parties du document qui sont mises à jour. Appliquer une nouvelle politique de contrôle d'accès sur une vieille version du document peut révéler des parties obsolètes non autorisées. De la même manière, appliquer une ancienne politique de contrôle d'accès sur une version récente du document peut révéler des parties à jour non-autorisées.

Ces deux problèmes peuvent être résolus en utilisant un système de versions basé sur des références croisées entre les données et les droits d'accès. Les droits d'accès de chaque utilisateur[15] sont stockés sur le serveur sous la forme de tuples (*ts*, *droit*, *docts*, *sig*) où *ts* dénote une estampille qui est incrémentée pour chaque nouvelle règle de contrôle d'accès, *droit* est la définition de la règle de contrôle d'accès sous sa forme chiffrée, *docts* est l'estampille du document au moment où la règle de contrôle d'accès a été défini et *sig* est la

[15] Dans le cas où l'on a un grand nombre d'utilisateurs, une table pourra être partagée par un groupe d'utilisateurs qui prendra en compte les droits d'accès communs et individuels.

signature du tuple. De la même manière, le document contient un entête signé constitué d'une estampille *docts* et pour chaque utilisateur la dernière estampille *ts* au moment où le document a été mis à jour.

Quand le SOE demande une mise à jour des droits, il récupère toutes les règles de contrôle d'accès de l'utilisateur associé qui ont un *ts* supérieur à celui obtenu lors de la dernière connexion. Puis il vérifie que les droits d'accès sont correctement chainés entre eux grâce à leur estampille (aucune règle ne manque). Quand il récupère le document, il vérifie que le *docts* (resp. *ts*) du document est supérieur (resp. inférieur) au *docts* (resp. *ts*) contenu dans le dernier droit d'accès, garantissant ainsi que le document et les droits d'accès sont cohérents même si ils ne sont pas à jour.

8 Vérification de l'intégrité sur des accès aléatoires

Des techniques de chiffrement et de hachage sont requises pour garantir respectivement la confidentialité et l'intégrité des documents. Malheureusement, les méthodes standard de vérification d'intégrité ne sont pas adaptées dans notre contexte pour deux raisons majeures. Premièrement, la limitation mémoire du SOE impose un contrôle de l'intégrité en flux. Deuxièmement, la vérification de l'intégrité doit prendre en compte les accès en avant et en arrière générés par l'*Index de Saut* et par le réassemblage des fragments en attente. Dans cette section, nous présentons une solution pour faire face aux attaques potentielles sur le document source.

Dans un contexte basé sur le client, le pirate peut être l'utilisateur lui-même. Par exemple, un utilisateur ayant acquis l'accès à un dossier médical *X* peut essayer d'extraire les information non autorisées d'un dossier *Y*. Supposons que le document soit chiffré avec un algorithme classique de chiffrement par bloc (e.g., DES ou triple-DES) et que ces blocs soient chiffrés indépendemment (e.g., en suivant le mode ECB [77][103]). Dans ce cas, des blocs en clairs identiques vont générer des blocs chiffrés identiques. Ainsi, le pirate peut conduire différentes attaques : en substituant des blocs du dossier *X* et *Y* pour tromper le gestionnaire de contrôle d'accès et déchiffrer ainsi les parties de *Y* ; en construisant un dictionnaire de couple texte en clair/texte chiffré à partir d'information autorisée (e.g., le dossier *X*) et en l'utilisant pour déduire des information du texte chiffré (e.g., le dossier *Y*) ; en faisant des inférences statistiques sur le texte chiffré. En plus, si aucun contrôle d'intégrité n'a lieu, le pirate peut modifier aléatoirement quelques blocs, entrainant un mauvais fonctionnement de l'évaluateur de règles (e.g., Bob est autorisé à accéder au dossier des patients agés de plus de 80 ans et il modifie aléatoirement le texte chiffré contenant l'âge des patients).

Pour faire face à ces attaques, nous exploitons deux techniques. Concernant le chiffrement, l'objectif est de générer des textes chiffrés différents pour des instances de la même valeur. Cette propriété est obtenue en utilisant le chainage de blocs chiffrés (CBC) au lieu de ECB, ce qui signifie que le chiffrement d'un bloc dépend du bloc précédent [77][103]. Cependant,

cela introduirait un surcoût important au niveau du temps de déchiffrement si des accès aléatoires sont effectués sur le document. Comme alternative, nous combinons la position d'une donnée avec la donnée elle-même, i.e., nous effectuons un XOR (noté $\oplus$) entre chaque bloc de 8 octets et la position du bloc dans le document avant de chiffrer le résultat en utilisant un simple mode de chiffrement ECB. Le chiffrement lui-même est effectué en utilisant l'algorithme de *Triple-DES* mais d''autres algorithmes peuvent être utilisés dans ce but (e.g., *AES*). Par conséquent, un bloc en clair *b* à la position *p* dans le document est chiffré par $E_k(b \oplus p)$, où *k* est la clé secrète attachée au document et stockée dans le SOE. La clé *k* peut être stockée de manière permanente dans le SOE ou peut être téléchargée de manière sécurisée en même temps que les règles de contrôle d'accès associées au couple (document, utilisateur).

Le chiffrement à lui seul n'est pas suffisant pour garantir l'intégrité du document puisque le pirate peut effectuer des modifications aléatoires et des substitutions dans le texte chiffré. Nous proposons dans la suite une solution où l'intégrité est vérifié par le SOE en coopération avec le terminal non-sécurisé. Une solution simple est de découper le document en morceaux (e.g., 2Ko) et d'utiliser une fonction de hachage résistante aux collisions (e.g., SHA-1) pour calculer le hachage *RésuméMorceau* de chaque morceau et ainsi empêcher toute forme de modification illicite de se produire sans impact sur ce résumé. Chaque morceau contient un identifiant qui indique sa position dans le document de telle sorte qu'une substitution de blocs puisse être facilement détectée. Quand le SOE accède à *n* octets à la position *pos* dans un morceau, le terminal calcule le hachage des *pos-1* octets précédents et transmets les résultats intermédiaires au SOE. Puisque le hachage est calculé incrementalement, le SOE peut continuer le calcul du hachage sur les données chiffrées et vérifier l'intégrité des données reçues en comparant le hachage final avec le *RésuméMorceau*. On remarquera que *RésuméMorceau* doit être chiffré pour empecher le terminal de calculer lui-même un nouveau résumé correspondant aux données modifiées illicitement. Cette solution entraine d'envoyer *taille(RésuméMorceau)* + *taille(Morceau)* – *(pos* – *1)* octets au SOE et de déchiffrer *taille(RésuméMorceau)* + *n* octets dans le SOE.

Bien que correct, la solution précédente limite le bénéfice des petits sauts dans le document puisque le morceau cible doit toujours être lu par le SOE à partir de la position *pos* demandé jusqu'à la fin. Ainsi *taille(Morceau) – (pos-1) – n* octets non pertinents doivent être reçus par le SOE. Pour alléger cette faiblesse, nous adaptons le principe de l'arbre de hachage de Merkle introduit dans [78] comme suit. Chaque morceau est découpé en *m* fragments (e.g., 256 octets), où *m* est une puissance de 2, et ces *m* fragments sont organisés dans un arbre binaire. Une valeur de hachage est calculé pour chaque fragment et attaché à chaque feuille de l'arbre binaire. Chaque nœud intermédiaire de l'arbre contient une valeur de hachage calculé comme la concaténation des valeurs de hachage de ses nœuds fils. Le *RésuméMorceau* correspond à la valeur de hachage attachée à la racine de l'arbre binaire. Quand le SOE accède *n* octets à la position *pos* dans le fragment *f* d'un morceau donné, le terminal envoie : 1) les octets à partir de la position *pos* jusqu'à *taille(fragment) – (pos-1)*

octets incluant les *n* octets demandés ; 2) le calcul des hachages intermédiaires des *pos -1* octets du fragment *f* ; 3) Les informations de hachage calculées à partir des autres fragments suivant la stratégie de l'arbre de hachage de Merkle ; et 4) le *RésuméMorceau* chiffré. Grâce à ces information, le SOE peut recalculer la racine de l'arbre de hachage de Merkle et la comparer au *RésuméMorceau* comme illustré dans la Figure 66.

