UNIVERSITY
OF TRENTO

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

38050 Povo – Trento (Italy), Via Sommarive 14
http://www.dit.unitn.it

QUERY REWRITING OVER GENERALIZED XML SECURITY VIEWS

Gabriel Kuper Fabio Massacci Nataliya Rassadko

October 2005

Technical Report # DIT-05-060

Query Rewriting over Generalized XML Security Views

Gabriel Kuper Fabio Massacci Nataliya Rassadko
University of Trento
38050, Povo, Trento, Italy
{kuper|massacci|rassadko}@dit.unitn.it

Abstract

We investigate the experimental effectiveness of XML security views. Our model consists of access control policies specified over DTDs with XPath expression for data-dependent access control policies. We provide the notion of *security views* for characterizing information accessible to authorized users. This is a transformed (sanitized) DTD schema that is used by users for query formulation. To avoid the overhead of view materialization in query answering, these queries later undergo rewriting so that they are valid over the original DTD schema, and thus the query answer is computed from the original XML data. We provide an algorithm for query rewriting and show its performance compared with the naive approach, i.e. the approach that requires view materialization.

1 Introduction

Specification of access control models for XML data has been a fairly active field of research in recent years [3, 5, 7, 10, 13]. Most of this previous work enforces security constraints at the document level by fully annotating the entire XML document.

As a result, one major limitation of these models is the lack of support for authorized users to query the data: they either do not provide schema information of the accessible data, or expose the entire original DTD (or its so-called "loosened" variant). In both cases, the solution is hardly practical for large and complex documents. Furthermore, fixing the access control policies at the instance level without providing or computing a schema, makes it difficult for the security officer to understand how the authorized view of a document for a user or a class of users will actually look like. On the other side, revelation of excessive schema information might lead to security breaches: an unauthorized user can deduce or infer confidential information via multiple queries and analysis of the schema even if only the accessible nodes are queried.

To overcome this limitations, the notion of XML security views was initially proposed by Stoica and Farkas [14] and later refined by Fan et al. [8] and Kuper et al. [11]. The basic idea is to provide a schema that describes the data that can be seen by the user, as well as a (hidden) set of Xpath expressions that describe how to compute the data in the view from the original data.

In the current paper, we implement and test experimentally the performance of the security view model of [11]. To this end, we define a rewriting algorithm that takes a user query over the a security view, and rewrites the query into a query over the original database. We then compare the cost of evaluating this query with that of evaluating the original query over a materialized view of the data, and show that significant performance improvements.

The paper is organized as follows. In Sec. 2 we present preliminary notions on XML and XPath. Next we introduce the notion of security specification (Sec. 3) and the notion of view (Sec. 4). In Sec. 5 we show algorithm for rewriting queries. Implementation issues are discusssed in Sec. 6. Evaluation of rewriting algorithm is provided in Sec. 7. Finally, we conclude the paper in Sec. 8.

2 A Primer on XML and XPath

We first review DTDs (Document Type Definitions [4]) and XPath [6] queries.

Definition 2.1: A DTD D is a triple $(Ele, P, \texttt{root})$, where Ele is a finite set of *element types*; `root` is a distinguished type in Ele, and P is a function defining element types such that for each A in Ele, $P(A)$ is a regular expression over $Ele \cup \{\texttt{str}\}$, where `str` is a special type denoting PCDATA, We use ϵ to denote the empty word, and "$+$", ",", and "$*$" to denote disjunction, concatenation, and the Kleene star, respectively. We refer to $A \rightarrow P(A)$ as the *production* of A. For all element types B occurring in $P(A)$, we refer to B as a *subelement type* (or a *child type*) of A and to A as a *generator* (or a *parent type*) of B. □

We assume that DTD is non-recursive, i.e., that the graph has no cycles.

Definition 2.2: An XML tree T *conforms to* a DTD D iff

1. the root of T is the unique node labelled with `root`;
2. each node in T is labelled either with an Ele type A, called an A *element*, or with `str`, called a *text node*;
3. each A element has a list of children of elements and text nodes such that their labels form a word in the regular language defined by $P(A)$;
4. each text node carries a `str` value and is a leaf of the tree.

We call T an *instance* of D if T conforms to D. □

We consider a class of XPath queries, which corresponds to the CoreXPath of Gottlob et al. [9] augmented with the union operator and atomic tests, and which is denoted by Benedict et al. [8] as $\mathcal{X}$.

The XPath axes we consider as primitive are `child`, `parent`, `ancestor-or-self`, `descendant-or-self`, `self`. Gottlob, Koch and Pichler [9] show how the semantics of such axes can be computed in polynomial time. In the sequel we denote by θ one of those primitive axes and by θ^{-1} its inverse. Notice that each primitive axis has its inverse within the same set of primitives. For instance $\texttt{descendant-or-self}^{-1}$ = `ancestor-or-self`.

Definition 2.3: An XPath expression in $\mathcal{X}$ is defined by the following grammar:

$$
\begin{array}{rcl}
\langle xpath \rangle & ::= & \langle path \rangle \mid `/` \langle path \rangle \\
\langle path \rangle & ::= & \langle step \rangle (`/` \langle step \rangle)^* \\
\langle step \rangle & ::= & \theta \mid \theta`[` \langle qual \rangle `]` \mid \langle path \rangle ` \cup ` \langle path \rangle \\
\langle qual \rangle & ::= & A \mid ` * ` \mid op\, c \mid \langle xpath \rangle \mid \\
 & & \langle qual \rangle \texttt{ and } \langle qual \rangle \mid \langle qual \rangle \texttt{ or } \langle qual \rangle \mid \\
 & & \texttt{not } \langle qual \rangle \mid `(` \langle qual \rangle `)`
\end{array}
$$

where θ stands for an axis, c is a `str` constant, A is a label, op stands for one of $=, <, >, \leq, \geq$. The result of the $qual$ production is called *qualifier* and is denoted by q. □

For sake of readability, we ignore the difference between $xpath$ and $path$, we denote both with p. We also abbreviate `self` with ϵ, $\texttt{child}[A]/p$ with A/p, $\texttt{descendant-or-self}[A]/p$ with $//A/p$, $q[op \;\; c]$ with $q \;\; op\, c$ and $p = p_1/p_2$, where p_2 is $//p_2'$, is written p as $p_1//p_2'$. The ancestor axis is also abbreviated as ../.

