Client-Based Access Control Management for XML documents

Luc Bouganim, Dang Ngoc, Philippe Pucheral

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Client-Based Access Control Management for XML documents

Luc Bouganim, François Dang Ngoc, Philippe Pucheral

N° 5282

Juillet 2004

ISSN 0249-6399

Client-Based Access Control Management for XML documents

Luc Bouganim[1], François Dang Ngoc[2], Philippe Pucheral[3]

Projet SMIS

Rapport de recherche n° 5282 – Juillet 2004 - 27 pages

Abstract: The erosion of trust put in traditional database servers and in Database Service Providers, the growing interest for different forms of data dissemination and the concern for protecting children from suspicious Internet content are different factors that lead to move the access control from servers to clients. Several encryption schemes can be used to serve this purpose but all suffer from a static way of sharing data. With the emergence of hardware and software security elements on client devices, more dynamic client-based access control schemes can be devised. This paper proposes an efficient client-based evaluator of access control rules for regulating access to XML documents. This evaluator takes benefit from a dedicated index to quickly converge towards the authorized parts of a – potentially streaming – document. Additional security mechanisms guarantee that prohibited data can never be disclosed during the processing and that the input document is protected from any form of tampering. Experiments on synthetic and real datasets demonstrate the effectiveness of the approach.

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

[1] INRIA SMIS – Luc.Bouganim@inria.fr
[2] INRIA SMIS/University of Versailles – Francois.Dang-Ngoc@inria.fr
[3] INRIA SMIS/University of Versailles – Philippe.Pucheral@inria.fr

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 Le Chesnay Cedex (France)
Téléphone : +33 1 39 63 55 11 — Télécopie : +33 1 39 63 53 30

Client-Based Access Control Management for XML documents

Résumé: La confiance limitée accordée aux serveurs traditionnels de bases de données et aux Database Service Provider (DSP), l'intérêt croissant pour la dissémination de données et le besoin de filtrer certains contenus digitaux (e.g., contrôle parental) sont différents facteurs motivant la migration du contrôle d'accès du serveur vers le client. Plusieurs schémas de chiffrement peuvent être utilisés dans ce but mais tous imposent un partage statique des données. L'apparition d'éléments sécurisés matériels et logiciels sur les terminaux clients permet d'envisager des schémas dynamiques de contrôle d'accès.
Cet article présente une solution de ce type pour évaluer efficacement des règles de contrôle d'accès à des documents XML, éventuellement transmis en flux. Une technique spécifique d'indexation est proposée pour converger rapidement vers les parties autorisés du document XML. Des mécanismes additionnels de sécurité garantissent que les données prohibées ne sont jamais révélées lors du traitement et que le document en entrée est protégé contre toute forme de modification illicite. L'approche est validée par des expériences sur des données synthétiques et réelles.

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

1 Introduction

Access control management is one of the foundation stone of database systems and is traditionally performed by the servers, the place where the trust is. This situation, however, is rapidly evolving due to very different factors: the suspicion about Database Service Providers (DSP) regarding data confidentiality preservation [HIL02, BoP02], the increasing vulnerability of database servers facing external and internal attacks [FBI03], the emergence of decentralized ways to share and process data thanks to peer-to-peer databases [NOT03] or license-based distribution systems [XrM] and the ever-increasing concern of parents and teachers to protect children by controlling and filtering out what they access on the Internet [PIC].

The common consequence of these orthogonal factors is to move access control from servers to clients. Due to the intrinsic untrustworthiness of client devices, all client-based access control solutions rely on data encryption. The data are kept encrypted at the server and a client is granted access to subparts of them according to the decryption keys in its possession. Sophisticated variations of this basic model have been designed in different contexts, such as DSP [HIL02], database server security [HeW01], non-profit and for-profit publishing [MiS03, BCF01, Med] and multilevel databases [AkT82, BZN01, RRN02]. These models differ in several ways: data access model (pulled vs. pushed), access right model (DAC, RBAC, MAC), encryption scheme, key delivery mechanism and granularity of sharing. However these models have in common to minimize the trust required on the client at the price of a static way of sharing data. Indeed, whatever the granularity of sharing, the dataset is split in subsets reflecting a current sharing situation, each encrypted with a different key, or composition of keys. Thus, access control rules intersections are precompiled by the encryption. Once the dataset is encrypted, changes in the access control rules definition may impact the subset boundaries, hence incurring a partial re-encryption of the dataset and a potential redistribution of keys.

Unfortunately, there are many situations where access control rules are user specific, dynamic and then difficult to predict. Let us consider a community of users (family, friends, research team) sharing data via a DSP or in a peer-to-peer fashion (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) [ABM03], may evolve over time (e.g., depending on the patient's treatment) and may be subject to provisional authorizations [KmS00]. 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 [BoP02]. Regarding parental control, neither Web site nor Internet Service Provider can predict the diversity of access control rules that parents with different sensibility are willing to enforce. Finally, the diversity of publishing models (non-profit or lucrative) leads to the definition of sophisticated access control languages like XrML or ODRL [XrM, ODR]. The access control rules being more complex, the encrypted content and the licenses are managed through different channels, allowing different privileges to be exercised by different users on the same encrypted content.

In the meantime, software and hardware architectures are rapidly evolving to integrate elements of trust in client devices. Windows Media9 [Med] is an example of software solution securing published digital assets on PC and consumer electronics. Secure tokens and smart cards plugged or embedded into different devices (e.g., PC, PDA, cellular phone, set-top-box) are hardware solutions exploited in a growing variety of applications (certification, authentication, electronic voting, e-payment, healthcare, digital right management, etc.). Finally, TCPA [TCP] is a hybrid solution where a secured chip is used to certify the software's installed on a given platform, preventing them from hacking. Thus, Secure Operating Environments (SOE) become a reality on client devices [Vin02]. SOE guarantee a high tamper-resistance, generally

on limited resources (e.g., a small portion of stable storage and RAM is protected to preserve secrets like encryption keys and sensitive data structures).

The objective of this 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 access rights from encryption. The considered input documents are XML documents, the de-facto standard for data exchange. Authorization models proposed for regulating access to XML documents use XPath expressions to delineate the scope of each access control rule [BCF01, GaB01, DDP02]. Having this context in mind, the problem addressed in this paper can be stated as follows.

Problem statement

- *To propose an efficient streaming access control rules evaluator*
  The streaming requirement is twofold. First, the evaluator must adapt to the memory constrained SOE, thereby precluding materialization (e.g., building a DOM representation of the document). Second, some target applications mentioned above are likely to consume streaming documents. 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.

Contributions

To tackle this problem, we make the following four 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. 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 [DF03, GMO03, CFG02]). However, access control rules are not independent. They may generate conflicts or become redundant on given parts of the document. The proposed evaluator detects accurately these situations and exploits them to stop eagerly rules becoming irrelevant.
2. *Skip index*
   We design a streaming and compact index structure allowing to quickly converge towards the authorized parts of the input document, while skipping the others, and to compute the intersection with a potential query expressed on this document (in a pull context). Indexing is of utmost importance considering the two limiting factors of the target architecture: the cost of decryption in the SOE and the cost of communication between the SOE, the client and the server. This second contribution complements the first one to match the performance objective.
3. *Pending predicates management*
   Pending predicates (i.e., a predicate P conditioning the delivery of a subtree S but encountered after S while parsing the document) are difficult to manage. We propose a strategy to detect eagerly the pending parts of the document, to skip them at parsing time and to reassemble afterwards the relevant pending parts at the right place in the final result.