Pour conclure, le document est protégé contre toute modification illicite et attaque sur la confidentialité tout en restant agnostic vis à vis des algorithmes de chiffrement utilisés pour chiffrer une donnée élémentaire. Contrairement à l'approche développée par Bouganim et Pucheral [20] et Hacigumus et al. [62], nous ne faisons aucune hypothèse sur la manière de chiffrer les données ou qui puissent faciliter l'exécution des requêtes au prix d'une robustesse plus faible contre les attaques par cryptanalyse.

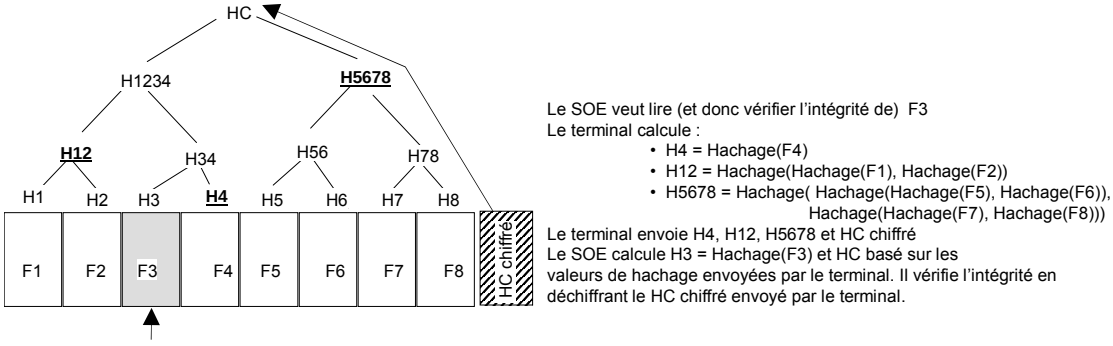


Figure 66. Vérification de l'intégrité à accès aléatoire

9 Résultats expérimentaux

Cette section présente les résultats expérimentaux obtenus à partir de jeux de données synthétiques et réels. Nous donnons d'abord les détails sur la plate-forme d'expérimentation. Nous analysons ensuite le surcoût du stockage induit par *l'Index de Saut* et le comparons avec des variantes possibles. Puis nous analysons les performances de la gestion du contrôle d'accès et de l'évaluation de requêtes. Finalement, les performances globales de la solution proposée sont évaluées sur quatre jeux de données ayant chacune des caractéristiques différentes.

Plate-forme d'expérimentation

L'architecture cible abstraite présentée dans la Section 3 peut être instanciée de plusieurs manières différentes. Dans ces expérimentations, nous considérons que le SOE est intégré dans une plate-forme avancée de carte à puce. Bien que les cartes à puce existantes soient déjà puissantes (processeur 32 bits cadencé à 30Mhz, 4 KO de RAM, 128KO d'EEPROM), elles sont trop limitées pour supporter notre architecture, plus particulièrement en terme de

bande passante de communication (9.6Kbps). Notre partenaire industriel Axalto a annoncé pour la fin de l'année la sortie de cartes à puce équipées d'un processeur 32 bits cadencé à 40Mhz, 8KO de RAM, 1MO de Flash et supportant un protocole USB à 1MBps. Axalto nous a fourni un simulateur permettant de faire des mesures au cycle près pour la carte à puce à venir. Notre prototype a été développé en C et a été évalué en utilisant ce simulateur. La précision au cycle près garantit une prédiction exacte des performances qui seront obtenues avec la plate-forme matérielle cible.

Comme cette section le montrera, notre solution est principalement limitée par les coûts de déchiffrement et de communication. Les chiffres donnés dans la Table 3 nous permettent de faire des projections sur les performances obtenues dans cette section pour des architectures cibles différentes. Les chiffres donnés pour le débit de communication correspondent au cas le plus défavorable où chaque donnée lue par le SOE prend part dans le résultat. Le coût de déchiffrement correspond à celui de l'algorithme 3DES, implanté dans un module matériel de la carte à puce (ligne 1) et mesuré sur un PC cadencé à 1Ghz (ligne 2 et 3).

Contexte	Communication	Déchiffrement
Matériel (e.g., cartes à puce futures)	0.5 MO/s	**0.15 MO/s**
Logiciel – Connexion Internet	**0.1 MO/s**	1.2 MO/s
Logiciel – Connexion dans un réseau local	10 MO/s	**1.2 MO/s**

Table 3. Coûts de communication et de déchiffrement

	WSU	**Sigmod**	**Treebank**	**Hôpital**
Taille	1.3 MO	350KO	59MO	3.6 MO
Taille du texte	210KO	146KO	33MO	2,1 MO
Profondeur maximale	4	6	36	8
Profondeur moyenne	3.1	5.1	7.8	6.8
# tags distincts	20	11	250	89
# nœuds texte	48820	8383	1391845	98310
# éléments	74557	11526	2437666	117795

Table 4. Caractéristiques des documents

Dans cette expérience, nous considérons trois jeux de données réels : *WSU* correspondant à des cours d'université, *Sigmod records* qui contient un index d'articles et *Treebank* qui contient des phrases annotées avec des parties de discours [114]. De plus, nous avons généré des documents synthétiques pour l'exemple de l'Hôpital décrit dans la Section 3 (les jeux de données réels sont difficiles à obtenir dans ce domaine), grâce au générateur ToXgene [110]. Les caractéristiques principales de ces documents sont résumées dans la Table 4.

Surcoût de stockage de l'Index de Saut

L'*Index de Saut* est une compilation de trois techniques pour encoder respectivement les tags, les listes des tags des descendants et les tailles des sous-arbres. Des variantes de l'Index de Saut peuvent être envisagées en combinant ces techniques différemment (e.g., encoder les tags et les tailles du sous-arbres sans encoder les listes des tags des descendants).

Ainsi, pour évaluer le surcoût associé à chacune de ces méta-données, nous comparons les techniques suivantes. NC correspond au document original non compressé. TC correspond à une compression classique de tags et servira de référence. Dans TC, chaque tag est encodé

par un nombre exprimé sur *Log_2(#tags distincts)* bits. Nous désignons par TCS (compression des tag et taille des sous-arbres) la méthode qui stocke la taille des sous-arbres pour permettre de sauter les sous-arbres. La taille du sous-arbre est encodée sur *Log_2(taille du document compressé)* bits. Dans TCS, le tag fermant est inutile et peut être supprimé. TCSB complémente TCS avec un bitmap des tags des descendants encodé sur *#tag distincts* bits pour chaque élément. Finalement, TCSBR est la variante récursive de TCSB et correspond en fait à l'Index de Saut décrit dans la Section 5. Dans toutes ces méthodes, les méta-données doivent être alignées sur une frontière d'octet. La Figure 67 compare ces cinq méthodes sur les jeux de données réels introduits précédemment. Ces jeux de données ayant des caractéristiques différentes, l'axe des ordonnées est exprimé en terme de taux *structure/(taille du texte)*.