The semantics of XPath is obtained by adapting to our fragment the $\mathcal{S}_{\rightarrow}$, $\mathcal{S}_{\leftarrow}$, $\mathcal{E}$ operators proposed by Gottlob et al. [9] and identical to proposal of Benedickt et al. [2]. Intuitively $\mathcal{S}_{\rightarrow}[|p|](N)$ gives all nodes that are reachable from a node in N using the path p. The $\mathcal{S}_{\leftarrow}[|p|]$ functions gives all nodes from which a path p starts and arrives at some node. The $\mathcal{E}[|q|]$ function evaluates qualifiers and returns all nodes that satisfy q.

For sake of readability we overload the θ-symbol to stand for both the semantics and the syntax of axes. So given a set of nodes N of a document T we have that $\theta(N) = \{m \mid n\, \theta\, m \text{ for } n \in N\}$. In other words, $\theta(N)$ returns the nodes that are reachable according the axis from a node in N. By $\mathcal{T}(A)$ we denote the set of nodes that have element type A. By $\mathcal{T}(*)$ we denote all nodes of a document.

The semantics of the other operators is shown in Fig. 1.

3 Security Specifications

An *access specification* S is an extension of a document DTD D that associates security annotations with produc-

$$
\begin{array}{rcl}
\mathcal{S}_{\rightarrow}\,[|/p|]\,(N) & = & \mathcal{S}_{\rightarrow}\,[|p|]\,(\{\texttt{root}\}) \\
\mathcal{S}_{\rightarrow}\,[|\theta[q]|]\,(N) & = & \theta(N) \cap \mathcal{E}\,[|q|] \\
\mathcal{S}_{\rightarrow}\,[|\theta[q]/p|]\,(N) & = & \theta(\mathcal{S}_{\rightarrow}\,[|p|]\,(N)) \cap \mathcal{E}\,[|q|] \\
 & & \\
\mathcal{S}_{\rightarrow}\,[|p_1 \cup p_2|]\,(N) & = & \mathcal{S}_{\rightarrow}\,[|p_1|]\,(N) \cup \mathcal{S}_{\rightarrow}\,[|p_2|]\,(N) \\
\mathcal{S}_{\rightarrow}\,[|(p_1 \cup p_2)/p|]\,(N) & = & \mathcal{S}_{\rightarrow}\,[|p_1/p|]\,(N) \cup \mathcal{S}_{\rightarrow}\,[|p_2/p|]\,(N) \\
\mathcal{S}_{\leftarrow}\,[|/p|] & = & \begin{array}{ll} \{n \text{ occurs } inT\} & \text{if } \texttt{root} \in \mathcal{S}_{\leftarrow}\,[|/p|] \\ \emptyset & \text{otherwise} \end{array} \\
\mathcal{S}_{\leftarrow}\,[|\theta[q]|]N & = & \theta^{-1}(N \cap \mathcal{E}\,[|q|]) \\
\mathcal{S}_{\leftarrow}\,[|\theta[q]/p|]N & = & \theta^{-1}(\mathcal{S}_{\leftarrow}\,[|p|] \cap \mathcal{E}\,[|q|]) \\
 & & \\
\mathcal{S}_{\leftarrow}\,[|p_1 \cup p_2|] & = & \mathcal{S}_{\leftarrow}\,[|p_1|] \cup \mathcal{S}_{\leftarrow}\,[|p_2|] \\
\mathcal{S}_{\leftarrow}\,[|(p_1 \cup p_2)/p|] & = & \mathcal{S}_{\leftarrow}\,[|p_1/p|] \cup \mathcal{S}_{\leftarrow}\,[|p_2/p|] \\
\mathcal{E}\,[|A|] & = & \mathcal{T}\,(A) \\
\mathcal{E}\,[|q_1 \texttt{and} q_2|] & = & \mathcal{E}\,[|q_1|] \cap \mathcal{E}\,[|q_2|] \\
\mathcal{E}\,[|q_1 \texttt{or} q_2|] & = & \mathcal{E}\,[|q_1|] \cup \mathcal{E}\,[|q_2|] \\
\mathcal{E}\,[|\texttt{not} q|] & = & \{n \text{ occurs in } T\} \setminus \mathcal{E}\,[|q_2|] \\
\mathcal{E}\,[|p|] & = & \mathcal{S}_{\leftarrow}\,[|p|]
\end{array}
$$

Figure 1: The semantics of operators

tions of D.

Definition 3.1: A *authorization specification* S is a pair (D, ann), where D is a (document) DTD, ann is a partial mapping such that, for each production $A \rightarrow P(A)$ and each child element type B in $P(A)$, $\mathsf{ann}(A, B)$, if explicitly defined, is an annotation of the form:

$$\mathsf{ann}(A, B) \quad ::= \quad \mathsf{Q}[q] \quad | \quad \mathsf{Y} \quad | \quad \mathsf{N}$$

where $[q]$ is a qualifier in our fragment $\mathcal{X}$ of XPath. A special case is the root of D, for which we define $\mathsf{ann}(root) = \mathsf{Y}$ by default. □

Intuitively, annotating production rule $P(A)$ of the DTD with an unconditional annotation is a security constraint expressed at the schema level: Y or N indicates that the corresponding B children of A elements in an XML document conforming to the DTD will always be accessible (Y) or always inaccessible (N), no matter what the actual values of these elements in the document are. If $\mathsf{ann}(A, B)$ is not explicitly defined, then B *inherits* the accessibility of A. On the other hand, if $\mathsf{ann}(A, B)$ is explicitly defined it may *override* the accessibility of B obtained via propagation.

We should emphasize that semantics of qualifiers presented in this paper is *different* from that of Fan et al. [8]. According to [8] a false evaluation of the qualifier is considered as "no label" and requires the inheritance of an access from ancestors, while we assume that once evaluated on the document, a qualifier is mapped to either Y or N. This simplifies the intuition of annotations.