4. *Random integrity checking*
   We combine hashing and encryption techniques to make the integrity of the document verifiable despite the forward and backward random accesses generated by the two preceding contributions.

The paper is organized as follows. Section 2 introduces the XML access control model we consider and illustrates it on a motivating example. Sections 3 - 6 detail the four contributions mentioned above. Section 7 presents experimental results based on both synthetic and real datasets. Section 8 concludes. Related works are addressed throughout each section.

2 Access control model

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 [BCF01, GaB01, DDP02], even though RBAC and MAC models have also been considered [Cha00, CAL02]. We introduce below a simplified access control model for XML, inspired by Bertino's model [BCF01] and Samarati's model [DDP02] that roughly share the same foundation. Subtleties of these models are ignored for the sake of simplicity.

In this simplified model, access control rules, or access rules for short, take the form of a 3-uple <*sign, subject, object*>. *Sign* denotes either a permission (positive rule) or a prohibition (negative rule) for the read operation. *Subject* is self-explanatory. *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 $XP^{\{[],*,//\}}$ [MiS02]. This subset, widely used in practice, consists of node tests, the child axis (/), the descendant axis (//), wildcards (*) and predicates or branches […]. Attributes are handled in the model similarly to elements and are not further discussed.

The cascading propagation of rules is implicit in the model, meaning that a rule propagates from an object to all its descendants in the XML hierarchy. Due to 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 assume two rules R1 and R2 of opposite sign. These rules may conflict either because they are defined on the same object, or because they are defined respectively on two different objects O1 and O2, linked by an ancestor/descendant relationship (i.e., O1 is ancestor of O2). In the former situation, the *Denial-Takes-Precedence* policy favors the negative rule. In the latter situation, the *Most-Specific-Object-Takes-Precedence* policy favors the rule that applies directly to an object against the inherited one (i.e., R2 takes precedence over R1 on O2). Finally, if a subject is granted access to an object, the path from the document root to this object is granted too (names of denied elements in this path can be replaced by a dummy value). This *Structural* rule keeps the document structure consistent with respect to the original one.

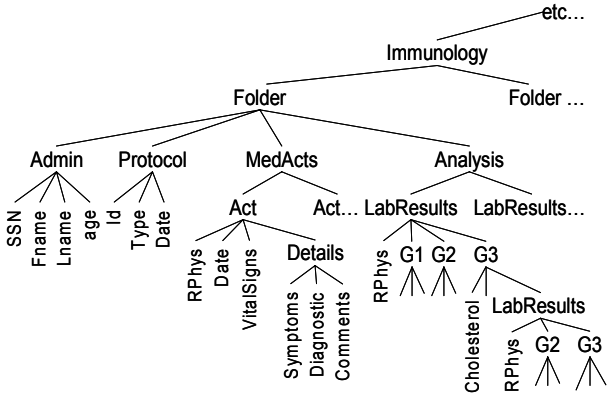
The set of rules attached to a given subject on a given document is called an *access control policy*. This policy defines an authorized view of this document and, depending on the application context, this view may be queried. We consider that queries are expressed with the same XPath fragment as access 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 rules predicates can apply on any part of the initial document.

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 only to the laboratory results and the age of patients who have subscribed to a protocol test of type G3, provided the measurement for the element Cholesterol does not exceed 250mg/dL.

Medical applications exemplify the need for dynamic access rules. For example, a researcher may be granted an exceptional and time-limited access to a fragment of all medical folders where the rate of Cholesterol exceeds 300mg/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 due to 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 due to the very high level 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).



Doctor access control policy
D1: ⊕, //Folder/Admin D2: ⊕, //MedActs[//RPhys = USER] D3: ⊖, //Act[RPhys != USER]/Details D4: ⊕, //Folder[MedActs//RPhys = USER]/Analysis
Researcher access control policy
R1: ⊕, //Folder[Protocol]//Age R2: ⊕, //Folder[Protocol/Type=G3//LabResults//G3 R3: ⊖, //G3[Cholesterol > 250] Rules 2 & 3 occur for each of the 10 groups {G1..G10}
Secretary access control policy
S1: ⊕, //Admin

Figure 1. Hospital XML document

Target architectures

Figure 2 pictures an abstract representation of the target architecture for the motivating example as well as for the applications mentioned in the introduction. The access control being evaluated on the client, the client device has to be made tamper resistant thanks to a Secure Operating Environment (SOE). As discussed in the introduction, this SOE can rely on software or hardware solutions or on a mix of them. In the sequel of this paper, and up to the perform-

ance evaluation section, we make no assumption on the SOE, except the traditional ones: 1) the code executed by the SOE cannot be corrupted, 2) the SOE has at least a small quantity of secure stable storage (to store secrets like encryption keys), 3) the SOE has at least a small quantity of secure working memory (to protect sensitive data structures at processing time). In our context, the SOE is in charge of decrypting the input document, checking its integrity and evaluating the access control policy corresponding to a given (document, subject) pair. This access control policy as well as the key(s) required to decrypt the document can be permanently hosted by the SOE, refreshed or downloaded via a secure channel from different sources (trusted third party, security server, parent or teacher, etc).

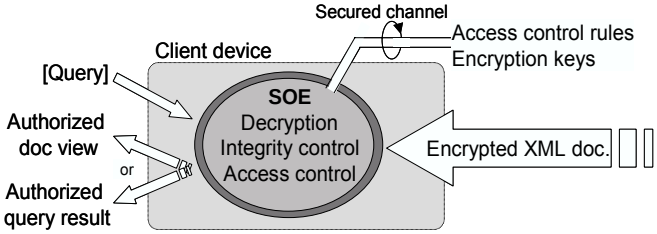


Figure 2. Abstract target architecture

3 Streaming the access control

While several access control models for XML have been proposed recently, few papers address the enforcement of these models and, to the best of our knowledge, no one considers access control in a streaming fashion. 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 [DF03, GMO03, CFG02]. These studies consider a very large number of XPath expressions (typically tens of thousands). The primary goal here is to select the subset of queries matching a given document (the query result is not a concern) and the focus is on indexing and/or combining a large amount of queries. One of the first works addressing the precise evaluation of complex XPath expressions over streaming documents is due to [PfC03] which proposes a solution to deliver parts of a document matching a single XPath. While access rules are expressed in XPath, the nature of our problem differs significantly from the preceding ones. Indeed, the rule propagation principle along with its associated conflict resolution policies (see section 2) makes access rules not independent. The interference between rules introduces two new important issues:

– *Access 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 rules scopes determines the authorization outcome for that node. This problem is made more complex by the fact that some rules are evaluated lazily due to pending predicates.

– *Access control optimization:* the nesting of rule scopes associated with the conflict resolution policies inhibits the effect of some rules. The rule evaluator must take advantage of this inhibition to suspend the evaluation of these rules and even to suspend the evaluation of all rules if a global decision can be reached for a given subtree.

3.1 Access 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* and *close* events respectively for each opening, text and closing tag in the input document.

