# Dynamic access-control policies on XML encrypted data

Luc Bouganim, François Dang Ngoc, Philippe Pucheral

# Dynamic Access-Control Policies on XML Encrypted Data

LUC BOUGANIM, FRANCOIS DANG NGOC, and PHILIPPE PUCHERAL
INRIA Rocquencourt and PRiSM Laboratory, University of Versailles

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 paper 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 benefits from a dedicated index to quickly converge toward the authorized parts of a potentially streaming XML document. Pending situations (i.e., where data delivery is conditioned by predicates, which apply to values encountered afterward in the document stream) are handled gracefully, skipping, whenever possible the pending elements and reassembling relevant parts when the pending situation is solved. Additional security mechanisms guarantee that (1) the input document is protected from any form of tampering and (2) no forbidden information can be gained by replay attacks on different versions of the XML document and of the access-control rules. Performance measurements on synthetic and real datasets demonstrate the effectiveness of the approach. Finally, the paper reports on two experiments conducted with a prototype running on a secured hardware platform.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection;* H.3.1

Authors' addresses: Luc Bouganim, François Dang Ngoc, and Philippe Pucheral, INRIA Rocquencourt, Domaine de Voluceau, 78153 Le Chesnay Cedex, France; email: {Luc.bouganim}@inria.fr; François Dang Ngoc and Philippe Pucheral, University of Versailles, 45 avenue des Etats-Unis, 78035 Versailles Cedex, France; email: {Philippe.Pucheral}@prism.uvsq.fr.

General Terms: Security, Design

Additional Key Words and Phrases: Access-control, data confidentiality, ubiquitous data management, smartcard

**ACM Reference Format:**

Bouganim, L., Ngoc, F. D., and Pucheral, P. 2008. Dynamic access-control policies on XML encrypted data. ACM Trans. Inform. Syst. Secur. 10, 4, Article 16 (January 2008), 37 pages. DOI = 10.1145/1284680.1284684 http://doi.acm.org/ 10.1145/1284680.1284684

## 1. INTRODUCTION

The access-control management is traditionally performed by the servers–the place where the trust is. This situation, however, is rapidly evolving because of very different factors: the suspicion toward Database Service Providers (DSP) hosting outsourced databases [Hacigumus et al. 2002; Bouganim and Pucheral 2002], the increasing vulnerability of database servers facing external and internal attacks [Computer Security Institute 2003], the emergence of decentralized ways to share and process data resulting from peer-to-peer databases [Ng et al. 2003] or license-based distribution systems [XrML], and the ever-increasing concern of parents and teachers to protect children by controlling and filtering out what they access on the Internet [W3C PICS]. The common consequence of these orthogonal factors is to move access-control from servers to clients. Because of 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 DSP [Hacigumus et al. 2002], database server security [He and Wang 2001], nonprofit and for-profit publishing [Miklau and Suciu 2003, Bertino et al. 2001, Microsoft] and hierarchical access-control [Akl and Taylor. 1983, Birget et al. 2001, Ray et al. 2002]. These models differ in several ways: data access model (pull Versus. push), access-control model (DAC, RBAC, MAC), encryption scheme, key delivery mechanism, and granularity of sharing. However, these models have in common to minimize the trust required of the client at the price of a static way of sharing data. Indeed, whatever the granularity of sharing is, 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 reencryption of the dataset and a potential redistribution of keys.

Unfortunately, there are many situations where access-control rules are user specific, dynamic, and, thereby, 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 (agendas, 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) [El Kalam et al. 2003], may evolve over time (e.g., depending on the patient's treatment) and may be subject to provisional authorizations [Kudo and Hada. 2000]. 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 [Bouganim and Pucheral. 2002]. Regarding parental control, neither Web site nor Internet Service Provider (ISP) 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 [XACML, XrML, ODRL]. 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. Secure tokens and smart cards plugged or embedded into different devices (e.g., PC, PDA, cellular phone, set-top-box) are tamper-resistant hardware solutions exploited in a growing variety of applications (certification, authentication, electronic voting, e-payment, healthcare, etc.) [Henderson et al. 2001]. Secure chips are now at the heart of computing systems such as the TCPA architecture to protect PC platforms against piracy [TCPA] or the SmartRight architecture to enforce digital right management in the rendering devices [SmartRight]. Secure chips are also being integrated in a large diversity of usual objects, making them smarter, to form an ambient and secure intelligence surrounding. Thus, secure operating environments (SOE) have become a reality on client devices [Vingralek 2002]. Hardware-based SOE guarantee a high tamperresistance, 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 paper 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 the access-control management from encryption. The considered input documents are XML documents, the defacto 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 [Bertino et al. 2001, Carminati et al. 2005, Damiani et al. 2002, Gabillon 2004]. Having this context in mind, the problem addressed in this paper can be stated as follows.

## 1.1 Problem Statement

- *To propose an efficient access-control rules evaluator coping with the SOE hardware constraints*
  The SOE being the unique element of trust, the access-control rules evaluator must be embedded within the SOE. First, the limited amount of SOE secured memory precludes any technique based on materialization (e.g., building a DOM [W3C DOM] representation of the document). Second, limited CPU

power and limited communication bandwidth lead to minimization of the amount of data to be downloaded and decrypted in the SOE. Efficiency is, as usual, an important concern.

- *To guarantee that prohibited information is never disclosed*
  The access-control being realized on the client device, no clear-text data but the authorized ones must be made accessible to the untrusted part of this client device.
- *To protect the input document from any form of tampering*
  Under the assumption that the SOE is secure, the only way to mislead the access-control rule evaluator is to tamper the input document, for example, by substituting or modifying encrypted blocks.

## 1.2 Contributions

To tackle this problem, we make the following five contributions:

1. *Accurate streaming access-control rules evaluator*
   We propose a streaming evaluator of XML access-control rules, supporting a robust subset of the XPath language. The choice of a streaming evaluator allows coping with the SOE memory constraint. Streaming is also mandatory to cope with target applications consuming streaming documents. At first glance, one may consider that evaluating a set of XPath-based access-control rules and a set of XPath queries over a streaming document are equivalent problems [Diao and Frauklin 2003, Green et al. 2004, Chan et al. 2002]. However, access-control rules are not independent. They may generate conflicts or become redundant on given parts of the document. The proposed evaluator detects these situations accurately and exploits them to stop, as soon as possible, rules becoming irrelevant.
2. *Skip Index*
   We design a streaming and compact index structure allowing to quickly converge toward 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 cost of communication between the SOE, the client, and the server. This second contribution complements the first to match the performance objective.
3. *Pending predicates management*
   Pending predicates (i.e., a predicate S, conditioning the delivery of a subtree S, but encountered after S in the document) are difficult to manage. We propose a strategy to detect the pending parts of the document, to skip them at parsing time (whenever possible) and to reassemble afterward the relevant pending parts at the right place in the final result.
4. *Integrity checking with random accesses*
   We combine hashing and encryption techniques to make the integrity of the document verifiable despite the forward and backward random accesses generated by the *skip index* and by the support of pending predicates.

5. *Dynamic access-control policy management*
   The dynamicity of access-control policies requires refreshing the access-control rule definitions on the SOE. We propose a solution to ensure the confidentiality and integrity of this refreshing mechanism as well as to guarantee the consistency of the rule updates with respect to the processed document in order to avoid any unauthorized access.

The paper is organized as follows. Section 2 gives an overview of the existing approaches to secure the access-control on XML documents. Section 3 introduces the XML access-control model we consider in this paper, and illustrates it on a motivating example. Sections 4–8 detail the five contributions mentioned above. Section 9 presents experimental results based on both synthetic and real datasets. Section 10 reports on two experiments conducted with a prototype of our approach, developed on a smart-card platform acting as an SOE. Section 11 concludes.

