Policy Classes and Query Rewriting Algorithm for XML Security Views*

Nataliya Rassadko

The University of Trento, via Sommarive 14, 38050 Povo(TN), Italy
rassadko@dit.unitn.it

Abstract. Most state-of-the-art approaches of securing XML documents are based on a partial annotation of an XML tree with security labels which are later propagated to unlabeled nodes of the XML so that the resulting labeling is full (i.e. defined for every XML node). The first contribution of this paper is an investigation of possible alternatives for policy definition that lead to a fully annotated XML. We provide a classification of policies using different options of security label propagation and conflict resolution. Our second contribution is a generalized algorithm that constructs a full DTD annotation (from the the partial one) w.r.t. the policy classification. Finally, we discuss the query rewriting approach for our model of XML security views.

1 Introduction

In [1], we presented a generalized notion of XML security views. The intuition behind XML security views is similar to that of multi-level security views for a relational database [2]: *views* are virtual tables that are defined by multilevel relational expressions over the multilevel relations and are evaluated each time the view is used; view evaluation yields a derived multilevel relation.

In a hierarchical structure like XML, it is hardly possible to define accessibility via a single query. Thus, for XML, we define a partial assignment of security labels to XML nodes; then, a security policy is applied to these security labels so that the partially annotated XML becomes fully annotated; finally, the latter is "sanitized", i.e. (some) nodes with negative authorizations are hidden (deleted or encrypted), but their permitted children are revealed (e.g., moved up to a permitted ancestor if a forbidden parent is deleted). This approach is used, for example, in [3], [4], [5], [6]. The resulting XML tree is called *authorized* (T_A). Another approach to XML view calculation enforces security annotations on the schema level. The result is a DTD schema of the permitted data (or in other words, a DTD view D_v) as in [1], [7], [8]. Then, the *materialized* version of XML document (T_M) is constructed from the initial XML document by deleting forbidden nodes w.r.t. D_v so that T_A is isomorphic to T_M.

* This work has been partially supported by MIUR under the project FIRB-ASTRO, by PAT under the project PAT-MOSTRO and by the EU Commission under the project EU-IST-IP-SERENITY.

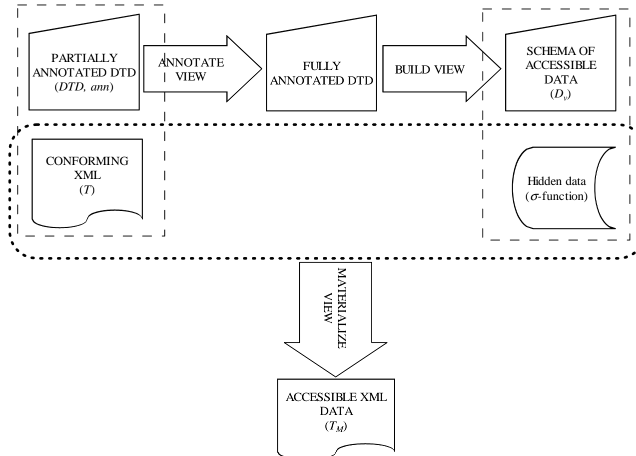

Fig. 1. Schema of accessible data materialization

A diagram of the methodology to construct a schema for the accessible data is shown in Fig. 1 (see [1] for details).

The construction of the fully annotated document, in which every node is labeled, depends on the overall *security policy* [9] that is used. The first contribution of this paper is an investigation of different alternatives for policy definition and enforcement at the level of an XML tree. Our analysis shows that not all combinations of policy options satisfy the properties of *completeness* and *consistency*, i.e., result in a single fully annotated tree. We provide a classification of policies using different options of security label propagation and conflict resolution. The second contribution is a generic algorithm that constructs a fully annotated DTD D_F (from the the partial one) according to the policy classification so that D_F reflects a full annotation of a corresponding XML document.

The final phase of XML view construction is a computation of the document T_M which conforms to D_v, i.e. materialization of accessible data. However, the user often wants to know only a small part of the materialized view, e.g., an answer on some XPath query expressed in terms of D_v. In this case, the materialization of the security view can be avoided by rewriting user queries over T_M conforming to D_v into queries over the original data, and then evaluating this query. The third contribution of this paper is the description of an algorithm for such a query rewriting.

The paper is organized as follows. First, in Sec. 2, we provide a classification of XML security policies that can be used in construction of a fully annotated XML from a partial one. Second, a general algorithm for calculation of a fully annotated DTD is presented in Sec. 3. Next, we discuss query rewriting algorithm in Sec. 4. Finally, Sec. 5 presents related work and concludes the paper.

2 Classification of Policies

We can classify security policies by *completeness* and *consistency* [9]. The former handles *unassigned values*, and the latter is to handle *conflicting assignments.*

Definition 1. *A policy is* complete *and* consistent *if every partially annotated tree can be extend to a single fully annotated tree.*

We list here several possible policies. These are variations of classical security policies [9]:

Local Propagation (LP): "open", "closed", or "none";
Hierarchy Propagation (HP): "topDown" (td), "bottomUp" (bu), or "none";
Structural Conflict Resolution (SC): "localFirst" (lf), "hierarchyFirst" (hf), or "none";
Value Conflict Resolution (VC): "denialTakesPrecedence" (dtp), "permissionTakesPrecedence" (ptp), or "none".