We represent each access rule (i.e., XPath expression) by a non-deterministic finite automaton (NFA) [HoU79]. Figure 3.b pictures the Access Rules Automata (*ARA*) corresponding to

two rather simple access rules expressed on an abstract XML document. This abstract example, used in place of the motivating example introduced in Section 2, 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 thanks to a unique stack-based data structure called *Token Stack*. The top of the stack contains all active NT and PT tokens, i.e. tokens that can trigger a new transition at the next incoming event. Tokens created by a triggered transition are pushed in the stack. The stack is popped at each *close* event. The goal of Token Stack is twofold: allowing a straightforward backtracking in all ARA and reducing the number of tokens to be checked at each event (only the active ones, at the top of the stack, are considered).

– *Rule status and Authorization Stack:* Let assume for the moment that access 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 *pending* if the final state of its navigational path contains a token while the final state of some predicate path has not yet been reached. The *Authorization Stack* registers the NT tokens having reached the final state of a navigational path, at a given depth in the document. The scope of the corresponding rule is bounded by the time the NT token remains in the stack. This stack is used to solve conflicts between rules. The status of a rule present in the stack can be fourfold: *positive-active* (denoted by $\oplus$), *positive-pending* (denoted by $\oplus^?$), *negative-active* (denoted by $\ominus$), *negative-pending* (denoted by $\ominus^?$). By convention, the bottom of the stack contains an implicit *negative-active* rule materializing a closed access control policy (i.e., by default, the set of objects the user is granted access to is empty).

– *Rule instances materialization:* Taking into account the descendant axis (//) in the access 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[4]. 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*[5]. Consequently, a token (proxy) must hold the following information: RuleId (denoted by R, S, …), Navigational/Predicate status (denoted by n or p), StateId and Depth[6]. For example, $Rn2_2$ and $Rp4_2$ (also noted 2_2, 4_2 to simplify the

[4] The complexity of this problem has been highlighted in [PfC03].

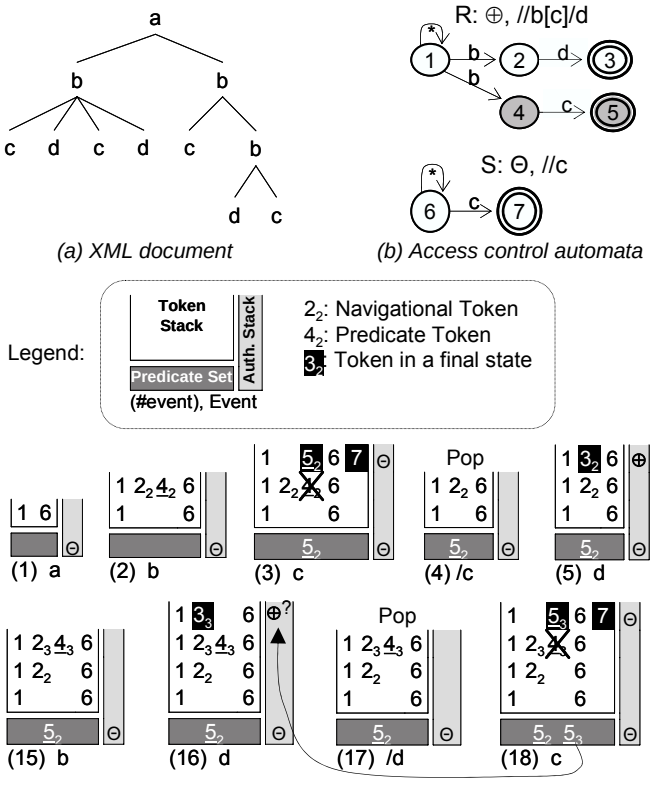
[5] To illustrate this, let us consider the rule R and the right subtree of the document presented in Figure 3. The predicate path final state *5* (expressing //b[c]) can be reached from two different instances of b, respectively located at depth 2 and 3 in the document, while the navigational path final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

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

figures) denotes the navigational and predicate tokens created in Rule R's ARA at the time element b is encountered at depth 2 in the document. If the transition between states 4 and 5 of this ARA is triggered, a token proxy $Rp5_2$ will be created and will represent the progress of the original token $Rp4_2$ in the ARA. All these tokens refer to the same rule instance since they are labeled by the same depth. A rule instance is said to be *active* or *pending* under the same condition as before, taking into account only the tokens related to this instance.

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

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 3.c represents an execution snapshot based on these data structures. This snapshot being almost self-explanatory, we detail only a small subset of steps.



(c) Snapshots of the stack structure

Figure 3. Execution snapshot

– Step 2: the *open* event *b* generates two tokens $Rn2_2$ and $Rp4_2$, participating in the same rule instance.

– Step 3: the ARA of the negative rule S reaches its final state and an active instance of S is pushed in the Authorization Stack. The current authorization remains negative. Token $Rp5_2$ enters the Predicate Set. The corresponding predicate will be considered true until level 2 of the Token Stack is popped (i.e., until event */b* is produced at step 9). Thus, there is no need to continue to evaluate this predicate in this subtree and token $Rp4_2$ can be discarded from

the Token Stack.

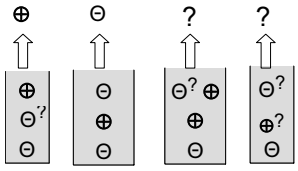
- Step 5: An active instance of the positive rule R is pushed in the Authorization Stack. The current authorization becomes positive, allowing the delivery of element d.
- Step 16: A new instance of R is pushed in the Authorization Stack, represented by token $Rn3_3$. This instance is pending since the token $Rp5_2$ pushed in the Predicate Set at step 12 (event *c*) does not participate in the same rule instance.
- Step 18: Token $Rp5_3$ enters the Predicate Set, changing the status of the associated rule instance to *positive-active*. The management of pending predicates and their effect on the delivery process is more deeply studied in section 5.