At the data level, the intuition is the following: given an XML document T, the document is typed with respect to the DTD, and the annotations of the DTD are attached to the corresponding nodes of the document, resulting in a *partially annotated* XML document. Then we convert the document T to a *fully annotated* one by labelling all of the unlabelled nodes with Y or N. This is done by evaluating the qualifiers and replacing them by Y or N annotations, and then using a suitable policy for completing the annotation of the yet labelled nodes of the tree. When everything is labelled we remove all N-labelled nodes from T.

The construction of the fully annotated document depends heavily on the overall security policy that is chosen to get completeness [11]. The top-down procedure that we describe next is the result of the *most-specific-takes-precedence* policy which simply says that an unlabelled node takes the security label of its first labelled ancestor. Damiani et al. [7] use a *closed* policy as default: if a node is not labelled then label it as N.

Definition 3.2: Let (D, ann) be a authorization specification and T a XML document conforming to D. The *authorized version* T_A of T according to the authorization specification is obtained from T as follows:

1. Type T with respect to D and label nodes with ann values;

2. Evaluate qualifiers top down starting from the root and replace annotations by Y or N depending on the result;

3. For each unlabelled node, label it with the annotation of its nearest labelled ancestor;

4. Delete all nodes labelled with N from the result, making all children of a deleted node v into children of v's parent.

The annotation of the document, before deleting nodes in the last step, is called the *full annotation* of T. □

4 Security Views

We now turn to the enforcement of an access specification. To this end, we introduce the notion of *security view* which consists of two parts. The first part is a schema that is seen by the user, while the second part is a function that is hidden from the user, which describes how the data in the new schema should be derived from the original data. The intuition behind our approach is similar to that of security views for relational databases in multi-level security [12] and the notation is borrowed from [8].

4.1 The Definition of Security Views

We first present the syntactic definition of security views.

Definition 4.1: Let D be a DTD. A *security view* for D is a pair (D_v, σ) where D_v is a DTD and σ is a function from pairs of element types such that for each element type A in D_v and element type B occurring in $P(A)$, $\sigma(A, B)$ is an expression in $\mathcal{X}$. □

Definition 4.2: Let $\mathcal{S} = (D_v, \sigma)$ be a security view. The semantics of $\mathcal{S}$ is a mapping from documents T conforming to D to documents T_v such that

1. T_v conforms to D_v
2. The nodes of T_v are a subset of the nodes of T, and their element type is unchanged.
3. For any node n of T which is in T_v, let A be the element type of n, and let $B_1, \dots, B_m$ be the list of element types that occur in $P(A)$. Then the children of n in T_v are

   $$\bigcup_{1 \le i \le m} \mathcal{S}_{\rightarrow} [|\sigma(A, B_i)|] (\{n\}) \ .$$

   These nodes should be ordered according to the document order in the original document.

T_v is called the *materialized version* of T w.r.t. the view $\mathcal{S}$. □

Definition 4.3: A *valid* security view is one for which the semantics are always well-defined, i.e., if for every document T, its materialized version conforms to the security view DTD. □

Not all views are valid: wrong typing, violated cardinality constraints, and other problems could be all causes of of a view to be invalid.

Security specification and views are related as follows.

Definition 4.4: Let (D, ann) be a authorization specification, and let $\mathcal{S} = (D_v, \sigma)$ be a security view for D. We say that $\mathcal{S}$ is *data equivalent* to (D, ann) iff for every document T, conforming to D, the materialized version T_v coincides with the authorized version T_A. □

In our previous work [11] we have presented an algorithm for the construction of views and have shown that the view that is built by our algorithm is data equivalent to security annotations for non-recursive DTDs.

The idea behind our algorithm is to eliminate qualifiers by expanding each qualifier into a union of two element types: one is the original element type, which is annotated Y, and the other is a new type, essentially a copy of the original type, which is annotated N. Since the tag of an element uniquely determines the type, it follows that new type names cannot match any nodes in a document that conforms to the original DTD. This is not a serious problem, as all of these new type names are ultimately deleted in the final security view.

The next step expands the annotation to a "full annotation". The notion of a full annotation was defined on annotated documents, and we showed that every document has a unique full annotation. At the schema level, however, this is not the case, as there may be several "paths" in the DTD that reach the same element type, each of which results in a different annotation. We use a similar technique to the way we handle qualifiers, i.e., we introduce new element types, and label the original one Y and the "copy" N. Finally, we delete all the element types that are labelled N, modifying the regular expressions and the σ functions correspondingly.

We show the algorithm ANNOTATE VIEW in Fig. 2 and algorithm BUILD VIEW in Fig. 3.

Definition 4.5: Let $\mathcal{S} = (D, \mathsf{ann})$ be an authorization specification. The DTD constructed by ANNOTATE VIEW algorithm is the *fully annotated* DTD corresponding to (D, ann). □

Theorem 4.1: *[11] Let (D, ann) be a security specification where D is non-recursive. Algorithms* ANNOTATE VIEW *and* BUILD VIEW *terminate and produce a valid*

Algorithm: ANNOTATE VIEW
Input: A authorization specification (D, ann)
Output: Fully annotated DTD D

1: Initialize $D_v := D$ where ann is defined on D_v as on D;
2: **for** all production rules $A \to P(A)$ in D_v **do**
3: **for** all element types B occurring in $P(A)$ **do**
4: initialize $\sigma(A \to P(A), B) := B[\epsilon]$
// Here we will eliminate qualifier annotation
5: **for** all element types B with $\mathsf{ann}(B) = \mathsf{Q}[q]$ **do**
6: add to D_v a new element type B' and a production rule $B' \to P\ \ B'$
7: set $P\ \ B'\ \ := P(B)$
8: **for** all element types C occurring in $P\ \ B'$ **do**
9: $\sigma\ \ B' \to P\ \ B'\ \ , C\ \ := \sigma(B \to P(B), C)$
10: set $\mathsf{ann}(B) = \mathsf{Y}$ and $\mathsf{ann}(B') = \mathsf{N}$
11: **for** all production rules $A \to P(A)$ **do**
12: **if** B occurs in $P(A)$ **then**
13: $\sigma(A \to P(A), B) := B[q]$;
14: $\sigma\ \ A \to P(A), B'\ \ := B[\neg q]$;
15: replace B by $B + B'$ in $P(A)$
16: **while** $\mathsf{ann}(B)$ of some element types B is undefined **do**
// Here we will get fully annotated DTD D
17: **if** all generators A of B have defined $\mathsf{ann}(A)$ **then**
18: **if** all $\mathsf{ann}(A) = \mathsf{Y}$ **then**
19: set $\mathsf{ann}(B) := \mathsf{Y}$;
20: **else if** all $\mathsf{ann}(A) = \mathsf{N}$ **then**
21: set $\mathsf{ann}(B) := \mathsf{N}$;
22: **else**
23: add to D_v a new element type B' and a production rule $B' \to P\ \ B'$
24: set $P\ \ B'\ \ := P(B)$
25: **for** all element types C occurring in $P\ \ B'$ **do**
26: $\sigma\ \ B' \to P\ \ B'\ \ , C\ \ := \sigma(B \to P(B), C)$
27: set $\mathsf{ann}(B) = \mathsf{Y}, \mathsf{ann}(B') = \mathsf{N}$,
28: **for** all generators A of B **do**
29: **if** $\mathsf{ann}(A) = \mathsf{N}$ **then**
30: replace B with B' in $P(A)$