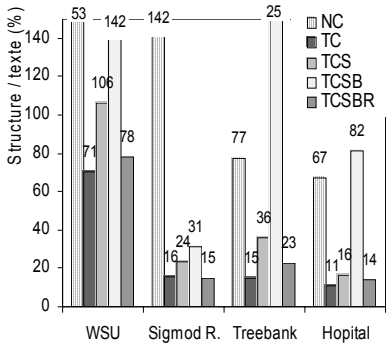


Figure 67. Surcoût du stockage de l'index

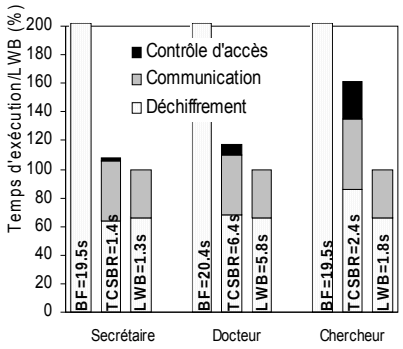


Figure 68. Surcoût du contrôle d'accès

Surcoût du contrôle d'accès

Clairement, TC réduit de manière importante la taille de la structure de tous les jeux de données. En ajoutant la taille des sous-arbres aux noeud (TCS), la taille de la structure passe de 50% à 150% (les gros documents nécessitent un encodage sur 5 octets pour la taille des sous-arbres et le tag de l'élément alors que de plus petits documents nécessitent seulement 3 octets). Le bitmap des tags des descendants (TCSB) a un coût encore plus élevé, plus particulièrement dans le cas du document Treebank qui contient 250 tags distincts. TCSBR réduit de manière importante le surcoût de stockage en le ramenant à une valeur proche de celle obtenue avec TC. Cela s'explique par le fait que la taille de la structure diminue rapidement, tout comme le nombre de tags distincts dans chaque sous-arbre. Pour le document Sigmod, TCSBR offre une meilleure compacité que TC.

Pour évaluer l'efficacité de notre stratégie (basé sur TCSBR), nous le comparons avec : (i) une stratégie force brute (BF) qui filtre le document sans utilisation d'aucun index et (ii) une borne inférieure (LWB). Les performances de LWB ne peuvent être atteintes par aucune stratégie et elles correspondent au temps que mettrait un oracle pour ne lire que les fragments autorisés du document et les déchiffrer. Evidemment, cet oracle doit pouvoir prédire l'issue de tous les prédicats – en attente ou non – sans avoir à les vérifier et deviner où sont les données pertinentes dans le document.

La Figure 68 montre le temps d'exécution requis pour évaluer la vue autorisée des trois profils (Secrétaire, Docteur et Chercheur) introduits dans la Section 3 sur le document Hôpital. La vérification de l'intégrité n'est pas prise en compte ici. La taille du document compressé est de 2,5MO et l'évaluation de la vue autorisée retourne 135KO pour la Secrétaire, 575KO pour le Docteur et 95KO pour le Chercheur. Pour pouvoir comparer ces trois profils malgré la différence de taille des résultats, l'axe des ordonnées représente le rapport entre le temps d'exécution et sa borne inférieure LWB respective. Le temps d'exécution réel est reporté dans chacun des histogrammes. Pour mesurer l'impact d'une politique de contrôle d'accès relativement complexe, nous considérons que le Chercheur est autorisé à accéder à 10 protocoles médicaux au lieu d'un, chacun d'eux étant exprimé par une règle positive et une règle négative comme décrit dans la Section 3.

Les conclusions qui peuvent être tirées de cette figure sont triples. Premièrement la stratégie force brute (BF) donne des performances désastreuses, ce qui s'explique par le fait que la carte doit lire et déchiffrer tout le document pour pouvoir l'analyser. Deuxièmement, la performance de notre stratégie TCSBR est en général très proche de LWB (rappelons-nous que LWB est une borne inférieure théorique et non-atteignable), ce qui met en avant l'importance de minimiser le flux entrant dans le SOE. La différence la plus importante entre ces deux stratégies est observée pour le profil du chercheur à cause des prédicats exprimés sur l'élément protocol qui peut rester en attente jusqu'à la fin de chaque élément folder. En effet, si ce prédicat est évalué à faux, l'évaluateur continuera – sans nécessité dans notre cas – à chercher une autre instance de ce prédicat. Troisièmement, le coût du contrôle d'accès (de 2% à 15%) est largement dominé par le coût de déchiffrement (de 53% à 60%) et par le coût de communication (de 30% à 38%). Le coût du contrôle d'accès est déterminé par le nombre de jetons actifs qui doivent être gérés en même temps. Ce nombre dépend du nombre d'*ARA* dans la politique de contrôle d'accès, du nombre de transition de descendance (*//*) et des prédicats présents dans chaque *ARA*. Cela explique un coût plus important pour la politique de contrôle d'accès du chercheur.

Impact des requêtes sur les performances

Pour mesurer de manière précise l'impact des requêtes sur les performances globales, nous considérons la requête *//Folder[//Age>v]* (où *v* nous permet de faire varier la sélectivité de la requête) qui est exécutée sur cinq vues différentes construites à partir des profils correspondant à : la secrétaire (S), un docteur à temps partiel (PTD) qui a à sa charge quelques patients, un docteur à temps plein (FTD) qui a à sa charge un grand nombre de patients, un chercheur junior (JR) ayant accès à quelques résultats d'analyses et un chercheur senior (SR) qui aura accès à un grand nombre de résultats d'analyses.

La Figure 69 représente le temps d'exécution de la requête (incluant celui du contrôle d'accès) comme une fonction de la taille du résultat de la requête. Le temps d'exécution diminue linéairement plus la sélectivité de la requête et de la vue augmentent, ce qui montre la précision de TCSBR. Même si le résultat de la requête est vide, le temps d'exécution n'est

pas nul puisque des parties du document doivent être analysées avant d'être sautées. Les parties du document qui nécessitent d'être analysées dépendent de la vue et de la requête. La figure en vignette montre la même linéarité pour des tailles plus importantes du résultat de la requête.

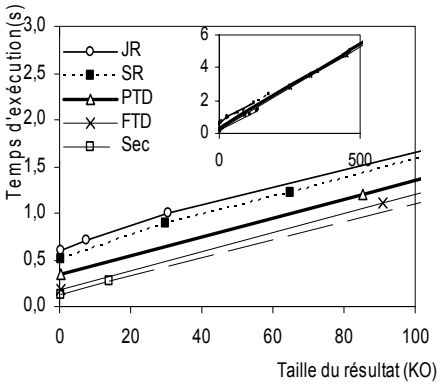


Figure 69. Impact des requêtes

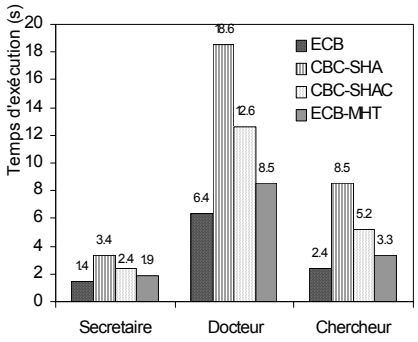


Figure 70. Surcoût du contrôle d'intégrité

Surcoût du contrôle d'intégrité

La Figure 70 représente le temps d'exécution requis pour construire la vue autorisée des profils de la Secrétaire, du Docteur et du Chercheur, incluant la vérification de l'intégrité. En comparant ces résultats avec la Figure 68, on voit que le coût dédié à la vérification de l'intégrité reste acceptable quand on utilise la technique proposée Section 8 (de 32% à 38%). Pour évaluer le bénéfice de cette technique basé sur ECB et l'arbre de hachage de Merkle (ECB-MHT), nous la comparons avec : (ECB) un mode de chiffrement ECB sans hachage qui garantit la confidentialité mais pas la modification illicite des données ; (CBC-SHA) un mode de chiffrement CBC complémenté par le hachage SHA-1 qui s'applique sur le texte en clair de blocs complets (cette solution représente l'application directe des techniques de l'état de l'art) ; (CBC-SHAC) qui est similaire à CBC-SHA excepté que les hachages s'appliquent aux blocs chiffrés, permettant ainsi au SOE de vérifier l'intégrité des blocs sans avoir à les déchiffrer. Les résultats représentés dans la figure se comprennent d'eux-même.