3.2 Conflict Resolution

From the information kept in the Authorization Stack, the outcome of the current document node can be easily determined. The conflict resolution algorithm presented in Figure 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 5. 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 rules. Things are slightly more complex if queries are considered too. Queries are expressed in XPath and are translated in a non-deterministic finite automaton in a way similar to access rules. However, a query cannot be regarded as an access 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.

```
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

Figure 4. Conflict resolution algorithm

3.3 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 …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≠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≠T_i.Sign) $\Rightarrow$ $\{S_i, \ldots S_k\}$ can be eliminated. Note that this strong elimination condition is sufficient but not necessary. For instance, let R and S be two positive rules respectively expressed by /a and /a/b[P1] and T be a negative rule expressed by /a/b[P2]/c. S can still be eliminated while $T \not\subset S$, because the containment holds for each subtree where the two rules are active together. The problem is particularly complex considering that the query containment problem itself has been shown co-NP complete for the class of XPath expressions of interest, that is $XP^{\{[],//,*\}}$ [MiS02]. This issue could be further investigated since more favorable results have been found for subclasses of $XP^{\{[],//,*\}}$ [ACL01], but this work is outside the scope of this paper.

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

Without any additional information on the input document, a common decision can be taken for a complete subtree rooted at node *n* iff: (1) the DecideNode algorithm can deliver a decision D (either $\oplus$ or $\ominus$) for *n* itself and (2) a rule R whose sign contradicts D cannot become active inside this subtree (meaning that all its final states, of navigational path and potential predicate paths, cannot be reached altogether). These two conditions are compiled in the algorithm presented in Figure 5. In this algorithm, AS denotes the Authorization Stack, TS the Token Stack, TS[i].NT (resp. TS[i].PT) the set of NT (resp. PT) tokens registered at level *i* in this stack and top is the level of the top of a stack. In addition, t.RuleInst denotes the rule instance associated with a given token, Rule.Sign the sign of this rule and Rule.Pred a boolean indicating if this rule includes predicates in its definition.

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

Figure 5. Decision on a complete subtree

The immediate benefit of this algorithm is to stop the evaluation for any active NT tokens and the main expected benefit is to skip the complete subtree if this decision is $\ominus$. Note however that only NT tokens are removed from the stack at line 6 The reason for this is that active PT tokens must still be considered, otherwise pending predicates could remain pending forever. As a conclusion, a subtree rooted at *n* can be actually skipped iff: (1) the decision for *n* is $\ominus$, (2) the DecideSubtree algorithm decides $\ominus$ and (3) there are no PT token at the top of the Token Stack (which turns to be empty). Unfortunately, these conditions are rarely met together, especially when the descendant axis appears in the expression of rules and predicates. 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 NT token of the query (or NT tokens when several instances of the same query can co-exist) becomes inactive (i.e., is no longer element of TS[top].NT). This token can be removed from the Token Stack but potential PT tokens related to the query must still be considered, again to prevent pending predicate to remain pending forever. As before, the subtree will be actually skipped if the Token Stack becomes empty.

4 Skip index

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

The first distinguishing feature of the required index is the necessity to keep it encrypted outside of the SOE to guarantee the absence of information disclosure. The second distinguishing feature (related to the first one and to the SOE storage capacity) is that the SOE must manage the index in a streaming fashion, similarly to the document itself. These two features lead to design a very compact index (its decryption and transmission overhead must not exceed its own benefit), embedded in the document in a way compatible with streaming. For these reasons, we concentrate on indexing the structure of the document, pushing aside the indexation of its content. Structural summaries [ABC04] or XML skeleton [BGK03] could be considered as candidate for this index. Beside the fact that they may conflict with the size and streaming requirements, these approaches do not capture the irregularity of XML documents (e.g., medical folders are likely to differ from one instance to another while sharing the same general structure).

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

4.1 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 3.3 are met. Keeping the compactness requirement in mind, the minimal structural information required to achieve this goal is the set of element tags, or tags for short, that appear in each subtree. While this metadata does not capture the tags nesting, it reveals oneself as a very effective way to filter out irrelevant XPath expressions. We propose below data structures encoding this metadata in a highly compact way. These data structures are illustrated in Figure 7.a on an abstract XML document.

– *Encoding the set of descendant tags*: The size of the input document being a concern, we make the rather classic assumption that the document structure is compressed thanks to a

dictionary of tags [ABC04, TpH02][7]. The set of tags that appear in the subtree rooted by an element *e*, named $DescTag_e$, can be encoded by a bit array, named $TagArray_e$, of length N_t, where N_t is the number of entries of the tag dictionary. A recursive encoding can further reduce the size of this metadata. Let us call *DescTag(e)* the bijective function that maps $TagArray_e$ into the tag dictionary to compute $DescTag_e$. We can trade storage overhead for computation complexity by reducing the image of *DescTag(e)* to $DescTag_{parent(e)}$ in place of the tag dictionary. The length of the *TagArray* structure decreases while descending into the document hierachy at the price of making the *DescTag()* function recursive. Since the number of element generally increases with the depth of the document, the gain is substantial. To distinguish between intermediate nodes and leaves (that do not need the TagArray metadata), an additional bit is added to each node.

- *Encoding the element tags*: In a dictionary-based compression, the tag of each element *e* in the document is replaced by a reference to the corresponding entry in the dictionary. $Log_2(N_t)$ bits are necessary to encode this reference. The recursive encoding of the set of descendant tags can be exploited as well to compress further the encoding of tags themselves. Using this scheme, $Log_2(DescTag_{parent(e)})$ bits suffice to encode the tag of an element *e*.
- *Encoding the size of a subtree:* Encoding the size of each subtree is mandatory to implement the skip operation. At first glance, log2(size(document)) bits are necessary to encode $SubtreeSize_e$, the size of the subtree rooted by an element *e*. Again, a recursive scheme allows to reduce the encoding of this size to $log_2(SubtreeSize_{parent(e)})$ bits. Storing the SubtreeSize for each element makes closing tags unnecessary.