Figure 2: Algorithm ANNOTATE VIEW

Algorithm: BUILD VIEW
Input: Fully annotated DTD D
Output: A security view (D_v, σ)

1: **for** all element types B with $\mathsf{ann}(B) = \mathsf{N}$ **do**
2: **for** all production rules $A \to P(A)$ **do**
3: **if** B occurs in $P(A)$ **then**
4: **for** all C that occurs in $P(B)$ **do**
5: set $\sigma(A \to P(A), C) := \sigma(A \to P(A), B) / \sigma(B \to P(B), C) \cup \sigma(A \to P(A), C)$
6: replace B by $P(B)$ in $P(A)$ if $B \to P(B)$ exists and by ϵ otherwise
7: D_v consists of all the element types A for which $\mathsf{ann}(A) = \mathsf{Y}$, with the σ function restricted to these types.

Figure 3: Algorithm BUILD VIEW

security view. □

Theorem 4.2: *[11] Let (D, ann) be a authorization specification, D is non-recursive, let (D_v, σ) the security view constructed by Algorithms* ANNOTATE VIEW *and* BUILD VIEW. *Let T be a document, T_A the authorized version of T and T_v the materialized version of T with respect to (D_v, σ). Then T_A is isomorphic to T_v.* □

Theorem 4.3: *[11] Let (D, ann) be a authorization specification for a non-recursive DTD, let P be size of the largest production rule in D. Let n_{Y} be the number of element types annotated with Y, and let n_{other} the number of element types otherwise annotated or not annotated. Then the size of the select function σ generated by the algorithm is bounded by $O(n_{other} \times |\mathsf{ann}|)$ and the size of the View DTD D_v is bounded by $O(n_{\mathsf{Y}} \times P^{n_{other}+1})$.* □

5 Query Rewriting

This section considers rewriting of user queries over security views $V = (D_v, \sigma)$. More precisely, user provided with the *DTD view* D_v poses a query over D_v. The query evaluation procedure may rely on two strategies:

- the *naive* strategy assumes that the user query is evaluated over the materialized security view T_v that has been extracted from initial data T by means of the σ-function or directly from the security annotation;
- the *rewriting* strategy transforms the user query q into an *equivalent* query q_t using the σ-function over the initial schema D. Query q_t can be then evaluated over the initial data set T without materialization of T_v.

The naive approach may be extremely time consuming in the case of very large XML files and multiple queries. On the other hand, one could precompute and store data views T_v. This approach may be inefficient for volatile data (e.g. auction or stock sells) or for data in which integrity across views is important. Rewriting cost is insignificant compared to the cost of view derivation from a large XML document.

Below we present our algorithm for query rewriting which has two phases: query parsing and further translation of parsed query into σ-functions.

The user query is parsed according to the grammar that we have shown in Definition 2.3. Initially, we consider the user query as $\langle xpath \rangle$. We process it recursively resulting in a *parse tree* according to the schema on Fig. 4.

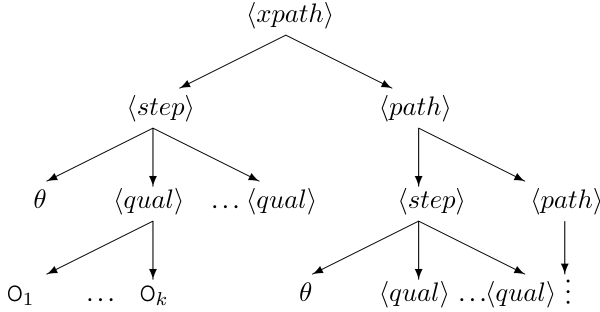

Figure 4: Parse tree schema

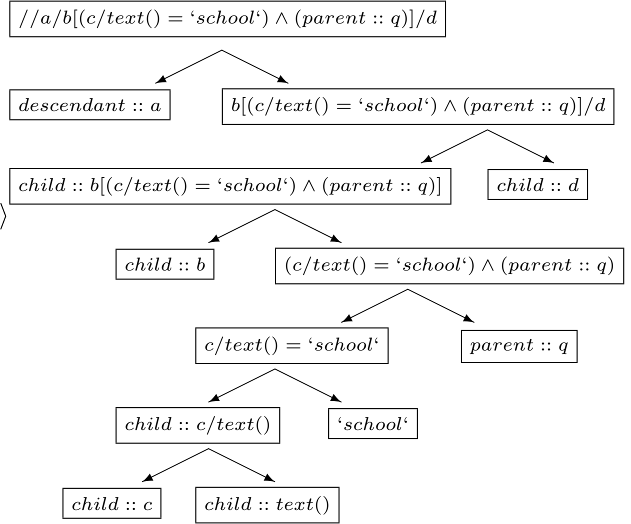

Figure 5: Parse tree of expression $//a/b[(c/text() =\text{'}school\text{'}) \wedge (parent :: q)]/d$

The intuition of parse tree schema is the following. We divide $\langle xpath\rangle$ into $\langle step\rangle$ and *remaining* $\langle path\rangle$. $\langle step\rangle$ consists of *node test* θ and zero or more qualifiers $\langle qual\rangle$. Each of these qualifiers represents a condition that the node test should satisfy. The condition is a boolean function of several arguments ($O_i, i = \overline{1,k}$) which are either $\langle path\rangle$, literal, or number.