Performance sur des jeux de données réels

Pour évaluer la robustesse de notre approche et montrer qu'elle supporte des structures de documents très différentes, nous mesurons les performances de notre prototype sur trois jeux de données réels WSU, Sigmod et Treebank. Pour ces documents nous générons des règles d'accès aléatoires (incluant des // et des prédicats). Chaque document possède des caractéristiques différentes intéressantes. Le document Sigmod est bien structuré, non récursif, de profondeur moyenne et la politique de contrôle d'accès générée est simple et peu sélective (50% du document est renvoyé). Le document WSU est peu profond et contient une grande quantité de très petits éléments (sa structure représente 78% de la taille du document

après indexation par TCSBR). Le document Treebank est de très grande taille, contient un nombre important de tags qui apparaissent récursivement dans le document et la politique de contrôle d'accès générée est complexe (8 règles). La Figure 71 reporte ces résultats. Nous avons ajouté dans la figure les mesures obtenues avec le document Hôpital comme base de comparaison. La figure représente les temps d'exécution en terme de débit pour notre méthode et pour la borne inférieure LWB, chacune d'elle avec et sans vérification de l'intégrité.

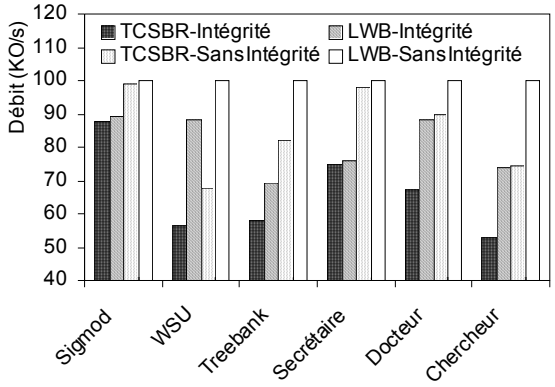


Figure 71. Performance sur des jeux de données réels

Nous montrons que notre méthode gère bien les différentes situations et produit un débit allant de 55KO/s à 85KO/s dépendant du document et de la politique de contrôle d'accès considérés. Ces résultats préliminaires sont encourageants comparés aux bandes passantes Internet obtenues actuellement avec de l'xDSL (de 16KO/s à 128 KO/s).

10 Applications et expérimentations

Notre prototype en C fonctionnant sur un simulateur matériel qui fournit des mesures au cycle CPU près, nous a permis de faire des prévisions quand aux performances de notre solution de gestion sur le client de contrôle d'accès sur des plateformes futures de cartes à puce et d'évaluer ainsi sa viabilité sur le moyen terme. Par ailleurs, notre équipe a développé un second prototype en JavaCard 2.2.1 et fonctionnant sur une plateforme réelle de carte à puce d'Axalto (SIMera [12]). L'objectif de ce second prototype est de démontré la versatilité de l'approche du point de vue applicatif. En vue d'atteindre ce but, deux scénarios ont été choisis pour exhiber deux profils d'applications relativement différentes concernant la manière dont l'information est accédé (pull vs. push), le type d'information (textuel vs. vidéo) et les contraintes de temps de réponses (patience de l'utilisateur / temps réel). Nous reportons brièvement les expériences ci-dessous.

La première application adresse le contexte du travail collaboratif. Une communauté d'utilisateurs désire organiser un espace confidentiel de partage de données via un DSP non

sûr pour échanger des données XML textuelles comme des agendas, carnets d'adresse, profils, documents de travail, etc. Notre technologie permet de définir facilement des droits d'accès puissant sur des données sensibles (e.g., des rendez-vous spécifiques dans un agenda ou une section financière dans un document de travail) tout en gérant la dynamicité des règles (de nouveaux partenaires peuvent se joindre ou quitter une communauté et des relations entre eux peuvent évoluer). Cette expérience nous a permis de remporter le prix Argent au concours international de logiciel de carte à puce *e-gate open 2004* [10].

Le deuxième scénario démontre une dissémination sélective de flux multimédia à travers des canaux non sécurisés. Les vidéos sont encodés en utilisant le standard MPEG7, qui permet de stocker des courtes descriptions de scènes dans des méta-données XML. Les exigences du contrôle parental et des modèles avancés des DRM peuvent être remplies grâce à des contrôles d'accès spécifiques et dynamiques. Pour résoudre le problème dû à la faiblesse de la bande passante des carte à puce actuelles, nous avons fait un compromis entre la sécurité et les performances. Ainsi, les méta-données et les flux vidéos sont séparés et seulement les métadonnées traversent la carte à puce. Cette expérience a été faite sur des téléphones portables équipés avec la nouvelle génération de cartes SIM. Elle a été récompensée par le prix Or du concours international de carte à puce sur téléphone portable *SIMagine 2005* [11].

Finalement, les composants internes de notre solution ont été démontrés à la conférence internationale SIGMOD'05 où l'accent a été mis sur le moteur d'automates non-déterministes à états finis et le sur la structure de l'*Index de Saut*. Les éléments de cette démonstration sont disponibles en ligne [24].

11 Conclusion et perspectives

11.1 Résumé

Plusieurs facteurs importants motivent aujourd'hui la délégation du contrôle d'accès aux terminaux clients. En compilant les politiques de contrôle d'accès dans le schéma de chiffrement, les solutions existantes de gestion du contrôle d'accès sur le client minimisent la confiance requise sur le client au prix d'un partage statique des données. Notre objectif est de tirer profit de nouveaux éléments de confiance présents sur les terminaux clients pour proposer une gestion du contrôle d'accès capable d'évaluer des règles d'accès dynamiques sur un document XML chiffré.

Dans cette thèse nous avons fait 5 contributions. Premièrement, nous avons proposé une évaluation en flux de règles de contrôle d'accès supportant un sous-ensemble conséquent du langage XPath. A notre connaissance, il s'agit de la première étude proposant une gestion du contrôle d'accès XML en flux. Deuxièmement, nous avons conçu une structure d'index en flux permettant de sauter les parties non pertinentes du document d'entrée, en fonction de la politique de contrôle d'accès et d'une requête éventuelle. Cet index est essentiel pour réduire

le goulot d'étranglement inhérent à l'architecture cible, à savoir le coût de déchiffrement et de communication. Bien que des méthodes plus complexes peuvent être envisagées, l'*Index de Saut* proposé combine la simplicité (une fonctionnalité importante dans les logiciels embarqués) et les performances sont proches de l'optimal (aussi bien en terme de taille que d'efficacité). Troisièmement, nous avons développé une gestion élégante des prédicats en attente compatible avec une délivrance en flux des parties autorisées du document. Quatrièmement, nous avons conçu un mécanisme sécurisé pour rafraichir les droits d'accès sur le SOE pour se prémunir contre un serveur malicieux qui peut utiliser des attaques de rejeu pour gagner accès à des données interdites. Cinquièmement, nous avons proposé une combinaison de techniques de hachage et de chiffrement pour rendre la vérification de l'intégrité possible malgré les accès en avant et en arrière aléatoires générés par les contributions précédentes.