- *Decoding the document structure*: The decoding of the document structure must be done by the SOE, efficiently, in a streaming fashion and without consuming much memory. To this end, the SOE stores the tag dictionary and uses an internal *SkipStack* to record the DescTag and SubtreeSize of the current element. When decoding an element *e*, $DescTag_{parent(e)}$ and $SubtreeSize_{parent(e)}$ are retrieved from this stack and used to decode in turn $TagArray_e$, $SubtreeSize_e$ and the encoded tag of *e*.
- *Updating the document:* In the worst case, updating an element *e* induces an update of the SubtreeSize, the TagArray and the encoded tag of each *e* ancestors and of their direct children. In the best case, only the SubtreeSize of *e* ancestors need be updated. The worst case occurs in two rather infrequent situations. The SubtreeSize of *e* ancestor's children have to be updated if the size of *e* father grows (resp. shrinks) and jumps a power of 2. The TagArray and the encoded tag of *e* ancestor's children have to be updated if the update of *e* generates an insertion or deletion in the tag dictionary.

4.2 Skip index usage

As said before, the primary objective of the Skip index is to detect rules and queries that cannot apply inside a given subtree. This means that any active token that cannot reach a final state in its ARA can be removed from the top of the Token Stack. Let us call *RemainingLabels(t)* the function that determines the set of transition labels encountered in the path separating the current state of a token *t* from the final state of its ARA, and let us call *e* the current element in the document. A token *t*, either navigational or predicate, will be unable to reach a final state in its ARA if $RemainingLabels(t) \not\subset DescTag_e$. Note that this condition is sufficient but not necessary since the Skip index does not capture the element tags nesting.

Once this token filtering has been done, the probability for the DecideSubtree algorithm to reach a global decision about the subtree rooted by the current element *e* is greatly increased since many irrelevant rules have been filtered. If this decision is negative (⊖) or pending (?), a

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

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. Let us remark that this algorithm should be triggered both on *open* and *close* events. Indeed, each element may change the decision delivered by the algorithm DecideNode, then DecideSubtree and finally SkipSubtree with the benefit of being able to skip a bigger subtree at the next step.

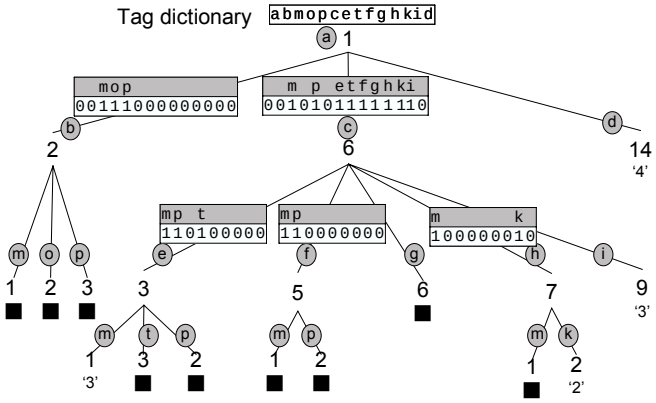
SkipSubtree () → Decision ∈ {true,false}

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
4: (TS[top].PT = ∅) then return true
5: else return false

Figure 6. Skipping decision

Figure 7 shows an illustrative XML document and its encoding, a set of access 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 thanks to $TagArray_b$ and the complete subtree can be skipped (the decision is ⊖ due to the closed access control policy). When element c is reached, 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 7.c.



(a) Encoded XML document

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

(b) Access Control Rules

a b m o p c e m3 t p f m p g h m k2 i3 d4

(c) Skipping

Figure 7. Skip index example

5 Management of pending predicates

An element in the input document is said pending if its delivery depends on a pending rule, that is a rule for which the navigational path final state has been reached but at least one predicate path final state remains to be reached. This unfavorable case is unfortunately frequent. Indeed, any rule of the form /…/e[P] lead invariably to a pending situation. Any rule of the form /../e[P]/../ generates also a pending situation until P has been evaluated to true. Indeed, a false evaluation of P does not stop the pending situation because another instance of P may be true elsewhere in the document. 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). Thus, when considering pending predicates, we make the assumption that the terminal has enough memory to buffer the pending parts of the document or that these parts can be read back from the server when pending predicates are solved. These pending parts can clearly not be buffered inside the SOE considering the assumption made on its storage capacity. The objective pursued is therefore to detect pending elements or subtrees and to leave them aside thanks to the Skip index until the pending situation is solved[8]. At that time, pending elements or subtrees are read back from the terminal. Although backward access to the document is unavoidable, the goal is to never read and analyze the same data twice. The skipping strategy and the associated reassembling strategy proposed below meet this requirement.

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 a d=6 and a 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 price 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 following information for each pending element: <*value*, *level*, *skiptree*, *condition*, *anchor*>. *value* is the node value (opening tag, text or closing tag) ; *level* refers to the element depth in the initial document (we use the term level to avoid any confusion with the depth attached to tokens) ; s*kiptree* is a boolean indicating whether the element is root of a skipped pending subtree (this information is required to externalize the complete subtree rooted at this element in case of delivery) ; *condition* is the logical expression conditioning the delivery of the element/subtree (since several pending elements are likely to depend on the same rule, logical expressions can be shared among them to gain internal storage) ; *anchor* references the target position of the element/subtree in the result. This information is kept for each pending element in a *Pending Stack*. The reassembling process is as follows:

– *Anchor assignation:* Let assume that each element *e* in the result document is labeled by a unique number *Ne* (representing for example the ordering in which elements are delivered). The future position of a pending element *e'* in the result can be uniquely identified by a single number using the following convention: *Ne* if *e'* is a potential right sibling of *e* or *-Ne* if *e'* is the potential leftmost child of *e*. No anchor need be memorized for pending right siblings and descendants of a pending element *e'* because: (1) they share *e'* anchor until one of their left sibling (see *element delivery*) or ancestor (see *embedded pending subtrees*) is delivered and (2) parent and sibling relationships among pending elements can be recovered from the Pending Stack as follows:

π *denotes the precedence relation in the Pending Stack*

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

[8] Note that all pending subtrees cannot be skipped. The skipping condition is that the pending subtree does not contain tags useful for any ARA. Otherwise, new predicates may become pending due to the skip and may even generate a deadlock between pending predicates. In addition, blocking the progress of a navigational token due to a skip would introduce a tricky problem to recover a consistent ARA state at the time the pending predicate is solved.

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

- *Element delivery*: At the time *e'* is delivered, its place in the result document is determined by its anchor (either memorized or recovered by the above mechanism). In turn, *Ne'* (resp. -*Ne'*) becomes the anchor of the pending right sibling (resp. pending leftmost child) of *e'*, if any. If *e' skiptree* is true, meaning that *e'* descendants share the same delivery condition as *e'*, the whole subtree is read back from the input document, decrypted and delivered. Finally, *e'* leaves the Pending Stack.