Each node of the parse tree representation of user query is called a *subquery*.

For example, the XPath expression $//a/b[(c/text() =\text{'}school\text{'}) \wedge (parent :: q)]/d$ selects all nodes d that is a child of b, b is a child of a and has parent q and child c with text node '`school`', a is a descendant of root node. The parse tree representation is depicted on Fig. 5

For each subquery p in XPath parse tree representation and for each element A in D_v we compute a local translation $rewrite(p, A)$ which is based on translations $rewrite(p_i, B_j)$, where p_i is a direct subquery (child in parse tree) of p and B_j is a node reachable (the graph of D_v has a path to B) from A.

The algorithm presented in Fig. 6 shows the translation procedure. More precisely, in lines 1, 17, 29, 35 we can distinguish whether the subexpression is $\langle path\rangle$, $\langle qual\rangle$, θ or $\theta[\langle qual\rangle]$ respectively. In the case of $\langle path\rangle$ we process first $\langle step\rangle$ (which is rewritten to p_1) and then the remaining part as $\langle path\rangle$ (which is rewritten to p_2) recursively. The final step of $\langle path\rangle$ processing consists in joining p_1 and p_2 into path p_1/p_2 which represents the rewritten form of initial $\langle path\rangle$. The joining procedure is shown in lines 4- 16 of algorithm QUERY REWRITE.

Parsing $\theta[\langle qual\rangle]$ handles separately predicate expression $\langle qual\rangle$ and node test θ. We should mention that rewriting of predicates in $\theta[\langle qual\rangle]$ depends on node test θ rather than iterated set of DTD nodes. In lines 21- 23 and 27-28 of QUERY REWRITE we perform joining procedure respectively for binary and unary function.

In $\langle qual\rangle$ we process each operand (either $\langle path\rangle$, literal or number) of the function. Since we deal with unary and binary functions, $\langle qual\rangle$ has no more than two operands.

Intuitively, processing of node test θ produces path in terms of σ from each element A of D_v to θ. If θ has `child` axis specifier then $rewrite(\theta, A) = \sigma(A, \theta)$. Since `parent` is inverse of `child` then $rewrite(\theta, A)$ for θ with `parent` axis specifier is $\sigma^{-1}(\theta, B)$. Steps 1–11 of algorithm $getTranslation$ depicted on Fig. 9 represent the process of calculating $\sigma^{-1}(B, nt)$.

This intuition corresponds to "neighbor" axis specifiers (e.g. `child` and `parent`). In case of `descendant-or-self` (`ancestor-or-self`) we have to calculate all descendants (ancestors) and all possible paths to each descendant (ancestor). Finally, all computed paths should be translated into the σ-

function corresponding to the reverse property of axis specifier. Obviously, descendant/ancestor processing requires a different approach. Thus we introduce two auxiliary functions: $processChildParent$ on Fig. 7 and $processDescendAncest$ on Fig. 8. We should mention that each of these functions also considers the case when the node label is $*$ (line 3 of $processChildParent$ and line 7 of $processDescendAncest$) which requires rewriting for a union of nodes reachable from considered DTD node according to axis specifier.

For rewriting of descendant/ancestor relations we use the data of the statically precomputed table $preRewrite$. The idea of $preRewrite$ calculation is borrowed from [8] where $recProc$ and $traverse$ procedures are intended to capture all the paths from all DTD nodes to all their corresponding descendants, and to translate these paths to an equivalent paths over the initial DTD D. We updated subroutines $recProc$ and $traverse$ so that they precompute not only `descendant-or-self` but also `ancestor-or-self` relations. Our $preRewrite$ table is a $recrw$ table of [8] extended with the third dimension representing the DTD graph traversal: either in bottom up (`ancestor-or-self`) or top down (`descendant-or-self`) direction.

6 Implementation

At the University of Trento we have implemented a preliminary version of a Java tool that accepts user queries and returns answers as an XML document that is constructed from the set of nodes which are both visible to the user and satisfy the query conditions.

The tool consists of the following main components:

- *DTD Parser*: we extended the Wutka DTD parser [1] to be able to extract the security policy from the root element and security annotation of each DTD element. The DTD Parser returns a special object `DTD` representing a set of DTD elements (`DTDElement`), their attributes (`DTDAttribute`) and children configuration. The latter is organized as a container (`DTDContainer` object) of items (`DTDItem` object). Each item is either a container or an element name (`DTDName` object). Moreover, containers can be of three kinds: sequence (`DTDSequence`, i.e. items delimited by commas),