Nos résultats expérimentaux ont été obtenus à partir d'un prototype écrit en C et tournant sur un simulateur matériel de carte à puce fourni par Axalto permettant de faire des mesures au cycle CPU près. Le débit global observé est d'environ 70KO/s et le coût relatif moyen du contrôle d'accès est inférieur à 20% du coût total. Ces premières mesures sont prometteuses et démontrent la viabilité de notre solution. Un prototype Javacard a été développé sur une plateforme de carte actuelle d'Axalto pour démontrer la versatilité de notre approche du point de vue applicatif. Ce prototype a permis de gagner 2 concours bien connus de carte à puce.

11.2 Perspectives de recherche

Dans cette thèse, nous avons montré que les solutions de sécurité sur le client mérite une attention toute particulière pour les nouvelles perspectives de recherche qu'elles ouvrent et pour leurs impacts à venir sur un nombre croissant d'applications. Nous décrivons ci-dessous certaines perspectives de recherche qui peuvent être envisagées :

- *Etendre notre solution pour prendre en compte les variations des autres modèles de contrôle d'accès pour XML*

  Les modèles de contrôles d'accès pour XML [36][54][84] convergent maintenant vers l'utilisation de vues sécurisés où l'on considère des restructurations de document afin d'empêcher des attaques d'inférence. Dans ces modèles, un élément du document original peut se retrouver à une place différente dans la vue sécurisée, plus particulièrement : (1) son numéro d'ordre dans la liste des nœuds fils peut être modifié, (2) il peut être rattaché à l'un de ses ancêtres (réduction de chemin), (3) un de ses ancêtres peut être anonymisé. Dans ce cas, nous devrons concevoir de nouvelles techniques de bufferisation pour que l'évaluation puisse se faire en flux tout en restant compatible avec les contraintes du SOE. Quand les éléments sont délivrés dans le résultat, ils devront être accompagnés d'informations supplémentaires pour pouvoir être intégrés correctement dans le résultat final (d'une manière similaire à XML Pool

Encryption [60]), tout en garantissant de ne pas divulguer de l'information pouvant être utilisée dans une attaque d'inférence. Ainsi, quand un élément est délivré, sa position dans le résultat peut être rendue floue en omettant son numéro ordre dans la liste des nœuds fils ou en omettant des informations sur ses ancêtres pour l'intégrer dans le document. Une autre extension possible est de considérer des accès provisionnel [72] où on devra enrichir la définition des règles de telles sorte à ce qu'un utilisateur gagne accès à une donnée si une condition est satisfaite. Cette condition peut être un cas d'urgence, un accord avec un autre utilisateur, etc. Par exemple, dans le cas d'urgence, une infirmière peut passer outre le contrôle d'accès pour avoir un accès complet au dossier médical d'un patient à condition que tout ses accès soient enregistrés de manière permanente dans le SOE.

- *Considérer de nouvelles générations de SOE*
  SSMSC (Smart Secure Mass Storage Cards) est un nouveau composant sécurisé qui combine sur le même support matériel, une puce sécurisé et une mémoire de stockage Flash d'une capacité de plusieurs gigaoctets. D'un point de vue technique, ce problème est similaire à notre contexte où un SOE a accès à un grand volume de données d'un serveur non sécurisé. Dans le cas du SSMSC, les accès entre la puce sécurisé et les données sont rapides car ils sont sur le même support physique. Cependant, la technologie Flash a une gestion complexe des accès en écriture (écriture incrémentale de blocs, effacement avant les mises à jour). Comme dans ce modèle, les accès en lecture sont rapides, il n'y a pas besoin d'avoir une gestion en flux des données. Par conséquent, nous pouvons envisager une nouvelle forme d'évaluation du contrôle d'accès bénéficiant d'un index à accès direct aux données. Bien evidemment, on devra vérifier l'intégrité de l'index et des données. Dans ce modèle, mettre à jour une donnée peut s'avérer être très coûteuse, mais ajouter une donnée a un coût raisonnable. Cela interdit l'utilisation de méthodes traditionnelles d'indexation comme les B-Tree qui ont besoin d'être équilibrée et donc mis à jour régulièrement. Ces paradigmes sont assez similaires aux problèmes adressés dans les mémoires de stockage WORM (Write Once Read Many) mais offre tout de même plus de flexibilité. Nous pouvons donc étudier de nouvelles techniques basées sur l'indexation incrémentale, basée sur des arbres non-équilibrées et des index de hachage pour s'adapter aux contraintes des accès en écriture.

- *Déléguer les traitements*
  Dans notre solution, le serveur a un rôle limité dans le traitement des données et consiste uniquement à envoyer des segments de données. Son rôle a été réduit au minimum afin de conserver un chiffrement opaque des données. Cependant dans l'approche C-SDA [20][21] développé dans notre équipe, le serveur a un rôle actif car il participe à l'évaluation des parties de la requêtes qui peuvent être exécutées sur les données chiffrées. Cela soulève deux problèmes. Premièrement, en utilisant un schéma de chiffrement ou des index, sur les données chiffrées (Hacigumus et al. [62]), on révèle de l'information. En donnant au serveur la possibilité de faire des traitements, on donne

implicitement aux pirates des armes supplementaires pour gagner accès à de l'information non-autorisée. Deuxièmement, comme le serveur n'est pas de confiance, cela signifie qu'il peut retourner des réponses incorrectes qui devront être vérifiées par le SOE. Des techniques existent pour cela (e.g., arbre de hachage de Merkle[78]) mais elles possèdent un surcoût non-négligable. Par conséquent, nous devrons étudier de nouvelles techniques pour réduire ce surcoût. Une alternative serait d'utiliser un composant sécurisé sur le serveur. Bien qu'il puisse simplifier les deux problèmes précedemment mentionés (dans le cas extrême, il évalue la requête comme dans la solution actuelle et renvoie le résultat au client), il est très possible qu'il devienne un goulot d'étranglement car il devra répondre à une multitude de requêtes des clients. Ainsi, nous devrons fournir de nouvelles techniques pour réduire le volume de données à lire et à traiter par le composant sécurisé. Pour cela, nous pouvons étudier de nouveaux types d'index (pas nécessairement des index sur des flux) qui peuvent être analysés rapidement par le SOE pour demander au serveur de renvoyer un surensemble restreint qui sera ensuite raffiné par le SOE du client.

References

[1] M. Abadi, B. Warinschi, *Security Analysis of Cryptographically Controlled Access to XML Documents*, In Proceedings of the ACM SIGMOD/PODS International Conference on the Principles of Databases Systems, Baltimore, USA, 2005.

[2] A. Abrial, J. Bouvier, M. Renaudin, P. Senn, P. Vivet, *A New Contactless Smart card IC using an On-Chip Antenna and an Asynchronous Micro-controller*, IEEE Journal of Solid-state Circuit, 2001.

[3] S. Akl, P. Taylor, *Cryptographic solution to a problem of access control in a hierarchy*, ACM Transactions on Computer Systems (TOCS). Volume 1 , Issue 3 (August 1983), 239-248, 1983.