## 2. RELATED APPROACHES

Despite the growing interest for XML encryption, on one side [W3C XMLENC, Chang and Hwang 2004] and XML access-control, on the other side [Abadi and Warinschi 2005; Bertino et al. 2001; Carminati et al. 2005; Damiani et al. 2002; Gabillon 2004; Finance et al. 2005; Kudo and Hada 2000; Miklau and Suciu 2003; XrML; XACML], few works actually combine both issues. As stated in the introduction, there are, as yet, a number of situations where access-control must be performed on client devices and encryption is seen as a prerequisite in this context. This section sketches different approaches to translate access-control policies into encryption schemes and highlight their limits with respect to the problem addressed in this paper. More specific related works are referenced throughout each section of the paper.

### 2.1 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, and (iii), each subject receives the keys needed to decrypt the fragment he/she is granted access to. The Author-X [Bertino et al. 2001] framework is representative of this approach. It considers a publish/subscribe model where encrypted XML documents are pushed toward subscribers. The encryption scheme follows an attribute-wise encryption (i.e., tags, attributes, and values are encrypted in place in the document with an attribute granularity). The decryption keys can be provided to the subscribers in different ways (e.g., through an LDAP directory or within the document itself, encrypted with the public key of each subscriber). The limiting factor in this approach is the number of keys to consider, since this number may grow exponentially with the number of users. The preceding issue (number of keys) can be solved using compatible keys [Ray and Ray 2002], which provide a way for an encrypted data to be decrypted using different keys. However, compatible keys rely on a costly asymmetric encryption, roughly three orders of magnitude slower than symmetric encryption.

## 2.2 Super Encryption

Miklau and Suciu [2003] and Abadi and Warinschi [2005] propose another encryption scheme based on super encryption (i.e., recursive encryption of the same data with different keys). 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. In 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 he/she is primarily granted access to and can keep decrypting the following subparts recursively as long as he/she obtains the proper decryption keys.

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 significant.

## 2.3 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 toward relevant parts of the document with respect to a potential query and/or access-control rules. The idea developed in Carminati et al. [2005] is to delegate part of query evaluation to an insecure server hosting the encrypted data. To this end, attribute-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 Hacigumus et al. [2002], appending to each encrypted value a plaintext index value relating to which interval the value belongs (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 in Devanbu et al. [2001] as a result of a Merkle hash tree [Merkle 1989] built on the queried document. However, Devanbu et al. [2001] considers completeness only for complete subtrees, thereby precluding the use of negative authorizations in an access-control policy. [Carminati et al. 2005] extend the Merkle hash tree toward 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 (i.e., limited memory, communication bandwidth, and CPU power within the SOE), this solution suffers from two weaknesses. 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- 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).

As a conclusion, while these methods offer different interesting features, they do not perform well in the context considered in this paper. Their common weakness is related to the management of dynamic access-control policies. Indeed, subparts of the data need be reencrypted whenever the access-control policies change.

## 3. ACCESS-CONTROL MODEL

### 3.1 Access-Control Model Semantics

Several authorization models have been recently proposed for regulating access to XML documents. Most of these models follow the well-established discretionary access-control (DAC) model [Bertino et al. 2001; Gabillon et al. 2004; Damiani et al. 2002], even though RBAC and MAC models have also been considered [Chandramouli 2000; Cho et al. 2002]. We introduce below a simplified access-control model for XML, inspired by Bertino's [Bertino et al. 2001] and Damiani's model [Damiani et al. 2002] that roughly share the same foundation. Subtleties of these models are ignored for the sake of simplicity.

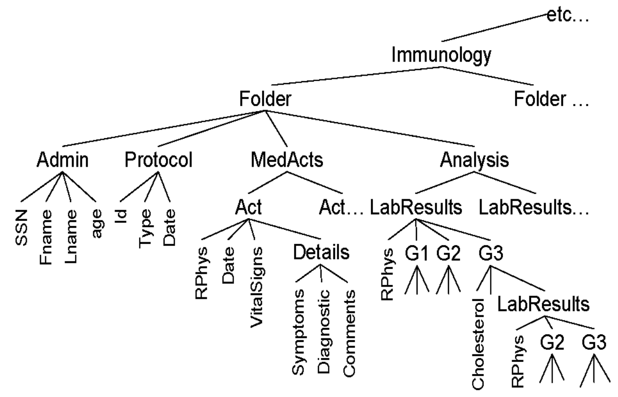
In this simplified model, access-control rules take the form of a three-uple <*sign, subject*, *object*>. *Sign* denotes either a permission (positive rule) or a prohibition (negative rule) for the read operation. *Subject* is a generic term, which can refer either to a user, a group, a role, etc. *Object* 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 paper, we consider a rather robust subset of XPath denoted by $\mathrm{XP}^{\{[],*,//\}}$ [Miklau and Sucui 2002]. This subset, widely used in practice, consists of node tests, the child axis (/), the descendant axis (//), wildcards (*), and predicates or branches [...]. The predicates can further include child and descendant axis, and wildcards, as well as node test or value test (i.e., comparison with a given value). Attributes are handled in the model similarly to elements and are not discussed further.

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. Because of this 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 us 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 also granted [Damiani et al. 2002] (names of denied elements in this path can be replaced by a dummy value [Fan et al. 2004, Gabillon 2004]. This *structural* rule keeps the document structure consistent with respect to the original one. 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-control rules, namely $XP^{\{[],*,//\}}$. Semantically, the result of a query is computed from the authorized view of the queried document (e.g., predicates cannot be expressed on denied elements even if these elements do not appear in the query result). However, access-control rules predicates can apply to any part of the initial document.

## 3.2 Motivating Example

We use an XML document representing medical folders to illustrate the semantics of the access-control model and to serve as motivating example. A sample of this document is pictured in Figure 1, 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 to the age of patients who have subscribed to any protocol test and only to the laboratory results of patients who have subscribed to a protocol test of type G3, provided the measurement for the element *Cholesterol* does not exceed 250 mg/dL.

Medical applications exemplify the need for dynamic access-control rules. For example, a researcher may be granted an exceptional and time-limited access to a fragment of all medical folders where the measurement of *cholesterol* exceeds 300 mg/dL (a rather rare situation). A patient having subscribed to a protocol to test the effectiveness of a new treatment may revoke this protocol at any time because of a degradation of her state of health or for any other personal reasons. Models compiling access control policies in the data encryption cannot tackle this dynamicity. However, the reasons to encrypt the data and delegate the access control to the clients are manifold: exchanging data among medical research teams in a protected peer-to-peer fashion, protect the data from external attacks, as well as from internal attacks. The latter aspect is particularly important in the medical domain because of the very high level



| **Doctor access control policy** |
|---|
| D1: ⊕, //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 |
| R3: ⊖, //G3[Cholesterol > 250] |
| Rules 2 & 3 occur for each of the 10 groups {G1..G10} |
| **Secretary access control policy** |
| S1: ⊕, //Admin |

Fig. 1. Hospital XML document.

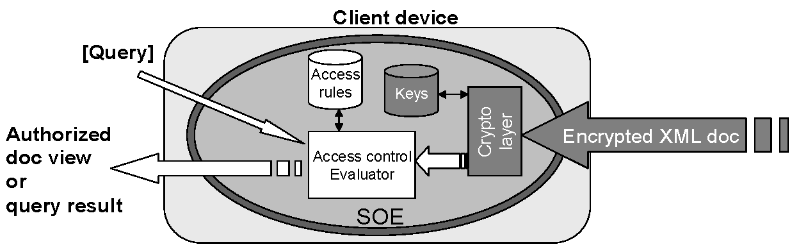


Fig. 2. Abstract target architecture.

of confidentiality attached to the data and to the very high level of decentralization of the information system (e.g., small clinics and general practitioners are prompted to subcontract the management of their information system).