[1] http://www.wutka.com/dtdparser.html

Algorithm: QUERY REWRITE
Input: a subquery q (as a string)
Output: a query p locally rewritten in terms of σ(as a string)
1: **if** q is $\langle path \rangle$ **then**
  // $q = firstStep/remainingSteps$
2:   q_1 = q.getFirstStep(); p_1 = QUERY REWRITE(q_1);
3:   q_2 = q.getRemainingSteps(); p_2 = QUERY REWRITE(q_2);
4:   $p = p_1/p_2$;
5:   **for** all elements A of D_v **do**
6:     **if** $rewrite(p_1, A) = \emptyset$ **then**
7:       $rewrite(p, A) = \emptyset$; $reach(p, A) = \emptyset$;
8:     **else**
9:       $newRw = \emptyset$;
10:       **for** each v in $reach(p_1, A)$ **do**
11:         $newRw = newRw \cup rewrite(p_2, v)$;
12:         $reach(p, A) = reach(p, A) \cup reach(p_2, v)$;
13:       **if** $newRw \neq \emptyset$ **then**
14:         $rewrite(p, A) = rewrite(p_1, A)/newRw$;
15:       **else**
16:         $rewrite(p, A) = \emptyset$; $reach(p, A) = \emptyset$;
17: **else if** q is $\langle qual \rangle$ **then**
18:   **if** q has two operands **then**
19:     q_1 is the first operand; p_1 = QUERY REWRITE(q_1);
20:     q_2 is the second operand; p_2 = QUERY REWRITE(q_2);
21:     $p = p_1$ q.getOperator() p_2;
22:     **for** all elements A of D_v **do**
23:       $rewrite(p, A)$ = $rewrite(p_1, A)$ q.getOperator() $rewrite(p_2, A)$;
24:   **else**
    // q has one operand, i.e. function is either not, unary minus
    // or empty operator. The latter means that q does not have
    // operator at all (e.g. q is $\langle path \rangle$)
25:     q_0 is the operand; p_0 = QUERY REWRITE(q_0);
26:     q.getOperator() $p = p_0$ q.getOperator();
27:     **for** all elements A of D_v **do**
28:       $rewrite(p, A)$ =q.$getOperator()rewrite(p_0, A)$;
29: **else if** q is θ **then**
30:   $label$ = q.getLabel(); $axisSpecifier$ = q.getAxisSpecifier();
31:   **if** $axisSpecifier$ is '`child`' or '`parent`' **then**
32:     p =processChildParent($label$, $axisSpecifier$);
33:   **else if** $axisSpecifier$ is '`descendant-or-self`' or '`ancestor-or-self`' **then**
34:     p =processDescendAncest($label$, $axisSpecifier$);
35: **else if** q is $\theta[\langle qual \rangle]$ **then**
  // $q = nodeTest[filter_1] \ldots [filter_n]$
36:   q_0 = q.getNodeTest();
37:   $p = q_0$;
38:   **for** all filters of q **do**
39:     q_i is the next filter; p_i = QUERY REWRITE(q_i);
40:     $p' = p[q_i]$;
41:     **for** all elements A of D_v **do**
42:       $rewrite(p', A) = rewrite(p, A)[rewrite(q_i, q_0)]$;
43:       $reach(p', A) = A$;
44:     $p = p'$;
45: **else if** (q is literal) or (q is number) **then**
46:   $p = q$;
47:   $rewrite(p, A) = p$;
48: return p;

Figure 6: Algorithm QUERY REWRITE

Algorithm: processChildParent
Input: node label $label$, node axis specifier $axisSpecifier$ (as a string)
Output: a query p locally rewritten in terms of σ

```
1: p = axisSpecifier::label;
2: for all elements A of D_v do
3:     if label = * then
4:         for each node v that is in relation axisSpecifier with A do
5:             σ = getTranslation(A,v,isReverse(axisSpecifier));
6:             rewrite(p, A) = rewrite(p, A) ∪ σ;
7:             reach(p, A) = reach(p, A) ∪ v
8:     else
9:         if label is in relation axisSpecifier with A then
10:            rewrite(p, A) =getTranslation(A,v,isReverse(axisSpecifier));
11:            reach(p, A) = label;
12:        else
13:            rewrite(p, A) = ∅; reach(p, A) = ∅;
14: return p;
```

Figure 7: Algorithm processChildParent

Algorithm: processDescendAncest
Input: node label $label$, node axis specifier $axisSpecifier$ (as a string)
Output: a query p locally rewritten in terms of σ

```
1: p = axisSpecifier::label;
2: if axisSpecifier = descendant-or-self then
3:     q =‘⇊’;
4: else
       // axisSpecifier = ancestor-or-self
5:     q =‘⇈’;
6: for all elements A of D_v do
7:     if label = * then
           // reach(q, A) and preRewrite(q, A, B) are precomputed
8:         for each B in reach(q, A) do
9:             if preRewrite(q, A, B) ≠ ∅ then
10:                rewrite(p, A) = rewrite(p, A) ∪ preRewrite(q, A, B);
11:                reach(p, A) = reach(p, A) ∪ B
12:    else
13:        if preRewrite(q, A, label) ≠ ∅ then
14:            rewrite(p, A) = rewrite(p, A) ∪ preRewrite(q, A, label);
15:            reach(p, A) = reach(p, A) ∪ label
16: return p;
```

Figure 8: Algorithm processDescendAncest

choice (`DTDChoice`, i.e. items are delimited by vertical bars), and mixed (`DTDMixed`, i.e. includes `PCDATA`). However Wutka's `DTDElement` object has two significant drawbacks: container configuration complicates the process of retrieval of children set, and `DTDElement` does not provides access to parents. To overcome these limitations, we added to `DTDElement` class two additional fields: `children` and `parents` representing plain lists of children and parents names respectively. Thus these fields represent graph structure of input DTD. Their content is formed at the step of DTD parsing.

Algorithm: getTranslation
Input: elements A, B of D_v (as string), node axis specifier direction $reverse$ (as boolean)
Output: a $\sigma(A, B)$ in direct or reverse direction

```
1: if reverse = true then
       // σ(B, A) is a PathExpression
2:     str =‘parent :: A’;
3:     σ(B, A) = σ(B, A).getRemainingSteps();
4:     while σ(B, A) ≠ ∅ do
5:         step = σ(A, B).getFirstStep();
6:         σ(B, A) = σ(B, A).getRemainingSteps();
7:         if σ(B, A) ≠ ∅ then
8:             p = self :: step/p;
9:         else
10:            p = parent :: step/p;
11:    return p
           // string p represents σ(B, A) in reverse order, i.e. as σ(A, B)
12: else
13:    return σ(A, B);
```

Figure 9: Algorithm getTranslation

- *View Builder*: implements algorithms ANNOTATE VIEW and BUILD VIEW.
- *Query Parser*: we used the SAXON [2] processor to parse XPath expression into their tree representation. Query Parser also performs evaluation of the rewritten query over XML source. This functionality is stipulated by the SAXON XPath query implementation via the `XPathEvaluator` object which is able to parse the XML source, to create the intermediate parse tree representation of the XPath query, and finally to evaluate parsed query over the XML document. In addition Query Parser performs output of answer set to an XML file.
- *Query Rewriter*: implements algorithm QUERY REWRITE
- *DOM Validator*: performs checks the validity of XML document (i.e. XML document should conform to the rules of DTD schema), parses XML into DOM tree, and produces the materialized view. We used Xerses [3] processor for these purposes.

To write the XML file (either materialized view or answer set), we use JAXP `DocumentBuilder` [4].