- *Embedded pending subtrees*: a pending element may in turn be parent of several pending elements or subtrees. Again, an innermost element/subtree may be delivered before the outermost ones. The delivery principle detailed above tackles this tricky situation with no additional difficulty, under the assumption that the Structural rule is enforced. This rule states that the result document must keep the same structure as the input one (cf . Section 2). Enforcing this rule in our context means that all pending elements in the path from the delivered innermost element/subtree to its anchor (recovered from its ancestor chain) have to be delivered too in the hierarchical order, still using the same delivery principle.

6 Random integrity checking

Encryption and hashing are required to guarantee respectively the confidentiality and the integrity of the input document. Unfortunately, standard integrity checking methods are badly adapted to our context for two important reasons. First, the memory limitation of the SOE imposes a streaming integrity checking. Second, the integrity checking must tackle the forward and backward random accesses to the document incurred by the Skip index and by the reassembling of pending document fragments. In this section, we sketch the solutions we propose to face potential attacks on an input document. For conciseness, the details of the solutions are omitted and consigned in Appendix A.

In a client-based context, the attacker is the user himself. For instance, a user being granted access to a medical folder X may try to extract unauthorized information from a medical folder Y. Let assume that the document is encrypted with a classic block cipher algorithm (e.g., DES or triple-DES) and that blocks are encrypted independently (e.g., following the ECB mode [Sch96]), identical plaintext blocks will generate identical ciphered values. In that case, the attacker can conduct different attacks: substituting some blocks of folders X and Y to mislead the access control manager and decrypt part of Y; building a dictionary of known plaintext/ciphertext pairs from authorized information (e.g., folder X) and using it to derive unauthorized information from ciphertext (e.g., folder Y); making statistical inference on ciphertext. Additionally, if no integrity checking occurs, the attacker can randomly modify some blocks, inducing a dysfunction of the rule processor (e.g., Bob is authorized to access folders of patients older than 80 and he randomly alters the ciphertext storing the age).

To face these attacks, we exploit two techniques. Regarding encryption, the objective is to generate different ciphertexts for different instances of a same value. This property could be obtained by using a Cipher Bloc Chaining (CBC) mode in place of ECB, meaning that the encryption of a block depends on the preceding block [Sch96]. 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. Regarding integrity checking, the document is split into chunks whose size is determined by the memory capacity of the SOE. Each chunk contains an encrypted *ChunkDigest* computed using a technique adapted from the Merkle hash Tree [Mer90]. This technique gracefully combines encryption and hashing functions to allow random accesses to any part of the document with an 8 bytes alignment. The most original part of the proposed strategy is that integrity is checked in cooperation with the untrusted terminal at the price of decrypting one digest per visited chunk in the worst case (i.e., when the chunks accessed are not contiguous). As a conclusion, the document is protected against tampering and confidentiality attacks while remaining agnostic regarding the encryption algorithm used to cipher the elementary data. Unlike [HIL02, BoP02],

we do no assumption on any particular way of encrypting data that could facilitate the query execution at the price of a weaker robustness against cryptanalysis attacks.

7 Experimental results

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

Experimentation platform

The abstract target architecture presented in Section 2 can be instantiated in many different ways. In this experimentation, we consider that the SOE is embedded in an advanced smart card platform. While existing smart cards are already powerful (32-bit CPU running at 30Mhz, 4 KB of RAM, 128KB of EEPROM), they are still too limited to support our architecture, especially in terms of communication bandwidth (9.6Kbps). Our industrial partner, Axalto (the Schlumberger's smart card subsidiary), announces by the end of this year a more powerful smart card equipped with a 32-bit CPU running at 40Mhz, 8KB of RAM, 1MB of Flash and supporting an USB protocol at 1MBps. Axalto provided us with a hardware cycle-accurate simulator for this forthcoming smart card. Our prototype has been developed in C and has been measured using this simulator. Cycle-accuracy guarantees an exact prediction of the performance that will be obtained with the target hardware platform.

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

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

Table 1. Communication and decryption costs

In the experiment, we consider three real datasets: *WSU* corresponding to university courses, *Sigmod records* containing index of articles and *Tree Bank* containing English sentences tagged with parts of speech [UWX]. In addition, we generate a synthetic content for the Hospital document depicted in Section 2 (real datasets are very difficult to obtain in this area), thanks to the ToXgene generator [ToX]. The characteristics of interest of these documents are summarized in Table 2.

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

Table 2. Documents characteristics

Index storage overhead

The Skip index is an aggregation of three techniques for encoding respectively tags, lists of descendant tags and subtree sizes. Variants of the Skip index could be devised by combining these techniques differently (e.g., encoding the tags and the subtree sizes without encoding the lists of descendant tags makes sense). Thus, to evaluate the overhead ascribed to each of these metadata, we compare the following techniques. NC corresponds to the original Non Compressed document. TC is a rather classic Tag Compression method and will serve as reference. In TC, each tag is encoded by a number expressed with log_2*(#distinct tags)* bits. We denote by TCS (Tag Compressed and Subtree size) the method storing the subtree size to allow subtrees to be skipped. The subtree size is encoded with log_2*(compressed document size)* bits. In TCS, the closing tag is useless and can be removed. TCSB complements TCS with a bitmap of descendant tags encoded with *#dictinct tags* bits for each element. Finally, TCSBR is the recursive variant of TCSB and corresponds actually to the Skip Index detailed in Section 4. In all these methods, the metadata need be aligned on a byte frontier. Figure 8 compares these five methods on the datasets introduced formerly. These datasets having different characteristics, the Y-axis is expressed in terms of the ratio *structure/(text length)*.

Clearly, TC drastically reduces the size of the structure in all datasets. Adding the subtree size to nodes (TCS) increases the structure size by 50%, up to 150% (big documents require an encoding of about 5 bytes for both the subtree size and the tag element while smaller documents need only 3 bytes). The bitmap of descendant tags (TCSB) is even more expensive, especially in the case of the Bank document which contains 250 distinct tags. TCSBR drastically reduces this overhead and brings back the size of the structure near the TC one. The reason is that the subtree size generally decreases rapidly, as well as the number of distinct tags inside each subtree. For the Sigmod document, TCSBR becomes even more compact than TC.

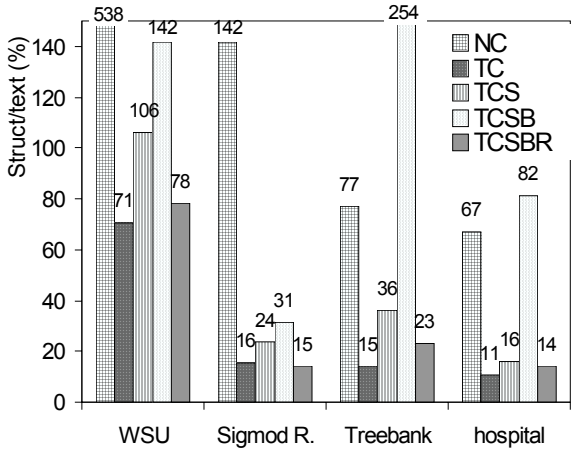


Figure 8. Index storage overhead

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 9 shows the execution time required to evaluate the authorized view of the three profiles (Secretary, Doctor and Researcher) introduced in Section 2 on the Hospital document. Integrity checking is not taken into account here. The size of the compressed document is