## 3.3 Target Architectures

Figure 2 illustrates an abstract representation of the target architecture for the motivating example, as well as for the applications mentioned in the introduction. As the access-control rules are evaluated on the client, the client device has to be made tamper resistant by creating a secure operating environment (SOE). As stated in the introduction, SOE usually rely on secure chips (e.g., smartcard, secure token, TCPA chip). In the sequel of this paper, and up to the performance evaluation section, 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 and refreshed or downloaded from different sources (including untrusted servers—see Section 7).

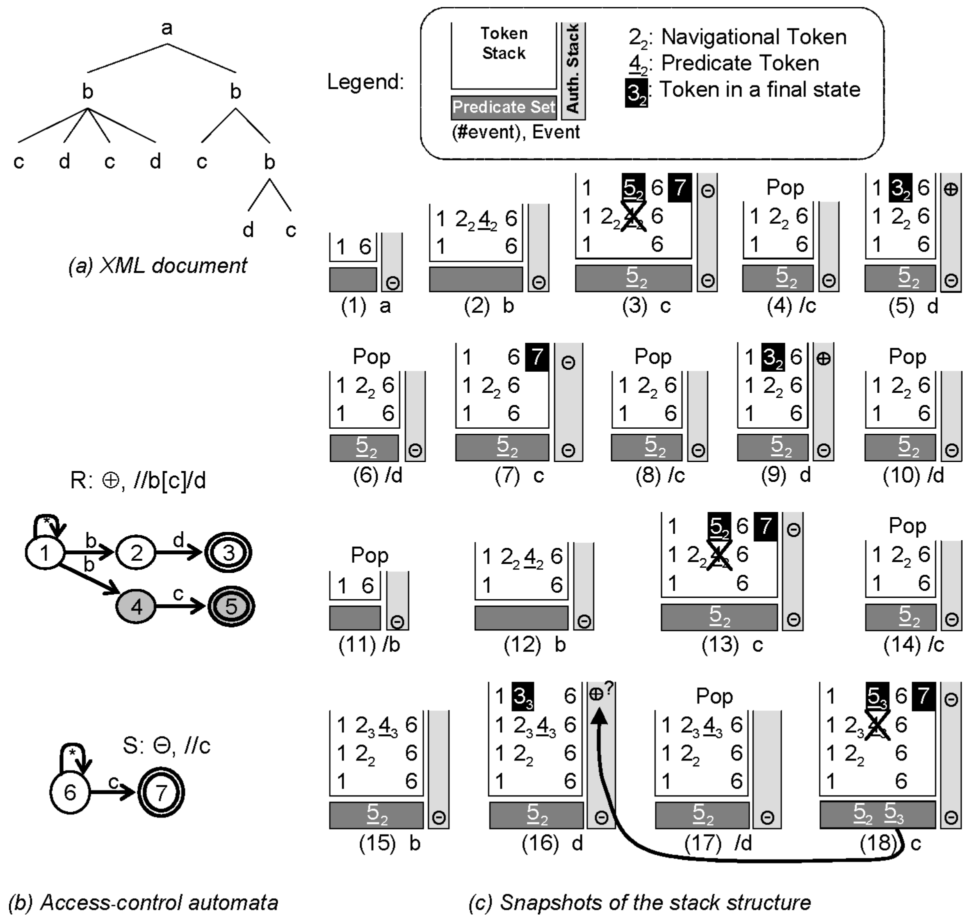
## 4. STREAMING THE ACCESS-CONTROL

While several access-control models for XML have been recently proposed, 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 [Diao and Franklin 2003; Green et al. 2004; Chan et al. 2002]. 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 number of queries. One of the first works addressing the precise evaluation of complex XPath expressions over streaming documents is by Peng and Chawathe [2003], which proposes 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 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 lazily evaluated because of 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.

### 4.1 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 [SAX]) raising *open*, *value*,



(a) XML document
R: ⊕, //b[c]/d
S: ⊖, //c
(b) Access-control automata
Legend:
Token Stack
Auth. Stack
Predicate Set
(#event), Event
$2_2$: Navigational Token
$\underline{4}_2$: Predicate Token
$3_2$: Token in a final state
(c) Snapshots of the stack structure
Pop

Fig. 3. Execution snapshot.

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 nondeterministic finite automaton (NFA) [Hopcroft and Ullman 1979]. Figure 3b shows the access-control rules Automata (*ARA*) corresponding to two rather simple access-control rules expressed on an abstract XML document. This abstract example, used in place of the motivating example introduced in Section 3, 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 the paper). The navigation progress in all *ARA* is memorized as a result of a unique stack-based data structure called *token stack*. The top of the stack contains all active navigational and predicate tokens, i.e., tokens that can trigger a new transition at the next incoming event. 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 us 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 navigational 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 navigational 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 created, materializing their participation in the same *rule instance*.[2] Consequently, a token (proxy) must hold the following information: RuleId (denoted by R, S,

[1]The complexity of this problem has been highlighted in Peng and Chawathe [2003].

[2]To illustrate this, let us consider the rule $R$ and the right subtree of the document presented in Figure 3. 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 token proxies labeled with the same depth 3.

. . . ), 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) denote 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 predicate tokens having reached the final state of a predicate path. A predicate 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.

Stack-based data structures are well adapted to the traversal of a hierarchical document. However, we need a direct access to any stack level to update pending information and to allow some optimizations detailed below. Figure 3c represents an execution snapshot based on these data structures. This snapshot is almost self-explanatory. We thus detail only a small subset of steps.

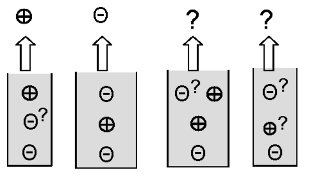
- *Step* 2: the *open* event *b* generates two tokens $Rn2_2$and $Rp4_2$, participating in the same rule instance.
- *Step* 3: the *ARA* of the negative rule S reaches its final state and an active instance of S is pushed in the *authorization stack*. The current authorization remains negative. Token $Rp5_2$ 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 11). 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 13 (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 thoroughly studied in Section 6.

## 4.2 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

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

```
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*

Fig. 4. Conflict-resolution algorithm.

presented in Figure 4 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 6. 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 also considered. Queries are expressed in XPath and are translated in a nondeterministic 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.

## 4.3 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 $\subseteq$ the containment relation between rules $R, S \ldots T$. 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 condition does not eliminate all irrelevant rules. 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

```
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 '?'
```

Fig. 5. Decision on a complete subtree.

*/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, $XP^{\{[],//,*\}}$ [Miklau and Suciu 2002]. This issue could be further investigated since more favorable results have been found for subclasses of $XP^{\{[],//,*\}}$ [Amer-Yahia et al. 2001], but this work is outside the scope of this paper.

A second form of optimization is to dynamically suspend 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 3c. The second optimization is to dynamically evaluate the containment relation between active and pending rules and to benefit from 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 and the communication costs. Therefore, the real challenge is being able to make 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 made 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 5. In this algorithm, *AS* denotes the *authorization stack*, *TS* the *token stack*, *TS[i].NT (resp. TS[i].PT)* the set of navigational (resp. predicate) 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 navigational tokens and the main expected benefit is to skip the complete subtree if this decision is $\ominus$. Note, however, that only navigational tokens are removed from the stack at line 4. The reason for this is that active predicate 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 $\ominus$, (2) the *DecideSubtree* algorithm decides $\ominus$, and (3) there are no predicate tokens 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. 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.

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

## 5. SKIP INDEX

This section introduces a new form of indexation structure, called *skip index*, designed to detect and skip the unauthorized fragments (w.r.t. an access-control policy) and the irrelevant fragments (w.r.t. 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 [Chen et al. 2004, Bayardo et al. 2004] and XML routing [Green et al. 2004], the authors devised a streaming index, which consists of appending its size to each subtree, allowing the possibility to skip it when no other 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 requirement is the ability to keep the desired index encrypted outside of the SOE to guarantee the absence of information disclosure. The second requirement (related to the first and to the SOE storage capacity) is the ability for the SOE to 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 [Arion et al. 2004] or XML skeleton [Buneman et al. 2003] could be considered as a 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.

## 5.1 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 4.3 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 tag nestings, it turns out to be very effective 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 7a see later 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 because of a dictionary of tags [Arion et al. 2004; Tolani and Harista 2002].[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 $\text{DescTag}_{\text{parent(e)}}$ 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 *DescTag()* function recursive. Since the number of elements 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.
- *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 further compress the encoding of tags themselves. Using this scheme, $Log_2(DescTag_{\text{parent(e)}})$ 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

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