[2] http://saxon.sourceforge.net/
[3] http://xml.apache.org/xerces2-j/
[4] http://java.sun.com

```
<!ATTLIST catgraph security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST regions security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST categories security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST person
    security_annotation_data CDATA #FIXED "Q"
    security_annotation_xpath
    CDATA #FIXED "self::node()[@id=$login]">
<!ATTLIST open_auction
    security_annotation_data CDATA #FIXED "Q"
    security_annotation_xpath CDATA
    #FIXED "./bidder/personref[@person=$login]">
<!ATTLIST closed_auction
    security_annotation_data CDATA #FIXED "Q"
    security_annotation_xpath CDATA
    #FIXED "./buyer[@person=$login]">
<!ATTLIST privacy security_annotation_data
    CDATA #FIXED "N">
```

Figure 10: Buyer policy

```
<!ATTLIST catgraph security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST regions security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST categories security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST creditcard
    security_annotation_data CDATA #FIXED "Q"
    security_annotation_xpath CDATA #FIXED
    "parent::person[@id=$login]">
<!ATTLIST profile
    security_annotation_data CDATA #FIXED "Q"
    security_annotation_xpath CDATA #FIXED
    "parent::person[@id=$login]">
<!ATTLIST buyer
    security_annotation_data CDATA #FIXED "Q"
    security_annotation_xpath CDATA #FIXED
    "parent::person/seller[@person=$login]">
<!ATTLIST open_auction
    security_annotation_data CDATA #FIXED "Q"
    security_annotation_xpath CDATA #FIXED
    "seller[@person=$login]">
<!ATTLIST closed_auction
    security_annotation_data CDATA #FIXED "N">
<!ATTLIST privacy security_annotation_data
CDATA #FIXED "N">
```

Figure 11: Seller policy

```
<!ATTLIST catgraph security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST regions security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST categories security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST buyer
    security_annotation_data CDATA #FIXED "Y">
<!ATTLIST seller
    security_annotation_data CDATA #FIXED "Y">
<!ATTLIST bidder
    security_annotation_data CDATA #FIXED "Y">
<!ATTLIST people security_annotation_data
    CDATA #FIXED "N">
<!ATTLIST open_auction
    security_annotation_data CDATA #FIXED "N">
<!ATTLIST closed_auction
    security_annotation_data CDATA #FIXED "N">
<!ATTLIST privacy security_annotation_data
    CDATA #FIXED "N">
```

Figure 12: Visitor policy

7 Experimental Results

7.1 Experimental framework

XML documents. To generate a set of XML documents we use XMark benchmark [1]. We generated 31 XML documents with factor $i/10000, i = \overline{100, 130}$. The size of these XML files varies from 1Mb to 1.2Mb.

Security annotation. XMark benchmark provides the DTD schema auctions.dtd which describes an auction scenario. It defines 77 elements describing a list of auction items, information about bidders, sellers, buyers, etc.

We have defined three user roles:

- *buyer*: can see personal information, open auctions where he is one of the bidders, closed auction where he is a buyer. Buyer cannot see privacy info, data about regions, category graph and categories. DTD representation of buyer's policy is depicted in Fig. 10.
- *seller*: is permitted to see own profile and credit card info, as well as open auctions where he is a seller. Seller can also see who buys his items. Seller cannot see privacy info, data about regions, category graph and categories. Seller's policy is shown in Fig. 11.
- *visitor*: is allowed to read information about bidders, sellers and buyers. Personal info and privacy info, as well as data about regions, category graph and categories are unavailable for visitor. Security annotation for seller is presented in Fig. 12.

For all three roles we assume that root `site` is annotated by `Y` policy propagation is performed in top down

Table 1: Query rewriting evaluation

	Q_1	Q_2	Q_3	Q_4	Q_5
buyer	12.5	11.2	7.2	15.7	11
seller	11	10.8	9.5	14.1	15.7
visitor	3.2	0	0	0	1.6

manner, default security policy is closed.

Queries. We consider the following set of queries to be evaluated over the data set:

$$\begin{aligned}
Q_1 &= .//person/name \\
Q_2 &= .//open_auction/(bidder|quantity) \\
Q_3 &= .//open_auction[seller\ and\ bidder] \\
Q_4 &= .//*[name]/parent::people/person \\
Q_5 &= .//bidder/parent::*
\end{aligned}$$

Thus all queries contain a step with axis specifier `descendant-or-self`. Moreover query Q_2 has union operation, predicate with $\wedge$ operation is included in query Q_3, examples of usage of $*$ and reverse axis specifier (`parent`) are shown in queries Q_4 and Q_5.

7.2 Evaluation

In Table 1 we show the time that is required to rewrite queries $Q_i, i = \overline{1,5}$ over DTD views built for roles *buyer*, *seller* and *visitor*. Since we rewrote queries for each XML file (we have 31 different XML files) and for each login (we have 10 logins), each cell of Table 1 presents time (in milliseconds) as arithmetic mean of 310 relevant values.

Next we compare two strategies of query answering: naive and advanced. For each XML document we ran evaluation of each query from the viewpoint of 10 users ($login = person_i, i = \overline{1,10}$). Moreover, each user tries to login under different roles. One dimension of our evaluation is query evaluation time depending on the size of initial XML file.

In advanced approach time depends on the following steps:

1. DTD parsing, DTD annotation and building of DTD view D_v;
2. query parsing;
3. **query rewriting**;
4. evaluation of query over **initial** XML source.

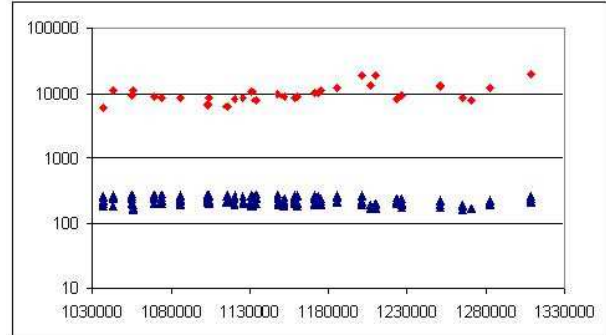

Figure 13: Query evaluation for buyer role

In naive approach time measurement is conditioned by the following steps:

1. DTD parsing, DTD annotation and building of DTD view D_v;
2. **building of sanitized XML source (view materialization)**;
3. query parsing
4. evaluation of query over **sanitized** XML source.

We emphasized with bold font those steps that are specific for a particular approach.