2.5MB and the evaluation of the authorized view returns 135KB for the Secretary, 575KB for the Doctor and 95 KB for the Researcher. In order to compare the three profiles despite this size discrepancy, the Y-axis represents the ratio between each execution time and its respective LWB. The real execution time in seconds is mentioned on each histogram. To measure the impact of a rather complex access control policy, we consider that the Researcher is granted access to 10 medical protocols instead of a single one, each expressed by one positive and one negative rule, as in Section 2.

The conclusions that can be drawn from this figure are threefold. First, the Brute-Force strategy exhibits dramatic performance, explained by the fact that the smart card has to read and decrypt the whole document in order to analyze it. Second, the performance of our TCSBR strategy is generally very close to the LWB (let us recall that LWB is a theoretical and unreachable lower bound), exemplifying the importance of minimizing the input flow entering the SOE. The more important overhead noticed for the Researcher profile compared to LWB is due to the predicate expressed on the protocol element that can remain pending until the end of each folder. Indeed, if this predicate is evaluated to false, the access rule evaluator will continue – needlessly in the current case – to look at another instance of this predicate (see Section 5). 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 explain the larger cost of evaluating the Researcher access control policy.

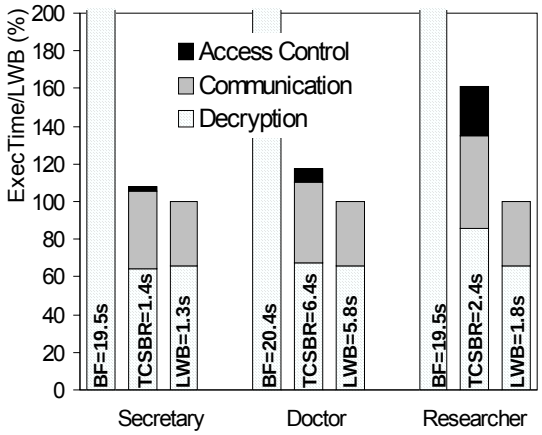


Figure 9. Access control overhead

Impact of queries

To measure accurately the impact of a query in the global performance, we consider the query //Folder[//Age>v] (v allows us to vary the query selectivity), executed over five different views built from the preceding profiles and corresponding to: a secretary (S), a part-time doctor (PTD) having in charge few patients, a full-time doctor (FTD) having in charge many patients, a junior researcher (JR) being granted access to few analysis results and a senior researcher (SR) being granted access to several analysis results.

Figure 10 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 analysed before being skipped. The parts of the document that need be analysed depends on the view and on the query. The embedded figure shows the same linearity for larger values of the query result size.

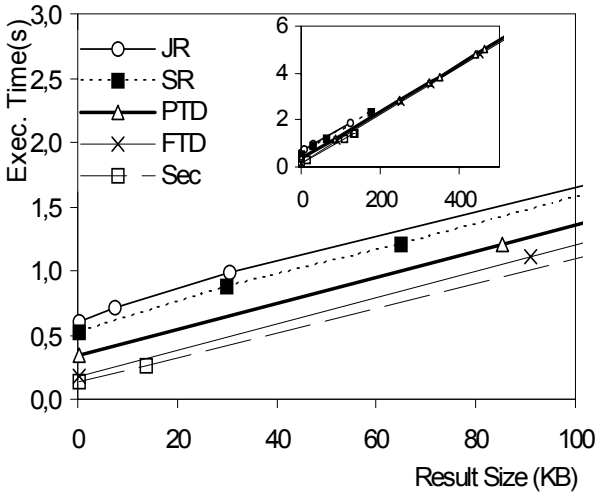


Figure 10. Impact of queries

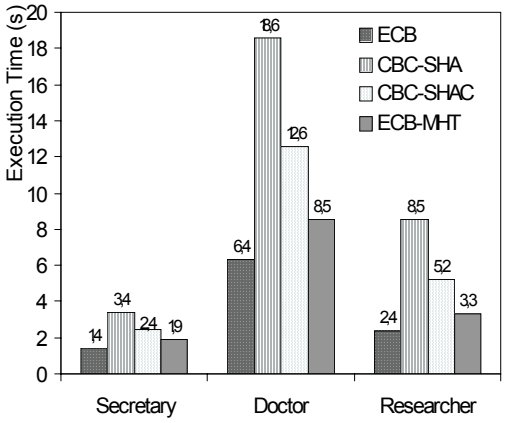


Figure 11. Impact of integrity control

Evaluation of the integrity control

Figure 11 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 9 shows that the cost ascribed to integrity checking remains quite acceptable when using the technique proposed in Section 6 (from 32% to 38%). To better capture the benefit of this technique, based on ECB and Merkle hash tree (ECB-MHT), we compare it with: (ECB) a basic ECB encryption scheme without hashing that enforces confidentiality but not tamper-resistance; (CBC-SHA) a CBC encryption scheme complemented by a SHA-1 hashing applied to the clear text version of complete chunks (this solution represents the most direct application of state-of-the-art techniques); (CBC-SHAC) that is similar to CBC-SHA except that the hashing applies to ciphered chunks, thereby allowing the SOE to check the chunk digest without decrypting the chunk itself. The results plotted in the figure are self-explanatory.

Performance on real datasets

To assess the robustness of our approach when different document structures are faced, we measured the performance of our prototype on the three real datasets WSU, Sigmod and Bank. For these documents we generated random access rules (including // and predicates). Each document exhibits interesting characteristics. The Sigmod document is well-structured, non-recursive, of medium depth and the generated access control policy was simple and not much selective (50% of the document was returned). The WSU document is rather flat and contains a large amount of very small elements (its structure represents 78% of the document size after TCSBR indexation). The Bank document is very large, contains a large amount of tags that appear recursively in the document and the generated access control policy was complex (8 rules). Figure 12 reports the results. We added in the figure the measures obtained with the Hospital document to serve as a basis for comparisons. The figure plots the execution time in terms of throughput for our method and for LWB, both with and without integrity checking. We show that our method tackles well very different situations and produces a throughput ranging from 55KBps to 85KBps depending on the document and the access control policy. These preliminary results as encouraging when compared with xDSL Internet bandwidth available nowadays (ranging from 16KBps to 128KBps).

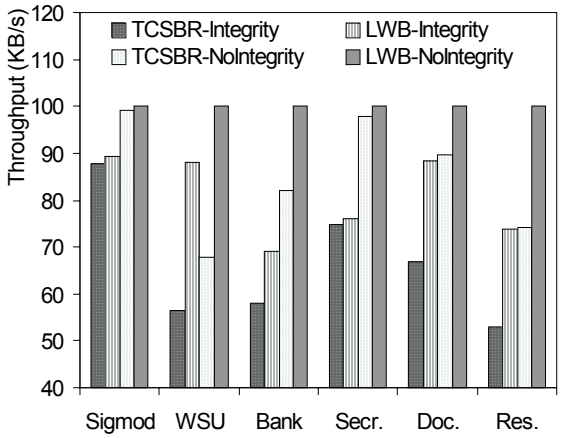


Figure 12. Performance on real datasets

8 Conclusion

Important factors motivate today the access control to be delegated 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 in client devices to propose a client-based access control manager capable of evaluating dynamic access rules on a ciphered XML document.

The contribution of this paper is fourfold. 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 paper 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 cost and the communication cost. Third, we proposed a graceful management of pending predicates. Fourth, 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 smart card simulator provided by Axalto. The global throughput measured is around 70KBps and the relative cost of the access control is less than 20% of the total cost. These first measurements are promising and demonstrate the applicability of the solution. A Javacard prototype has been developed on a USB e-gate smart card and was rewarded by the Silver Award of the e-gate open 2004 contest organized by Sun, Axalto and ST-Microelectronics.
Open issues concern the better use of query containment techniques to improve the optimization before and during the access rules evaluation as well as the definition of more accurate streaming indexation techniques. More generally, client-based security solutions deserve a special attention for the new research perspectives they broaden and for their foreseeable impact on a growing scale of applications.

Acknowledgments

Special thanks are due to Anaenza Maresca, physician at the Tenon hospital (Paris), for her contribution to the definition of the motivating example, inspired by a real-life experience. The authors wish also to thank Nicolas Sendrier and Anne Canteaut, from the cryptographic INRIA team CODES, for their helpful comments on Section 6.

9 Bibliography

[ABC04] A. Arion, A. Bonifati, G. Costa, S. D'Aguanno, I. Manolescu, A. Puglies - Efficient Query Evaluation over Compressed Data - *EDBT*, 2004.

[ABM03] A. El Kalam, S. Benferhat, A. Miege, R. Baida, F. Cuppens, C. Saurel, P. Balbiani, Y. Deswarte, G. Trouessin - Organization based access control - *IEEE 4th International Workshop on Policies for Distributed Systems and Networks*, 2003.

[AkT82] S. Akl and P. Taylor - Cryptographic solution to a problem of access control in a hierarchy - *ACM TOCS*, 1983.

[ACL01] S. Amer-Yahia, S. Cho, L. Lakshmanan, and D. Srivastava - Minimization of tree pattern queries - *ACM SIGMOD*, 2001.

[BCF01] E. Bertino, S. Castano, E. Ferrari - Securing XML documents with Author-X - *IEEE Internet Computing*, 2001.

[BGK03] P. Buneman, M. Grohe, C. Koch - Path Queries on Compressed XML - *VLDB*, 2003.

[BoP02] L. Bouganim, P. Pucheral - Chip-Secured Data Access: Confidential Data on Untrusted Servers - *VLDB*, 2002.