```
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
```

Fig. 6. Skipping decision.

size to $\log_2(\mathit{SubtreeSize}_{\mathrm{parent(e)}})$ 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*, $\mathrm{DescTag}_{\mathrm{parent(e)}}$ and $\mathrm{SubtreeSize}_{\mathrm{parent(e)}}$ are retrieved from this stack and used to decode, in turn, $\mathit{TagArray}_e$, $\mathit{SubtreeSize}_e$ 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* ancestor and 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.

## 5.2 Skip Index Usage

As stated earlier, 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 $\mathit{RemainingLabels(t)} \not\subset \mathit{DescTag}_e$. Note that since the *skip index* does not capture the element tags nesting, some rules that cannot apply may not be detected and their evaluation is carried out needlessly. 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.

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 6 decides whether the skip is possible or not. Note 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

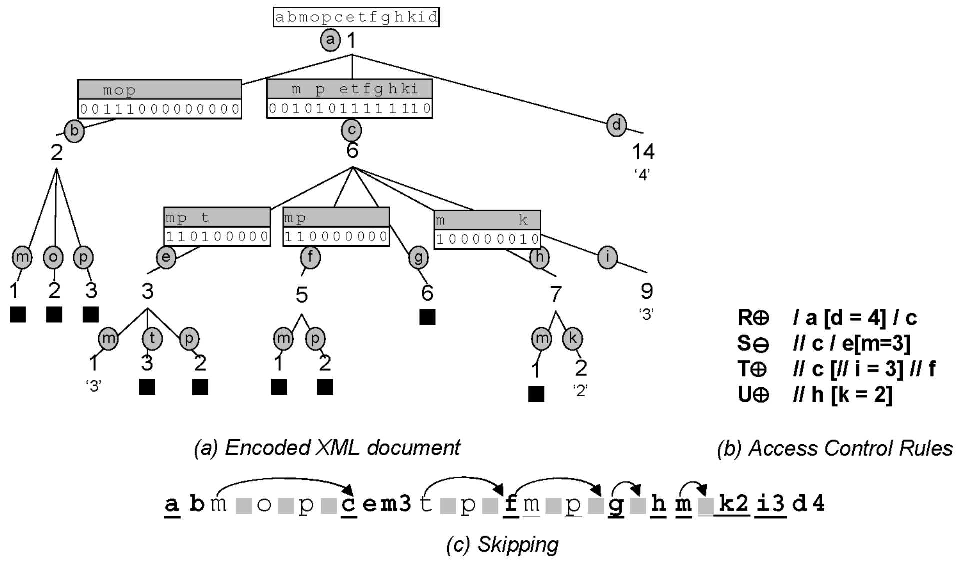


Fig. 7. Skip Index example.

*DecideSubtree*, and, finally, *SkipSubtree* with the benefit of being able to skip a bigger subtree at the next step.

Figure 7 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). At the time element *b* (leftmost subtree) is reached, all the active rules are stopped because of $TagArray_b$ and the complete subtree can be skipped (the decision is ⊖ because of the closed access-control policy). When element *c* is reached, Rule R becomes pending. However, the analysis of the subtree continues, since $TagArray_c$ does not allow more filtering. When element *e* is reached, $TagArray_e$ 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 illustrate 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 7c.

## 6. 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]* leads invariably to a pending situation. Any rule of the form /../*e[P]*/../ also generates 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. We then, provide a solution to cope with multiple pending predicates.

## 6.1 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). We then, describe a more accurate solution adapted to contexts where backward and forward accesses are allowed in the document (e.g., pull-based access, VCR).

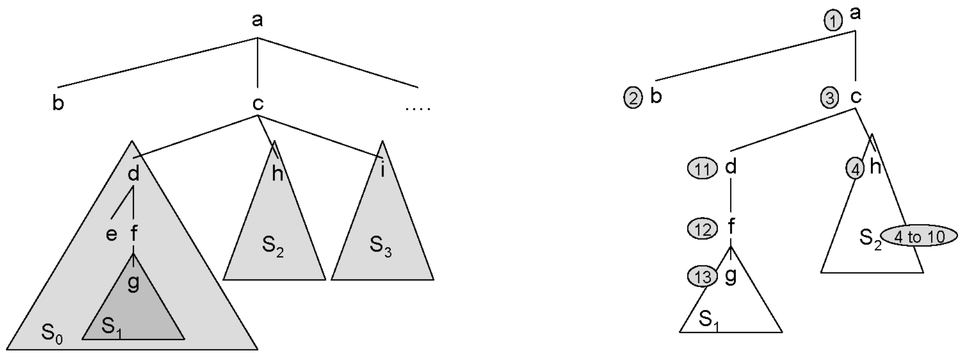
6.1.1 *Strict Streaming.* When receiving the document in a strict streaming fashion, the outcome of pending predicates cannot be known beforehand. Pending parts cannot be buffered inside the SOE considering the assumption made on its storage capacity. To tackle this problem, pending parts are externalized 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 otherwise discarded. 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 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 with the tag of Bi's pathlist.[6]

6.1.2 *Backward/Forward Access.* In the case of backward/forward access, we make the assumption that the pending parts can be read back (e.g., from the

[5]An alternative is to tag forbidden ancestors with a dummy value to comply with [Gabillon 2004] and Fan et al. [2004].

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



(a) Initial document with pending subtrees (b) Delivered document (with anchors)

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

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

(c) Snapshots of the pending list

Fig. 8. Pending predicate management.

server) when pending predicates are solved. The objective pursued is, therefore, to detect pending subtrees and to leave them aside as a result of the *skip index* until the pending situation is solved. The goal is to never read and analyze the same 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 8 illustrates a pending situation involving four subtrees $S_0$, $S_1$, $S_2$, and $S_3$, respectively, associated with pending predicates $P_0$, $P_1$, $P_2$, and $P_3$). *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 us 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 left-most child of $e$. No anchor needs to be memorized for pending right siblings and embedded subtrees of a pending subtree $e'$ (in Figure 8: $N_{S0} = -3$, 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:

$\prec$ denotes the precedence relation in the Pending List
A child_of B $\Leftrightarrow$ B$\prec$A $\wedge$ Level$_A$ = Level$_B + 1 \wedge \neg$ ($\exists$ C / B$\prec$C$\prec$A $\wedge$ Level$_C$ = Level$_B$)A right_sibling_of B $\Leftrightarrow$ B$\prec$A $\wedge$ Level$_A$ = Level$_B \wedge \neg$ ($\exists$ C / B$\prec$C$\prec$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 us 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 us 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 8).
- *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*$'$ becomes the anchor of the potential pending right sibling of $e'$ (e.g., $S_3$ anchor is updated when $S_2$ is delivered). Respectively, –*Ne*$'$ becomes the anchor of the potential pending right sibling of $e'$. 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).

Figure 8 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.

## 6.2 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

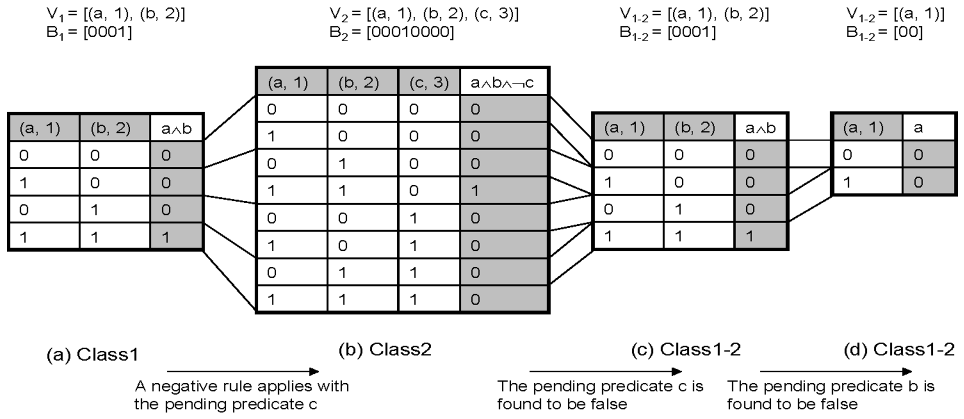

V1 = [(a, 1), (b, 2)]
B1 = [0001]
V2 = [(a, 1), (b, 2), (c, 3)]
B2 = [00010000]
V1-2 = [(a, 1), (b, 2)]
B1-2 = [0001]
V1-2 = [(a, 1)]
B1-2 = [00]
(a) Class1
(b) Class2
(c) Class1-2
(d) Class1-2
A negative rule applies with the pending predicate c
The pending predicate c is found to be false
The pending predicate b is found to be false

Fig. 9. Multiple pending predicate management.

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 9, 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 9 illustrates how the logical expressions are incrementally built and evaluated. Let us 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 because of a bitwise AND NOT between $B'$ and $B_c$ (Figure 9b). Let us 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 9c), that is all the rows (the first row being numbered

0) in the intervals $[k * 2^{index(c)} + 2^{index(c)-1}, (k+1)*2^{index(c)}]$, $k$ varying from 0 to $2^{(|V|-index(c))} - 1$, where $|V|$ is the cardinality of $V$. The resulting truth table of Class2 being the same as the one of C*lass*1, the two classes are merged into the *Class*1–2. Let us assume predicate $b$ is evaluated to false, the table is shrunk as pictured in Figure 9d. 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.