Figures 13, 14 and 15 show the dependency of query evaluation time on the size of the initial XML document for buyer, seller and visitor respectively. Horizontal axis represents XML size in bytes, vertical axis shows query evaluation time in milliseconds. In all three pictures we can see two main trends: upper trend (diamonds) is produced by the naive approach, lower one (triangles) stands for advanced approach. It is easy to see that naive approach answers user query much slower than the advanced one.

The second dimension of our evaluation is related to information about the space of documents involved into experiments. We tried to decrease processing time by storing materialized view. However, since policy for buyer

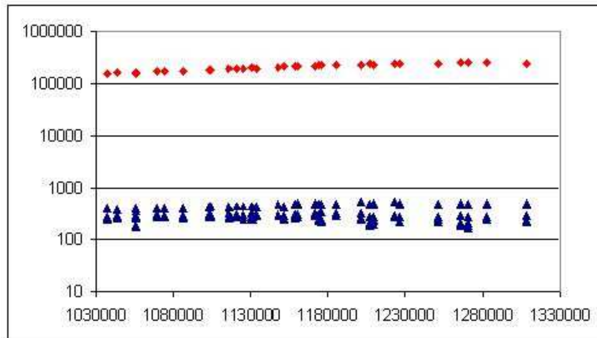

Figure 14: Query evaluation for seller role

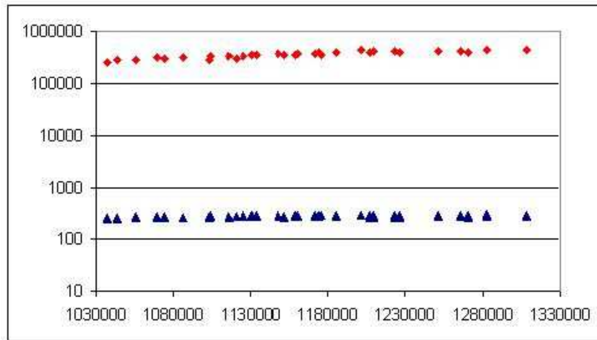

Figure 15: Query evaluation for visitor role

and seller include conditions on user login, we faced with the problem of preserving and selecting views for all logins and for all roles. For example, the smallest XML document that we generated by XMark has approximately 250 people identifiers. Each of these people may want to see the data stored in that XML.

In Fig. 16 we show the comparison of size of the initial XML document and its materialized view. The policy of visitor role does not contain any login-based conditions. Therefore views are the same for all logins. However, the size of materialized view is around 100Kb provided the initial XML file is 1Mb size. Views for seller are even bigger. And if we want to store the views for all sellers we should reserve 25Mb of space only for one role. Moreover real-life data may require much more space. Finally, maintaining the integrity of fast changing auction data in

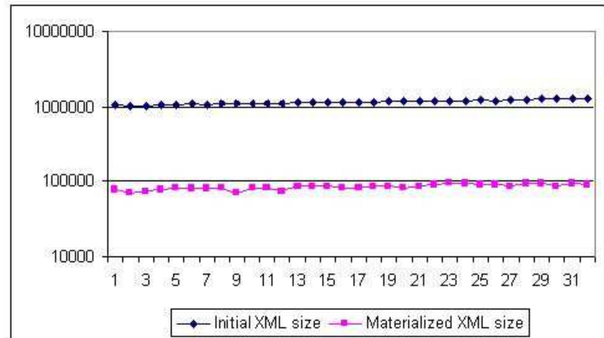

Figure 16: Comparison of size of initial and materialized XML files for visitor

250 views is hardly effective solution.

8 Conclusions

In this paper, we have studied the performance of answering queries on an XML database, subject to access control annotations applied on the original DTD. We show that the query rewriting approach compared to the naive one is more efficient in sense of time and space.

Time effectiveness takes place because we are delivered from view materialization which is a very time consuming operation. In our experimental benchmark the query rewriting strategy issues answer for user query approximately one hundred times faster than the naive strategy. Another considered point is the space preserving property of advanced method: naive approach in our experimental framework generates views that require 2.5 times more space than the initial data set. Moreover, the number of views can be extremely large that may cause problems with the maintenance of data integrity.

One main area of future work is to evaluate the effect of different security policies. We have used a top-down policy in the current paper, but some of the existing work in this area prefers other policies. Our previous paper describes which policies are reasonable, in the sense that they always annotate a document completely and unambiguously. The open problem is whether the notion of security view can be adapted to all, or some, of these security policies, and the design of efficient algorithms for

those cases where this is possible.

References

[1] XMark – An XML Benchmark Project. http://monetdb.cwi.nl/xml/index.html.

[2] M. Benedikt, W. Fan, and G. M. Kuper. Structural properties of XPath fragments. In *Proceedings of the International Conference on Database Theory*, 2003.

[3] E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Transactions on Information and System Security*, 5(3):290–331, 2002.

[4] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. *Extensible Markup Language (XML) 1.0*. W3C, Feb. 1998.

[5] S. Cho, S. Amer-Yahia, L. Lakshmanan, and D. Srivastava. Optimizing the secure evaluation of twig queries. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[6] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0. W3C Recommendation. http://www.w3.org/TR/xpath, November 1999.

[7] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *ACM Transactions on Information and System Security*, 5(2):169–202, 2002.

[8] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure XML querying with security views. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 587–598. ACM Press, 2004.

[9] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithm for processing XPath queries. In *Proceedings of the International Conference on Very Large Data Bases*, 2002.

[10] S. Hada and M. Kudo. XML Access Control Language: Provisional Authorization for XML Documents. http://www.trl.ibm.com/projects/xml/xacl/, 2000.

[11] G. Kuper, F. Massacci, and N. Rassadko. Generalized xml security views. In *SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies*, pages 77–84, New York, NY, USA, 2005. ACM Press.

[12] T. F. Lunt, D. E. Denning, R. R. Schell, M. Heckman, and W. R. Shockley. The SeaView security model. *IEEE Transactions on Software Engineering*, 16(6):593–607, 1990.

[13] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. In *Proceedings of the 10th ACM conference on Computer and communication security*, pages 73–84. ACM Press, 2003.

[14] A. Stoica and C. Farkas. Secure XML views. In *Research Directions in Data and Applications Security, IFIP WG 11.3 Sixteenth International Conference on Data and Applications Security*, volume 256, pages 133–146. Kluwer, 2003.