[4] S. Amer-Yahia, S. Cho, L. Lakshmanan, D. Srivastava, *Minimization of tree pattern queries*, In Proceedings of the ACM SIGMOD international conference on management of data, Santa Barbara, CA, USA, 2001.

[5] N. Anciaux, *Database Systems on Chip*, PhD thesis, University of Versailles, 2004.

[6] N. Anciaux, C. Bobineau, L. Bouganim, P. Pucheral, P. Valduriez, *PicoDBMS: Validation and Experience*, In Proceedings of the 27th Int. Conf. on Very Large Data Bases (VLDB), demo session, 2001.

[7] Apple iTunes, http://www.apple.com/itunes/.

[8] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, A. Puglies, *Efficient Query Evaluation over Compressed Data*, In Proceedings of the 9th Extending Database Technology (EDBT) international conference, Heraklion, Greece, 2004.

[9] ARM, TrustZone Technology Overview, http://www.arm.com/products/CPUs/arch-trustzone.html.

[10] Axalto e-gate 2004, Worldwide USB smart card developer contest. Second edition, held at CTST, Washington DC, USA, http://www.egateopen.axalto.com.

[11] Axalto SIMagine 2005, Worldwide Mobile Communication and Java Card™ developer contest. Sixth edition, held at 3GSM, Cannes, France,

http://www.simagine.axalto.com.

[12] Axalto. SIMera - Classic SIM Card. http://www.axalto.com/wireless/classic.asp.

[13] R. Bayardo, D. Gruhl, V. Josivofski, J. Myllymaki, *An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing*, In Proceedings of the 13th World Wide Web (WWW) international conference, New York, USA, 2004.

[14] D. Bell, L. La Padula, *Secure Computer Systems: Unified Exposition and Multics interpretation*, MITRE Corporation, Technical report, 1976.

[15] E. Bertino, B. Carminati, E. Ferrari, *A Temporal Key Management Scheme for Secure Broadcasting of XML Documents*, In Proceedings of the ACM Conference on Computer and Communications Security, Washington DC, USA, 2002.

[16] E. Bertino, E. Ferrari, *Secure and Selective Dissemination of XML Documents*, ACM Transactions on Information and System Security, Vol. 5, No. 3, August 2002, Pages 290–331.

[17] E. Bertino, G. Correndo, E. Ferrari, G. Mella, *An Infrastructure for Managing Secure Update Operations on XML Data*, In Proceedings of the 8th ACM Synopsium on Access Control Models And Technologies (SACMAT), Como, Italy, 2003.

[18] E. Bertino, S. Castano, E. Ferrari, G. Mella, *Securing XML Documents with Author-X*, In Proceedings of the IEEE International Conference on Internet Computing, Las Vegas, USA, 2001.

[19] J-C. Birget, X. Zou, G. Noubir, B. Ramamurthy, *Hierarchy-Based Access Control in Distributed Environments*, In Proceedings of the IEEE International Conference on Communication (ICC), Saint Petersbourg, Russia, 2001.

[20] L. Bouganim, P. Pucheral, *Chip-Secured Data Access: Confidential Data on Untrusted Servers*, In Proceedings of the 28th International Conference of Very Large Databases (VLDB), Hong Kong, 2002.

[21] L. Bouganim, F. Dang Ngoc, P. Pucheral, L. Wu, *Chip-Secured Data Access:Reconciling Access Rights with Data Encryption* , In Proceedings of the 29th International Conference of Very Large Databases (VLDB), Berlin, Germany 2003.

[22] L. Bouganim, F. Dang Ngoc, P. Pucheral, *Client-Based Access Control for XML Documents*, In Proceedings of the 30th International Conference of Very Large Databases (VLDB), Toronto, Canada, 2004.

[23] L. Bouganim, F. Dang Ngoc, P. Pucheral, *A Smart XML Access Right Controller for Mobile Applications*, In Proceedings of the 5th International

Conference e-smart, Sophia-Antipolis, France, 2004.

[24] L. Bouganim, C. Cremarenco, F. Dang Ngoc, N. Dieu, P. Pucheral, *Safe Data Sharing and Data Dissemination on Smart Devices*, Demo session, In Proceedings of the ACM SIGMOD International Conference of Management of Data, Baltimore, MD, USA, 2005, http://www-smis.inria.fr/Ecsxa.html.

[25] L. Bouganim, F. Dang Ngoc, P. Pucheral, *Tamper-resistant ubiquitous data management*. International Journal of Computer Systems Science & Engineering (IJCSSE), Volume 20, Number 2, pp. 147-158, 2005.

[26] L. Bouganim, D. Calegari, C. Cremarenco, F. Dang Ngoc, N. Dieu, P. Pucheral, 2004. *Chip-Secured XML Access*. Silver Award of the e-gate 2004 software contest at CTST, Washington DC, USA.

[27] L. Bouganim, C. Cremarenco, F. Dang Ngoc, N. Dieu, P. Pucheral, *MobiDiQ a Fair DRM*, Gold Award, 3GSM 2005, Cannes, France.

[28] P. Buneman, M. Grohe, C. Koch, *Path Queries on Compressed XML*, In Proceedings of the 29th international conference on Very Large Databases (VLDB), Berlin, Germany, 2003.

[29] B. Carminati, E. Ferrari, E. Bertino, *Assuring Security Properties in Third-party Architectures*, Technical Report UNISUBRIA.D.2, 2004.

[30] C. Chan, P. Felber, M. Garofalakis, R. Rastogi, *Efficient Filtering of XML Documents with XPath Expressions*, In Proceedings of the 18th International Conference on Data Engineering (ICDE), San Jose, CA, USA, 2002.

[31] R. Chandramouli, *Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks*, In Proceedings of the 5th ACM workshop on Role-based Access Control, Berlin, Germany, 2000.

[32] T. Chang, G. Hwang, *Using the Extension Function of XSLT and DSL to Secure XML Documents*, In Proceedings of the 18th International Conference on Advanced Information Networking and Applications (AINA), Fukuoka, Japan, 2004.

[33] Y. Chen, G. Mihaila, S. Davidson, S. Padmanahban, *EXPedite: A System for Encoded XML Processing*, In Proceedings of the 13th ACM International Conference on Information and Knowledge Management (CIKM), Washington DC, USA, 2004.

[34] S. Cho, S. Amer-Yahia, L. Lakshmanan, D. Srivastava, *Optimizing the secure evaluation of twig queries*. In Proceedings of the 28th international conference on Very Large Databases (VLDB), Hong Kong, 2002.

[35] Computer Security Institute 2003, *CSI/FBI Computer Crime and Security Survey*, http://www.gocsi.com/forms/fbi/pdf.html.

[36] F.Cuppens, N. Cuppens-Boulahia, T. Sans, *Protection of Relationships in XML Documents with the XML-BB Model*, In Proceedings of the International Conference on Information Systems Security (ICISS), Kolkata, India, 2005.

[37] F. Cuppens, *Modélisation formelle de la sécurité des systèmes d'informations*, Habilitation à Diriger les Recherches, Université Paul Sabatier, Toulouse, France, 2000.

[38] E. Damiani, S. De Capitani, S. Paraboschi, P. Samarati, *Fine Grained Access Control for SOAP E-Services*, In Proceedings of the International Conference World Wide Web (WWW), 2001.

[39] E. Damiani, S. De Capitani, S. Paraboschi, P. Samarati, *Design and Implementation of an Access Control Processor for XML Documents*, Computer Networks, 2000.

[40] E. Damiani, S. De Capitani, S. Paraboschi, P. Samarati, *Securing XML document*, In Proceedings of the International Conference on Extending Database Technology (EDBT), pp 121-135, Konstan, Germany, 2000.