## 7. INTEGRITY CHECKING WITH RANDOM ACCESSES

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 medical folder $X$ may try to extract unauthorized information from a medical folder $Y$. Let us assume that the document is encrypted with a classic block cipher algorithm (e.g., triple-DES) and that blocks are encrypted independently (e.g., following the ECB mode [Menezes et al. 1996; Schneier 1996]), 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. In addition, 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 block chaining (*CBC*) mode in place of *ECB*, meaning that the encryption of a block depends on the preceding block [Menezes et al. 1996; Schneier 1996]. 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 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 and is not known, even by the SOE owner.

Encryption alone is not sufficient to guarantee the document integrity, since the attacker can perform random modifications and substitutions in the ciphertext. Our goal is to provide a solution coping with the forward and backward random accesses to the document, minimizing the quantity of data transmitted to the SOE and the number of cryptographic primitives calls.

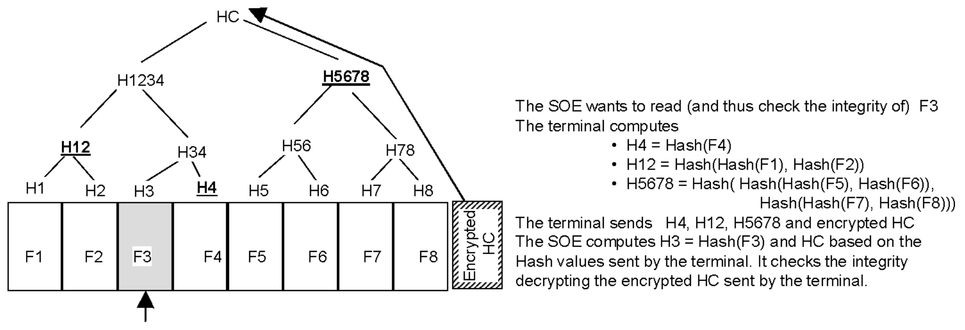
A basic solution would be to split the encrypted document into chunks (e.g., 2 KB) and use a collision resistant hash function (e.g., SHA-1) applied to the encrypted data to compute a digest of each chunk, called *ChunkDigest* that prevents any tampering to occur without impacting this digest. Each chunk contains an identifier reflecting its position in the document, so that block substitutions can be easily detected. Remark that *ChunkDigest* must be encrypted to prevent the terminal to compute by itself a new digest corresponding to tampered data. When accessing $n$ bytes[7] within a chunk, this basic solution would incur to communicate *sizeof(ChunkDigest)* + *sizeof(Chunk)* bytes to the SOE and to decrypt *sizeof(ChunkDigest)* + $n$ bytes in the SOE.

A more efficient solution can be devised and requires the cooperation of the untrusted terminal. Letus consider an iterated hash function based on the Merkle–Damgård construction, which uses a collision-resistant compression function $c$working on $r$ bytes blocks (e.g., with SHA-1, $r = 64$). When the SOE accesses $n$ bytes at position *pos*[8] in a chunk, the idea is (1) to start the computation on the untrusted terminal using function $c$ on the $pos - 1$ first bytes of the chunk; (2) to transmit the intermediate result to the SOE; (3) to complete the hash computation on the SOE; and (4) to check the integrity of the received data by comparing the final hashed value to *ChunkDigest*. 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.

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 Merkle [1989] 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

[7]For simplicity, we assume that $n$ is a multiple of the encryption block size (e.g., eight bytes for 3-DES) and is less than the chunk size. Removing the first assumption incurs an additional transmission of $n$ mod *encryption_block_size* bytes to the SOE, while removing the second assumption is straightforward.

[8]For simplicity, we assume that *pos* is aligned with $r$ (i.e., $pos \bmod r = 0$). Removing this assumption incurs, in the worst case, an additional transmission of $r - 1$ bytes to the SOE.



The SOE wants to read (and thus check the integrity of) F3
The terminal computes
- H4 = Hash(F4)
- H12 = Hash(Hash(F1), Hash(F2))
- H5678 = Hash( Hash(Hash(F5), Hash(F6)), Hash(Hash(F7), Hash(F8)))

The terminal sends H4, H12, H5678 and encrypted HC
The SOE computes H3 = Hash(F3) and HC based on the Hash values sent by the terminal. It checks the integrity decrypting the encrypted HC sent by the terminal.

Fig. 10. Integrity checking with random accesses.

at position *pos* in a fragment $f$ of a given chunk,[9] 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 computation using function $c$on 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*. As a result of this information, the SOE can recompute the root of the Merkle hash tree and compare it to *ChunkDigest*, as shown in Figure 10.

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 Hacigumus et al. [2002] and Bouganim and Pucheral [2002], we make no assumption about any particular way of encrypting data that could facilitate the query execution at the price of a weaker robustness against cryptanalysis attacks.

## 8. DYNAMIC ACCESS-CONTROL POLICY 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. Access-control rule updates may be done proactively, requiring updates systematically before accessing the document, reactively when rule updates are detected, or even be disseminated jointly with the data. The protocols may vary depending on the application scenarios (consider, for instance, two separate transmission channels for the data and for the access control rules updates as in a digital right management (DRM) architecture). The update protocol must ensure three complementary properties independently of the way it is actually implemented: (i) *confidentiality* since access-control rules definition may disclose unauthorized information; (ii) *integrity* since rule modification (even done randomly on an encrypted rule) may mislead the rule processor (cf. Section 7); (iii) *consistency,* meaning that the set of access-control rules stored on the SOE must be up-to-date with respect to the currently processed document.

[9]We used the same assumption as above. Moreover, we assume that $n$ is smaller than a fragment. Handling the case when $n$ is greater than a fragment is straightforward.

Access rules confidentiality and integrity are enforced because of encryption and hashing mechanisms presented in Section 7. Ensuring consistency is more difficult. Inconsistency between the set of access-control rules and the document may appear as a result of a malicious user who may, for instance, filter the update flow, replay the document or the update flow, etc. In any case, the SOE must detect it and must not deliver any data.

We should remark that 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. Inconsistencies appears when a new access-control policy is applied on an old document, thus potentially revealing unauthorized outdated data, or, conversely, when an outdated access-control policy is applied on a recent version of the document, thus revealing unauthorized current data.

Both problems may be solved using a cross-reference versioning between the data and the access-control rules. The access-control policy of every user[10] is stored in a form of a set of rules, each represented by a tuple with four attributes (*rule_ts, rule, doc_ts, sig*), where *rule_ts* is a timestamp incremented for every new access-control rule, *rule* is the access-control rule definition in its encrypted form, *doc_ts* 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 *doc_ts* as well as the last access-control rule timestamp, *rule_ts*, for each user at the time the document was updated.