[BZN01] J.-C. Birget, X. Zou, G. Noubir, B. Ramamurthy - Hierarchy-Based Access Control in Distributed Environments - *IEEE ICC*, 2001.

[CAL02] S. Cho, S. Amer-Yahia, L. Lakshmanan, D. Srivastava - Optimizing the secure evaluation of twig queries - *VLDB*, 2002.

[CFG02] C Chan, P. Felber, M. Garofalakis, R. Rastogi - Efficient Filtering of XML Documents with Xpath Expressions - *ICDE*, 2002.

[Cha00] R.Chandramouli - Application of XML Tools for Enterprise-Wide RBAC Implementation Tasks - *5th ACM workshop on Role-based Access Control*, 2000.

[DDP02] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati - A Fine-Grained Access Control System for XML Documents - *ACM TISSEC, vol. 5, n. 2*, 2002.

[DF03] Y. Diao, M. Franklin - High-Performance XML Filtering: An Overview of YFilter - *ICDE*, 2003.

[FBI03] Computer Security Institute - CSI/FBI Computer Crime and Security Survey - http://www.gocsi.com/forms/fbi/pdf.html.

[GaB01] A. Gabillon and E. Bruno - Regulating access to XML documents - *IFIP Working Conference on Database and Application Security*, 2001.

[GMO03] T. Green, G. Micklau, M. Onizuka, D. Suciu - Processing XML streams with Deterministic Automata - *ICDT*, 2003.

[HeW01] J. He, M. Wang - Cryptography and Relational Database Management Systems - *IDEAS*, 2001.

[HIL02] H. Hacigumus, B. Iyer, C. Li, S. Mehrotra - Executing SQL over encrypted data in the database-service-provider model - *ACM SIGMOD*, 2002.

[HoU79] J. Hopcroft, J. Ullman - Introduction to Automata Theory, Languages and Computation - *Addison-Wesley*, 1979.

[KmS00] M. Kudo, S. Hada - XML document security based on provisional authorization - *ACM CCS*, 2000.

[Med] Windows Microsoft Windows Media 9, http://www.microsoft.com/windows/windowsmedia/.

[Mer90] R. Merkle - A Certified Digital Signature - *Advances in Cryptology--Crypto'89*, 1989.

[MiS02] G. Miklau and D. Suciu - Containment and equivalence for an XPath fragment - *ACM PODS*, 2002.

[MiS03] G. Micklau, D. Suciu - Controlling Access to Published Data Using Cryptography - *VLDB*, 2003.

[NOT03] W. Ng, B. Ooi, K. Tan, A. Zhou - Peerdb: A p2p-based system for distributed data sharing - *ICDE*, 2003.

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

[PfC03] F. Peng, S. Chawathe - XPath Queries on Streaming Data - *ACM SIGMOD*, 2003.

[PIC] W3C consortium - PICS: Platform for Internet Content Selection - http://www.w3.org/PICS.

[RRN02] I. Ray, I. Ray, N. Narasimhamurthi - A Cryptographic Solution to Implement Access Control in a Hierarchy and More - *ACM SACMAT*, 2002.

[SAX] Simple API for XML - http://www.saxproject.org/.

[Sch96] B. Schneier - Applied Cryptography - *2nd Edition, John Wiley & Sons*, 1996.

[TCP] Trusted Computing Platform Alliance, http://www.trustedcomputing.org/.

[ToX] ToXgene - the ToX XML Data Generator - http://www.cs.toronto.edu/tox/toxgene/.

[TpH02] P. Tolani, J. Haritsa - XGRIND: A Query-Friendly XML Compressor - *ICDE*, 2002.

[UWX] UW XML Data Repository - http://www.cs.washington.edu/research/xmldatasets/.

[Vin02] R. Vingralek - GnatDb: A Small-Footprint, Secure Database System - *VLDB*, 2002.

[XrM] XrML eXtendible rights Markup Language - http://www.xrml.org/

Appendix A : Encryption and Random Integrity Checking

This appendix details the strategies introduced in Section 6 to ensure the integrity and the opacity of the input document, despite random accesses to this document and taking into account the memory limitation of the SOE. This means that no information can be inferred from the encrypted document and neither random modification nor substitution of encrypted blocks can be performed without being detected by the SOE. In this section, we consider an XML document of any size, split in chunks (e.g., 2 KB), divided in small fragments (e.g., 256 bytes), and in turn subdivided in blocks of 8 bytes. The chunk partition is required to make the integrity checking compatible with the memory capacity of the SOE, fragments are introduced to allow random accesses inside a chunk and the block is the unit of encryption.

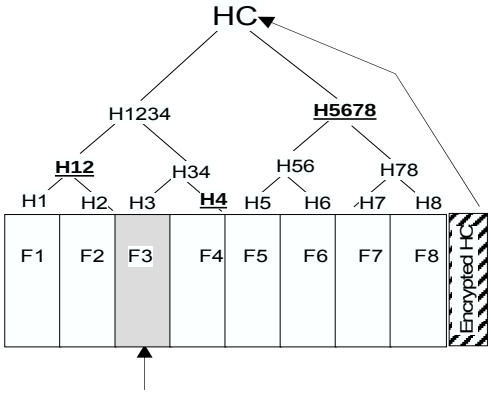
Let us first consider data encryption. As explained in Section 6, different ciphertexts must be generated for different instances of a same value in order to prevent basic attacks. Cipher Bloc Chaining (CBC) meets this requirement but incurs a non-negligible overhead. Indeed, each time the SOE accesses randomly a block in the document, the previous block has to be transferred too. In place of CBC, we perform an exclusive OR (denoted by $\oplus$) between each 8 byte block and the position of this block in the document, before encrypting the result in Electronic-CodeBook (EBC) mode. The encryption itself is performed with a triple-DES algorithm but other algorithms could be used to this purpose. Thus, a plaintext block b at absolute position p in the document is encrypted by $E_k(b \oplus p)$, where k is a secret key attached to the document and stored in the SOE. Key k can be permanently stored in the SOE or can be downloaded thanks to a secure channel along with the access control rules attached to a given (document, user) pair.

Encryption alone is not sufficient to guarantee the document integrity since the attacker can perform random modifications and substitutions in the ciphertext. Thus, we use a collision resistant hash function (e.g., SHA-1) to compute a digest of each chunk, called *ChunkDigest*, that prevents any tampering to occur without impacting this digest. Each chunk containing an identifier reflecting its position in the document, block substitutions can be easily detected. The hashing is done on the ciphertext in order to allow the cooperation of the terminal, at integrity checking time, in case of a random access (see below).

Based on this digest, a basic solution to control integrity can be envisioned. When the SOE accesses n bytes at position pos in a chunk, the terminal computes the hashing of the pos-1 preceding bytes and transmits its intermediate result to the SOE. Since hashing is computed incrementally, the SOE can continue the hashing computation on encrypted data and checks the integrity of the received data by comparing the final hashed value to ChunkDigest. Remark that ChunkDigest must be encrypted to prevent the terminal to compute by itself a new digest corresponding to tampered data. This solution incurs to communicate $sizeof(ChunkDigest) + sizeof(Chunk) - (pos\text{-}1)$ bytes to the SOE and to decrypt $sizeof(ChunkDigest) + n$ bytes in the SOE.

Although correct, the preceding 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, $sizeof(Chunk) - (pos\text{-}1) - n$ irrelevant bytes have to be transferred to the SOE. We propose below a more accurate solution that adapts the *Merkle hash tree* principle introduced in [Mer90]. In this solution, each chunk is divided into m fragments, 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 intermediated node of the tree contains a hash value computed on the concatenation of its children hash value. The ChunkDigest corresponds to the hash value attached to the binary tree root. When the SOE accesses n bytes at position pos in a fragment f of a given chunk, the terminal sends: 1) the bytes from pos up to the end of fragment f, that is $sizeof(fragment) - (pos-1)$ bytes including the n bytes of interest; 2) the intermediate hashing computation of the pos-1 first bytes of fragment f; 3) the hashing

information computed on the other fragments following the *Merkle hash tree* strategy; and 4) the encrypted ChunkDigest. Thanks to this information, the SOE can recompute the root of the Merkle hash tree and compare it to ChunkDigest as pictured in Figure F1.



The SOE accesses fragment 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 the encrypted HC
The SOE computes H3 = Hash(F3) and HC based on the hash values sent by the terminal. It decrypts the encrypted HC sent by the terminal and compares it to its own HC.

Figure F1. Random integrity checking