[41] P. Devanbu, M. Gertz, C. Martel, S. Stubblebine, *Authentic Third-party Data Publication*, In Proceedings of the International Conference on Database Security (DBSec), Amsterdam, The Netherlands, 2000.

[42] P. Devanbu, M. Gertz, A. Kwong, C. Martel, G. Nuckolls, S. Stubblebine, *Flexible Authentification of XML Documents*, In Proceedings of the 8th ACM International Conference on Computer and Communication Security, Philadelphia, USA, 2001.

[43] Y. Diao, M. Altinel, M. Franklin, H. Zhang, P. Fischer, *Path Sharing and Predicate Evaluation for High-Performance XML Filtering*, ACM TODS, 2003.

[44] The Digital Millennium Copyright Act (DMCA), http://www.copyright.gov/legislation/dmca.pdf.

[45] Department of Defense Standard, *Trusted Computer System Evaluation Criteria* (TCSEC), DoD 5200.28-STD http://www.radium.ncsc.mil/tpep/library/rainbow/5200.28-STD.html, 1985.

[46] DRM Watch Staff, *DRM Watch: Fnac Stretches Fair Use Rules in New Online Music Service*, http://www.drmwatch.com/ocr/article.php/3412101, 2004.

[47] Electronic Frontier Foundation, *Unintended Consequences: Five Years under the DMCA*, http://www.eff.org/IP/DMCA/.

[48] A. El Kalam, S. Benferthat, A. Miege, R. Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, G. Trouessin, *Organization based access control*, In Proceedings of the 4th IEEE international Workshop on Policies for Distributed Systems and Networks (POLICY). New York, USA, 2003.

[49] EMVCo EMV 4.1, http://www.emvco.com.

[50] European Union Copyright Directive (EUCD), http://www.fipr.org/copyright/eucd.html

[51] EuroSmart, Press Conference, Carte 2004, http://www.eurosmart.com/Update/Download/05-05/2004-Industries-Figures.pdf.

[52] Fair Use legal definition, http://usinfo.state.gov/topical/econ/ipr/ipr-glossary.htm.

[53] W. Fan, C. Chan, M. Garofalakis, *Secure XML Querying with Security Views,* In Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, 2004.

[54] B. Finance, S. Medjdoub, P. Pucheral, *The Case for Access Control on XML Relationships,* In Proceedings of the International Conference on Information and Knowledge Management (CIKM), Bremen, Germany, 2005.

[55] Fnacmusic.com : le test complet sur Ratiatum.com, *Le Peer-to-Peer (P2P) au delà du téléchargement,* http://www.ratiatum.com/p2p.php?id_dossier=1708&page=1. In French.

[56] Frost & Sullivan, http://www.smartcards.frost.com.

[57] I. Fundulaki, M. Marx, *Specifying access control policies for XML documents with XPath*, In Proceedings of the 9th ACM Symposium on Access Control Models and Technologies, New York, USA, 2004.

[58] A. Gabillon, E. Bruno, *Regulating Access to XML documents,* IFIP WG on Database Security, 2001.

[59] A. Gabillon, *A Formal Access Control Model for XML Databases*, In Proceedings of the second VLDB Workshop on Secure Data Management. Trondheim, Norway. September 2005.

[60] C. Geueur-Pollmann, *XML Pool Encryption*, In Proceedings of the ACM Workshop on XML Security, Fairfax *USA, 2002.*

[61] T. Green, A. Gupta, G. Miklau, M. Onizuka, D. Suciu, *Automata and Stream Indexes,* ACM TODS 2004.

[62] H. Hacigumus, B. Iyer, C. Li, S. Mehrotra, *Executing SQL over encrypted data in the database-service-provider model,* ACM SIGMOD 2002.

[63] J. He, M. Wang, *Cryptography and Relational Database Management Systems*, In Proceedings of the International Database Engineering and Application Symposium (IDEAS), Grenoble, France, 2001.

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

[65] IBM, *IBM PCI Cryptographic Coprocessor*, http://www-03.ibm.com/security/cryptocards/pcicc.shtml.

[66] IBM, *IBM XACL*, http://www.trl.ibm.com/projects/xml/xacl/xmlac-proposal.html.

[67] IBM-Harris, *Multinational Consumer Privacy Survey*, http://www.pco.org.hk/english/infocentre/files/westin.doc.

[68] International Federation of Phonographic Industry (IFPI), http://www.ifpi.org/.

[69] International Standardization Organization (ISO), *Integrated Circuit(s) Cards with Contacts - Part 1: Physical characteristics*, ISO/IEC 7816-1, 1987.

[70] International Standardization Organization (ISO), *Integrated Circuit(s) Cards with Contacts - Part 2: Dimensions and location of the contacts*, ISO/IEC 7816-2, 1988.

[71] International Standardization Organization (ISO), *Integrated Circuit(s) Cards with Contacts - Part 7: Interindustry Commands for Structured Card Query Language-SCQL*, ISO/IEC 7816-7, 1999.

[72] M. Kudo, S. Hada, *XML Document Security based on Provisional Authorization*, ACM CCS 2000.

[73] M. Kuhn, *Design Principles for Tamper-Resistant Smartcard Processors*, In Proceedings of the USENIX Workshop on Smartcard Technology, Chicago, 1999.

[74] C. Lim, S. Park, S. Son, *Access Control of XML Documents Considering Update Operations*, ACM Workshop on XML Security, 2003.

[75] B. Luo, D. Lee, W-C. Lee, P. Liu, *QFilter: Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting*, CIKM, 2004.

[76] MasterCard, *MasterCard Open Data Storage (MODS)*, 2002. https://hsm2stl101.mastercard.net/public/login/ebusiness/smart_cards/one_smart_card/biz_opportunity/mods.

[77] A. Menezes, P. Oorschot, S. Vanstone, *Handbook of Applied Cryptography*, CRC Press, 1996.

[78] R. Merkle, *A Certified Digital Signature*, Advances in Cryptology–Crypto, 1989.

[79] mi2g, SIPS & EVEDA, http://www.mi2g.com/cgi/mi2g/press/faq.pdf.

[80] Microsoft, *Windows Microsoft Windows Media*,

http://www.microsoft.com/windows/windowsmedia/.

[81] G. Miklau, D. Suciu, *Controlling Access to Published Data Using Cryptography*, In Proceedings of the 29th International Conference of the Very Large Databases (VLDB), Berlin, Germany, 2003.

[82] G. Miklau, D. Suciu, *Cryptographically Enforced Access Control for XML*, WebDB 2002.

[83] G. Miklau, D. Suciu, *Containment and equivalence of an XPath fragment*, In Proceedings of the ACM SIGMOD/SIGART Symposium on Principles of Database Systems, pp. 65-76, Madison, USA, 2002.

[84] S. Mohan, A. Sengupta, Y. Wu, J. Klinginsmith, *Access Control for XML - A Dynamic Query Rewriting Approach*, In Proceedings of the ACM Fourteenth Conference on Information and Knowledge Management (CIKM), Bremen, Germany, 2005.

[85] M. Murata, A. Tozawa, M. Kudo, *XML Access Control Using Static Analysis*, ACM CCS, Washington, 2003.

[86] W. Ng, B. Ooi, K. Tan, A. Zhou, *Peerdb: A p2p-based system for distributed data sharing*, In Proceedings of the IEEE International Conference on Data Engineering, Bangalore, India, 2003.