When the SOE receives an access-control policy updates, it gets all the access-control rules of the associated user having a *rule_ts* greater than the one got from the last connection. It then checks that the access-control rules are properly chained as a result of their *rule_ts* timestamp to ensure that no rules are missing. Let $R$ be the last downloaded access-control rule. When the SOE gets document $D$, it checks that *D.doc_ts* is greater or equal than *R.doc_ts* (if this condition does not hold, this means that a tampering attack tries to apply a new access-control policy on an old document). Similarly, the SOE checks that *R.rule_ts* is greater or equal to *D.rule_ts* (otherwise, it means that a tampering attack tries to apply an outdated access-control policy on a recent document).

## 9. EXPERIMENTAL RESULTS

This section presents experimental results obtained from both synthetic and real datasets. We first give details about the experimentation platform. We then analyze the storage overhead incurred by the *skip index* and compare it with possible variants. We next, study the performance of access-control management, query evaluation, and integrity checking. Finally, the global performance of the proposed solution is assessed on four datasets that exhibit different characteristics.

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

Table I. Document Characteristics

| | WSU | Sigmod | Treebank | Hospital |
|---|---|---|---|---|
| Size | 1.3 MB | 350 KB | 59 MB | 3.6 MB |
| Text size | 210 KB | 146 KB | 33 MB | 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 |

## 9.1 Experimentation Platform

The abstract target architecture presented in Section 3 can be instantiated in many different ways. In this experimentation, we consider that the SOE is embedded in an advanced smartcard platform. While existing smartcards are already powerful (32-bit CPU running at 30 Mhz, 4 KB of RAM, 128 KB of EEPROM), they are still too limited to support our architecture, especially in terms of communication bandwidth (9.6 Kbps). Our industrial partner, Axalto announces by the end of this year a more powerful smartcard equipped with a 32-bit CPU running at 40 Mhz, 8 KB of RAM, 1 MB of flash, and supporting an USB protocol at 1 MBps. Axalto provided us with a hardware cycle-accurate simulator for this forthcoming smartcard. 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.

As this section will make clear, our solution is strongly bounded by the decryption and the communication costs. We considered a smartcard communication bandwidth of 0.5 MB/s which corresponds to a worst case, where each data entering the SOE takes part in the result. We also measure the encryption/decryption bandwidth using the 3DES algorithm, hardwired in the smartcard and obtained 0.15 MB/s.

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 [UW XML]. In addition, we generate a synthetic content for the Hospital document depicted in Section 3 (real datasets are very difficult to obtain in this area), because of the ToXgene generator [ToXgene]. The characteristics of interest of these documents are summarized in Table I.

## 9.2 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 noncompressed document. TC is a rather classic tag compression method and will serve as

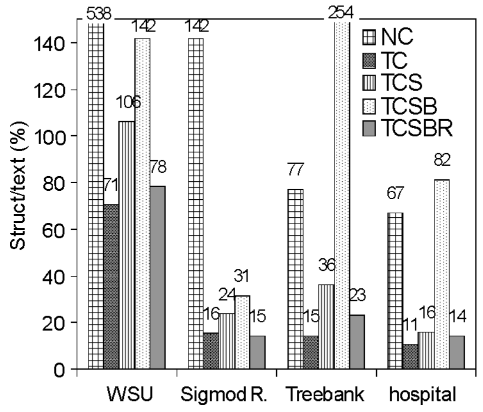


Fig. 11. Index storage overhead.

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 *#distinct tags* bits for each element. Finally, TCSBR is the recursive variant of TCSB and corresponds actually to the *skip index* detailed in Section 5. In all these methods, the metadata need be aligned on a byte frontier. Figure 11 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.

## 9.3 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 12 shows the execution time required to evaluate the authorized view of the three

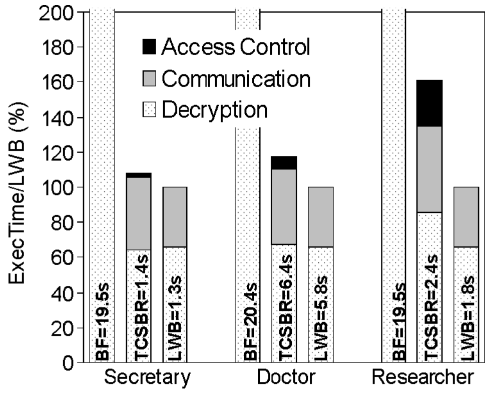


Fig. 12. Access-control overhead.

profiles (Secretary, Doctor, and Researcher) introduced in Section 3 on the Hospital document. Integrity checking is not taken into account here. The size of the compressed document is 2.5 MB and the evaluation of the authorized view returns 135 KB for the Secretary, 575 KB 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 s) 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 in Section 3.

The conclusions that can be drawn from this figure are threefold. First, the BF strategy exhibits very poor performance, explained by the fact that the smartcard 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 (recall that LWB is a theoretical and unreachable lower bound), exemplifying the importance of minimizing the input flow entering the SOE. The largest 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-control rule evaluator will continue—needlessly in the current case—to look at another instance of this predicate (see Section 6). 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.

## 9.4 Impact of Queries

To accurately measure the impact of a query in the global performance, we consider the query //Folder[//Age>v] (v allows us to vary the query selectivity),

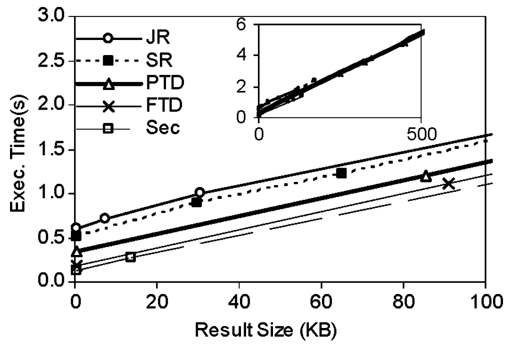


Fig. 13. Impact of queries.

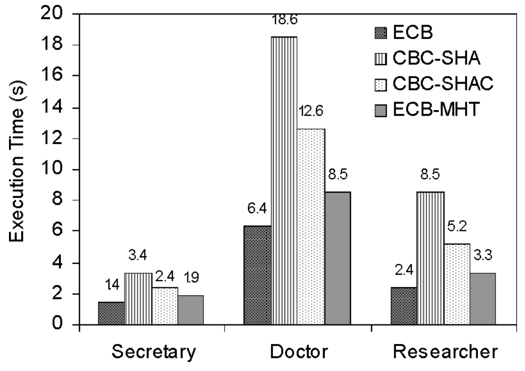


Fig. 14. Impact of integrity control.

executed over five different views built from the preceding profiles and corresponding to: a secretary (S), a part-time doctor (PTD) in charge of a few patients, a full-time doctor (FTD) in charge of 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 13 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 to 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.

## 9.5 Evaluation of the Integrity Control

Figure 14 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 12 shows that the cost ascribed to integrity checking remains quite acceptable when using the technique proposed in Section 8 (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,

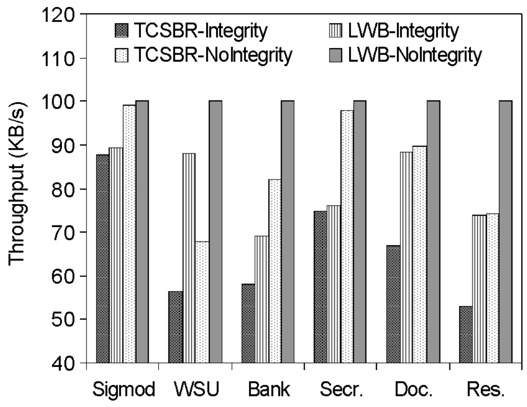


Fig. 15. Performance on real datasets.

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 Figure14 are self-explanatory.

### 9.6 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-control rules (including // and predicates). Each document exhibits interesting characteristics. The Sigmod document is well-structured, nonrecursive, of medium depth, and the generated access-control policy was simple and not very 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). 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 (eight rules). Figure 15 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 55 to 85 KBps, depending on the document and the access-control policy. These preliminary results are encouraging when compared with current available xDSL Internet bandwidth (ranging from 16 to 128 KBps).

## 10. APPLICATIONS AND EXPERIMENTS

Our C prototype running on a hardware cycle-accurate simulator has been developed to forecast the performance of client-based access-control management on future smartcard platforms and to assess the medium-term viability of the approach. Besides this objective, our team developed a second prototype in JavaCard 2.2.1 running on a real Axalto's smartcard platform (SIMera [Axalto]). The objective of this second prototype is to demonstrate the versatility of the approach from the application viewpoint. Toward this goal, two scenarios have been selected, exhibiting two rather different application's profiles regarding the way the information is accessed (pull versus push), the type of this information (textual versus video), and the response time requirements (user patience/real time). We report on these experiments below.

The first application scenario deals with collaborative works. A community of users desires organizing a confidential data sharing space via an untrusted DSP to exchange textual XML data like agendas, address books, profiles, working drafts, etc. Our technology permits to easily define powerful access-control rules on sensitive data (e.g., specific appointments in an agenda or financial section in a working document) while handling rule dynamicity (new partners join or leave the community and relationships among them evolve). This experiment has been the recipient of the silver award of the *e-gate open 2004* smartcard international software contest [Axalto E-Gate 2004].

The second scenario demonstrates a selective dissemination of multimedia streams through unsecured channels. Videos are encoded using the MPEG7 standard, which allows storing short descriptions of the scenes in the XML metadata. The requirements of parental control and advanced digital right-management models are matched as a result of specific and dynamic access-control rules. To tackle the current smartcard low communication bandwidth, we had to trade security for performance. Hence, metadata and video streams are separated, only the former actually traversing the smartcard. This experiment has been conducted on cell phones equipped with new-generation SIM cards. It has been rewarded by the gold award of the *SIMagine'05* smartcard international software contest [Axalto SIMagine 2005].

Finally, the internals of the solution have been demonstrated at the Sigmod'05 conference with a particular focus on the nondeterministic automata engine and the *skip index* structure. Elements of this demonstration are available on-line [Bouganim et al. 2005].

## 11. CONCLUSION

Important factors currently motivate the delegation of access control to client devices. 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. Our objective is to take advantage of new elements of trust on client devices to propose a client-based access-control manager capable of evaluating dynamic access-control rules on a ciphered XML document.

To achieve this goal, we made five complementary contributions. First, we proposed a streaming evaluator of access-control rules supporting a rather robust fragment of the XPath language. To the best of our knowledge, this is the first approach dealing with XML access-control in a streaming fashion. Second, we designed a streaming index structure allowing skipping the irrelevant parts of the input document, with respect to the access-control policy and to a potential query. This index is essential to circumvent the inherent bottlenecks of the target architecture, namely, the decryption and the communication costs. While more complex indexing methods could be devised, the proposed *skip index* combines simplicity (an important feature for embedded software) and performance close to the optimal (both in terms of size and efficiency). Third, we proposed a graceful management of pending predicates compatible with a streaming delivery of the authorized parts of the document. Fourth, we proposed a secure mechanism to refresh the SOE access-control rules from a potentially malicious server, which can use replay attacks to gain access to forbidden data. Fifth, we proposed a combination of hashing and encryption techniques to make the integrity of the document verifiable despite the forward and backward random accesses generated by the preceding contributions.

Our experimental results have been obtained from a C prototype running on a hardware cycle-accurate smartcard simulator provided by Axalto. The global throughput measured is around 70 KBps 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. A JavaCard prototype has been developed on a real Axalto's smartcard platform to demonstrate the versatility of the solution from the application's viewpoint. This second prototype has been the recipient of two well-known smartcard international contests.

This work demonstrates that client-based security solutions give rise to interesting research perspectives and may have a large impact on a growing scale of applications. Among potential research perspectives, we can mention: secured data centric computational models relying on a small source of tamper-resistant storage and computing power, cryptographic methods making integrity violations tamper-evident in a database context (specific access pattern and granularity), indexation techniques for encrypted data, and administration of access-control policies in a large decentralized environment.

## ACKNOWLEDGMENTS

The authors wish to thank Nicolas Dieu and Cosmin Cremarenco for their contribution to the prototypes implementation. Special thanks to Dennis Shasha for his helpful comments on the document and his careful proofreading.

## REFERENCES

Abadi, M. and Warinschi, B. 2005. Security analysis of cryptographically controlled access to XML documents. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*, Baltimore, MD.

Akl, S. and Taylor, P. 1983. Cryptographic solution to a problem of access-control in a hierarchy. *ACM Transactions on Computer Systems (TOCS) 1*, 3 (Aug.), 239–248.

Amer-Yahia, S., Cho, S., Lakshmanan, L., and Srivastava, D. 2001. Minimization of tree pattern queries. *In Proceedings of the ACM SIGMOD International Conference on Management of Data*. Santa Barbara, CA.

Arion, A., Bonifati, A., Costa, G., D'Aguanno, S., Manolescu, I., and Puglies, A. 2004. Efficient query evaluation over compressed data. In *Proceedings of the 9th Extending Database Technology (EDBT) International Conference*. Heraklion, Greece.

AXALTO E-GATE. 2004. Worldwide USB smartcard developer contest. 2nd ed. held at CTST, Washington, DC. http://www.egateopen.axalto.com.

AXALTO SIMAGINE 2005. Worldwide Mobile Communication and Java Card$^{TM}$ developer contest. 6th ed. held at 3GSM, Cannes, France. http://www.simagine.axalto.com.

AXALTO. SIMera—Classic SIM Card. http://www.axalto.com/wireless/classic.asp.

BAYARDO, R., GRUHL, D., JOSIFOVSKI, V., AND MYLLYMAKI, J. 2004. 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.

BERTINO, E., CASTANO, S., AND FERRARI, E. 2001. Securing XML documents with Author-X. In *Proceedings of the IEEE International Conference on Internet Computing*.

BIRGET, J.-C., ZOU, X., NOUBIR, G., AND RAMAMURTHY, B. 2001. Hierarchy-based access-control in distributed environments. In *Proceedings of the IEEE International Conference on Communication (ICC)*, Saint Petersbourg, Russia.