[87] NIST, *Advanced Encryption Standard (AES)*, FIPS Publication 197, 2001.

[88] NIST, *Data Encryption Standard (DES)*, FIPS Publication 46, 1977.

[89] NIST, *Role Based Access Control*, http://csrc.nist.gov/rbac/.

[90] NIST, *Security Requirements for Cryptographic modules*, FIPS Publication 140-2, 2002.

[91] OASIS, eXtensible Access Control Markup Language (XACML), http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

[92] D. Olteanu, T. Furche, F. Bry, *An efficient single-pass query evaluator for XML data streams*, In Proceedings of Symposium on Applied Computing (SAC) , Nicosia, Cyprus, 2004.

[93] The Open Digital Rights Language Initiative, http://odrl.net/.

[94] PC/SC Workgroup, http://www.pcscworkgroup.com.

[95] F. Peng, S. Chawathe, *XPath Queries on Streaming Data*, In Proceedings ot he ACM SIGMOD International Conference on Management of Data, San Diego, USA, 2003.

[96] P. Pucheral, L. Bouganim, P. Valduriez, C. Bobineau, *PicoDBMS: Scaling*

down Database Techniques for the Smart Card, Very Large Data Bases Journal (VLDBJ), Vol.10, n°2-3, 2001.

[97] Oracle Corp., *Advanced Security Administrator Guide*, Release 9.2, 2002.

[98] W. Rankl and W. Effing, *Smart Card handbook. Second Edition*, John Wiley, 1998.

[99] I. Ray, I. Ray, *Using Compatible Keys for Secure Multicasting in E-Commerce*, In Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS), Florida, USA, 2002.

[100] I. Ray, I. Ray, N. Narasimhamurthi, *A Cryptographic Solution to Implement Access Control in a Hierarchy and More*, In Proceedings of the 9th ACM Symposium on Access Control Models and Technologies (SACMAT),New York, USA, *2002.*

[101] R. Rivest, A. Shamir, L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*. In the Communications of the ACM (CACM), Volume 21, pp. 120-126, 1978.

[102] SAX Project, Simple API for XML, *http://www.saxproject.org/*.

[103] B. Schneier, *Applied Cryptography* – 2nd edition, John Wiley & Sons, 1996.

[104] SkyDesk: @Backup (Storage Service Provider), http://www.backup.com/index.htm.

[105] Smart Card Alliance, *HIPAA Compliance and Smart Cards: Solutions to Privacy and Security Requirements*, 2003.

[106] Smart Card Alliance, http://www.smartcardalliance.org/industry_news/industry_news_item.cfm?itemID=1364.

[107] J. Sterne, *Cryptography and the French Banking Cards: Past, Present, Future*, in Proceedings of Financial Cryptography, 2004.

[108] A. Stoica, C. Farkas, *Secure XML Views*, DBSec 2002, Cambridge.

[109] P. Tolani, J. Haritsa, XGRIND: *A Query-Friendly XML Compressor*. In Proceedings of the 18th International Conference on Data Engineering (ICDE), San Jose, CA, USA, 2002.

[110] ToXgene - *The ToX XML Data Generator*, http://www.cs.toronto.edu/tox/toxgene/.

[111] Transaction Processing Performance Council, http://www.tpc.org/.

[112] Trusted Computing Group (TCG), http://www.trustedcomputinggroup.org.

[113] J-P Tual, *DRM as new business driver for Smart-Card: potential reality or*

inaccessible dream, In Proceedings of the 5th edition of the International Conference e-smart, Sophia Antipolis, France, 2004.

[114] UW XML Data Repository, www.cs.washington.edu/research/xmldatasets/.

[115] R. Vingralek, *GnatDb: A Small-Footprint, Secure Database System*, In Proceedings of the 28th W3C international conference on Very Large Databases (VLDB), Hong Kong, 2002.

[116] X. Wang, H. Yu, *How to Break MD5 and Other Hash Functions*, In Proceedings of the Conference EUROCRYPT, Aarhus, Denmark, 2005

[117] Y. Wang, K. Tan, *A Scalable XML Access Control System*, WWW 2001.

[118] W3C, *Document Object Model (DOM)*, http://www.w3.org/DOM/.

[119] W3C, *eXtensible Markup Language (XML)*. http://www.w3.org/XML/.

[120] W3C, *PICS: Platform for Internet Content Selection*, http://www.w3.org/PICS.

[121] W3C, *XML Encryption Requirements*, http://www.w3.org/TR/xml-encryption-req.

[122] W3C, *XML Path Language 1.0*, http://www.w3.org/TR/xpath/.

[123] W3C, *XML Schema*, http://www.w3.org/XML/Schema.

[124] W3C, *XSL Transformations (XSLT)*, http://www.w3.org/XML/.

[125] W3C, *XQuery 1.0, an XML Query Language*, http://www.w3.org/TR/xquery/.

[126] M. Witteman, *Advances in Smartcard Security*, Information Security Bulletin, July 2002, Chapter 11.

[127] XrML, *eXtendible rights Markup Language*. http://www.xrml.org.

[128] T. Yu, D. Srivastava, L. Lakshmanan, H. Jagadish, *Compressed Accessibility Map: Efficient Access Control for XML*, VLDB 2002.

Abstract

The erosion of trust put in traditional database servers and in Database Service Providers and the growing interest for different forms of selective data dissemination are different factors that lead to move the access control from servers to clients. Different data encryption and key dissemination schemes have been proposed to serve this purpose. By compiling the access control rules into the encryption process, all these methods suffer from a static way of sharing data. With the emergence of hardware security elements on client devices, more dynamic client-based access control schemes can be devised. This thesis proposes a tamper-resistant client-based XML access right controller supporting flexible and dynamic access control policies. The access control engine is embedded in a hardware secure device and therefore must cope with specific hardware resources. This engine takes benefit from a dedicated index to quickly converge towards the authorized parts of a – potentially streaming – XML document. Additional security mechanisms guarantee that the input document is protected from any form of tampering and replay attacks. Finally, we provide performance measurements and show the viability of our approach on smart cards in various application contexts.

Keywords: XML access control, data confidentiality, ubiquitous data management, smart card

Résumé

La baisse de confiance accordée aux serveurs traditionnels de bases de données et aux hébergeurs de bases de données (Database Service Providers) ainsi que l'intérêt croissant pour la distribution sélective de données sont différents facteurs qui amènent à migrer le contrôle d'accès du serveur vers les terminaux clients. Plusieurs schémas de chiffrement et de dissémination de clés ont été proposés dans ce but mais en compilant les règles de contrôle d'accès dans le processus de chiffrement, ils imposent tous un partage statique des données. L'apparition de composants matériels sécurisés sur divers terminaux clients permet de lever cette limitation en permettant de concevoir de nouveaux schémas dynamiques de contrôle d'accès sur le client. Cette thèse présente un évaluateur sécurisé de politiques de contrôle d'accès flexibles et dynamiques sur des documents XML. Cet évaluateur est embarqué dans un composant matériel sécurisé et doit par conséquent s'adapter à des contraintes matérielles spécifiques. Il tire pour cela parti d'un index dédié pour converger rapidement vers les parties autorisées d'un document arrivant éventuellement en flux sur le terminal client. Des mécanismes supplémentaires de sécurité garantissent que le document d'entrée est protégé contre toute forme de modification illicite et d'attaque de rejeu. La viabilité de notre approche est montrée dans divers contextes applicatifs sur des cartes à puces actuelles et est validée par des mesures de performance.

Mots clés : contrôle d'accès XML, confidentialité des données, gestion ubiquitaire de données, carte à puce