BOUGANIM, B. AND PUCHERAL, P. 2002. Chip-secured data access: Confidential data on untrusted servers. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, Hong Kong.

BOUGANIM, L., CREMARENCO, C., DANG NGOC, F., DIEU, N., AND PUCHERAL, P. 2005. Safe data sharing and data dissemination on smart devices. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. http://www-smis.inria.fr/Ecsxa.html.

BUNEMAN, P., GROHE, M., AND KOCH, C. 2003. Path queries on compressed XML. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB)*, Berlin, Germany.

CARMINATI, B., FERRARI, E., AND BERTINO, E. 2005. Securing XML data in third-party distribution systems. IN *Proceedings of the 14th IEEE International Conference on Information and Knowledge Management (CIKM)*, Bremen, Germany.

CHAN, C., FELBER, P., GAROFALAKIS, M., AND RASTOGI, R. 2002. Efficient filtering of XML documents with XPath expressions. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, San Jose, CA.

CHANDRAMOULI, R. 2000. Application of XML tools for enterprise-wide RBAC implementation tasks. In *Proceedings of the 5th ACM Workshop on Role-Based Access-Control*, Berlin, Germany.

CHANG, T. AND HWANG, G. 2004. 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.

CHEN, Y., MIHAILA, G., DAVIDSON, S., AND PADMANAHBAN, S. 2004. EXPedite: A system for encoded XML processing. In *Proceedings of the 13th ACM International Conference on Information and Knowledge Management (CIKM)*, Washington, D.C.

CHO, S., AMER-YAHIA, S., LAKSHMANAN, L., AND SRIVASTAVA, D. 2002. Optimizing the secure evaluation of twig queries. In *Proceedings of the 28th International Conference on Very Large Databases (VLDB)*, Hong Kong.

COMPUTER SECURITY INSTITUTE. 2003. CSI/FBI computer crime and security survey. http://www.gocsi.com/forms/fbi/pdf.html.

DAMIANI, E., DE CAPITANI DI VIMERCATI, S., PARABOSCHI, S., AND SAMARATI, P. 2002. A fine-grained access-control system for XML documents. *ACM Transactions on Information and System Security (TISSEC) 5*, 2, 169–202.

DEVANBU, P., GERTZ, M., KWONG, A., MARTEL, C., NUCKOLLS, G., AND STUBBLEBINE, S. 2001. Flexible authentification of XML documents. In *Proceedings of the 8th ACM International Conference on Computer and Communication Security*, Philadelphia, PA.

DIAO, Y. AND FRANKLIN, M. 2003. High-performance XML filtering: An overview of filter. In *Proceedings of the 20th IEEE International Conference on Data Engineering (ICDE)*, Bangalore, India.

EL KALAM, A., BENFERHAT, S., MIEGE, A., BAIDA, R., CUPPENS, F., SAUREL, C., BALBIANI, P., DESWARTE, Y., AND TROUESSIN, G. 2003. Organization based access-control. In *Proceedings of the 4th IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY).* New York.

FAN, W., CHAN, C., AND GAROFALAKIS, M. 2004. Secure XML querying with security views. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Paris, France.

FINANCE, B., MEDJOUB, S., AND PUCHERAL, P. 2005. The case for access-control on XML relationships. In *Proceedings of the 14th IEEE International Conference on Information and Knowledge Management (CIKM)*, Bremen, Germany.

GABILLON, A. 2004. An authorization model for XML databases. In *Proceedings of the ACM Workshop on Secure Web Services*, Fairfax, VA.

GREEN, T., GUPTA, A., MIKLAU, G., ONIZUKA, M., AND SUCIU, D. 2004. Processing XML streams with deterministic automata and stream indexes. *ACM Transaction on Database Systems (TODS) 29*, 4 (Dec.), 752–788.

HACIGUMUS, H., IYER, B., LI, C., AND MEHROTRA, S. 2002. Executing SQL over encrypted data in the database-service-provider model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, WI.

HE, J. AND WANG, M. 2001. Cryptography and relational database management systems. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS),* Grenoble, France.

HENDERSON, N. J., WHITE, N. M., AND HARTEL, P. H. 2001. iButton enrolement and verification requirements for the pressure sequence smartcard biometric. In *Proceedings of the International Conference on Research in SmartCards*.

HOPCROFT, J. AND ULLMAN, J. 1979. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, Reading, MA.

KUDO, M. AND HADA, S. 2000. XML document security based on provisional authorization. In *Proceedings of the ACM International Conference on Computer and Communications Security (CCS),* Athens, Greece.

MENEZES, A., OORSCHOT, P., AND VANSTONE, S. 1996. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, FL.

MERKLE, R. 1989. A certified digital signature—Advances in cryptology. In *Proceedings of Crypto*, Santa Barbara, CA.

MICROSOFT, WINDOWS MICROSOFT MEDIA 9. http://www.microsoft.com/windows/windowsmedia/.

MIKLAU, G. AND SUCIU, D. 2002. Containment and equivalence for an XPath fragment. In *Proceedings of the ACM International Conference on Principles of Database Systems (PODS)*, Madison, WI.

MIKLAU, G. AND SUCIU, D. 2003. Controlling access to published data using cryptography. In *Proceedings of the 29th International Conference on Very Large Databases (VLDB),* Berlin, Germany.

NG, W., OOI, B., TAN, K., AND ZHOU, A. 2003. Peerdb: A p2p-based system for distributed data sharing. In *Proceedings of the IEEE International Conference on Data Engineering,* Bangalore, India.

ODRL. The Open Digital Rights Language Initiative. http://odrl.net/.

PENG, F., AND CHAWATHE, S. 2003. XPath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, San Diego, CA.

RAY, I., RAY, I., AND NARASIMHAMURTHI, N. 2002. 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.

RAY, I. AND RAY, I. 2002. Using compatible keys for secure multicasting in e-commerce. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS)*, FL.

SAX PROJECT. Simple API for XML. http://www.saxproject.org/.

SCHNEIER, B. 1996. *Applied Cryptography*, 2nd ed., Wiley, New York.

SMARTRIGHT. The SmartRight Content Protection System. http://www.smartright.org/

TCPA. Trusted computing platform alliance. http://www.trustedcomputing.org/

TOLANI, P. AND HARITSA, J. 2002. XGRIND: A query-friendly XML compressor. In *Proceedings of the 18th International Conference on Data Engineering (ICDE)*, San Jose, CA.

TOXGENE. The ToX XML Data Generator. http://www.cs.toronto.edu/tox/toxgene/.

UW XML. UW XML Data Repository. http://www.cs.washington.edu/research/xmldatasets/.

VINGRALEK, R. 2002. GnatDb: A small-footprint, secure database system. In *Proceedings of the 28th W3C International Conference on Very Large Databases (VLDB)*, Hong Kong.

W3C DOM. DOM: Document Object Model. http://www.w3.org/DOM.

W3C PICS. PICS: Platform for Internet Content Selection. http://www.w3.org/PICS.

W3C XMLENC. XML Encryption Requirements, http://www.w3.org/TR/xml-encryption-req

XACML. OASIS eXtensible access-control Markup Language (XACML). http://docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf.

XRML. XrML eXtendible rights Markup Language. http://www.xrml.org/

Received September 2005; revised July 2006 and February 2007; accepted May 2007