The LP option is similar to traditional policies for access control: in the case of "open" ("closed"), if a node is not labelled then it is labelled by Y (N); with the "none" option, an unlabeled node is not assigned any label.

The HP option specifies annotation inheritance in the tree. In the case of "td" ("bu"), an unlabelled node with a labelled parent (children) inherits the label of the latter; "none" means that no hierarchy propagation is applied. Note that the "bu" case can result in conflicts, and they should be addressed by the VC option.

The SC option specifies whether the local or hierarchy rule takes precedence ("lf" or "hf" respectively); in the case of "none", both kinds of inheritance are applied (if they are not "none") resulting in more than one possible annotations and the "winning" label is defined based on the VC option. The latter specifies how to resolve conflicts for unlabelled nodes that are assigned different labels by the preceding rules: N always has precedence over Y ("dtp"); Y always has precedence over N ("ptp"), and no choice ("none").

Finally, we also use *most-specific-takes-precedence* (MSTP) policy [9] that prohibits propagation of labels on already labeled nodes.

We represent all the possible policy options in Table 1, where symbol "$*$" means "any", i.e. any possible value from the appropriate set [1].

Definition 2. *The policy is called a* top-down/bottom-up/local/multilabel policy *if it satisfies conditions in lines 1-2/3-4/5-6/7 of Table 1.*

Proposition 1. *The top-down, bottom-up, local, and multilabel policies are complete and consistent.*

[1] Note that Table 1 indeed shows all 81 possible combination of security options, since symbols $*$ and $\neq$ mean, respectively, three and two possible values for a corresponding policy option.

Table 1. Policy alternatives

	HP	LP	SC	VC	additional condition
1	td	≠none	hf	∗	none
2	td	none	∗	∗	root is annotated
3	bu	≠none	hf	≠none	none
4	bu	none	∗	≠none	all leaves are annotated
5	∗	≠none	lf	∗	none
6	none	≠none	∗	∗	none
7	≠none	≠none	none	≠none	none
<u>8</u>	none	none	∗	∗	none
<u>9</u>	≠none	≠none	none	none	none
<u>10</u>	bu	∗	hf	none	none
<u>11</u>	bu	none	≠hf	none	none

All the other policies are classified as *unresolvable* since they do not result in a unique fully annotated tree.

In the next section, we will show how to construct a full DTD annotation (from the the partial one) for every specified policy class.

3 Construction of Security View

We start with the definition of a DTD.

Definition 3. *A DTD D is a triple (Ele, P, root), where Ele is a finite set of* element types*; root is the distinguished type in Ele called "root"; P is a function defining element types such that for each A in Ele, $P(A) = \alpha$, where α is a regular expression, defined as follows:*

$$\alpha := \mathtt{str} \mid Ele \mid \epsilon \mid \alpha + \alpha \mid \alpha, \alpha \mid \alpha *$$

where `str` *is a special type denoting PCDATA, ϵ is the empty word, and "+", ",", and "∗" denote disjunction, concatenation, and Kleene star, respectively. We refer to $A \to P(A)$ as the* DTD production rule *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* superelement type *(or a* parent type*) of B.*

We assume that DTD is finite and non-recursive, i.e., without cycles.

Definition 4. *An* authorization specification *is a pair (D, ann), where D is a DTD,* ann *is a partial mapping between adjacent DTD element types A and B:*

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

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

Every $\mathsf{ann}(A, B)$ *defines a* source element type A *denoted as* s*, a* destination element type B *denoted as* d*, and a* generator of security label for B *(or simply* generator*)* (A, B) *denoted as* g*. A mapping from a source type* A *to a set of destination types* B *is called an* annotation production *and is denoted* $P_{\mathsf{ann}}(A)$*. An* annotation production rule *is a mapping between* A *and* $P_{\mathsf{ann}}(A)$ *denoted* $A \rightarrow\ P_{\mathsf{ann}}(A)$*.*

We consider that A and B are adjacent element types, i.e., form a DTD edge [2]. Since we put annotations on DTD edges, the idea behind our algorithm is to "push" security labels from generators to destination types.

Definition 5. *If* $\mathsf{ann}(s, d) = a \neq \emptyset$*, we say that* s transmits *(or* propagates*) annotation* a *to* d *via* g*.*

After obtaining an annotation, a destination type d becomes a source type and may retransmit its annotation to generators where d is a source.

Remark 1. In the local policy, we suppose that $\mathsf{ann}(A, B)$ is an annotation between *parent* A and its *child* B, i.e., pushing security labels is performed in a top-down manner that assures that there are not any conflicts at tree level since every node B has only one parent A, i.e., only one generator. Hence, we consider the *local* policy as a subset of the *top-down* policy.

Definition 6. *The DTD document is called* fully annotated *if for every DTD node* A*, there is a function* $\mathsf{ann}_{\mathsf{data}}(A) ::= \mathsf{Y} \mid \mathsf{N}$ *called* full annotation *of the document DTD.*

The notion of a full annotation was defined for XML documents which have a unique full annotation provided a complete and consistent policy is given. At the schema level, however, there may be several "paths" transmitting different annotations to the same element type. Below we show how to resolve this problem.

Definition 7. *We denote the set of all generators of* d *as* $G(d)$*. An element type* d *with a generator* $g \in G(d)$ *such that* $\mathsf{ann}(g) = \emptyset$ *is called* expecting*.*

Definition 8. *We say that a subset* $\overline{G}(d)$ *of* $G(d)$ *has a* simultaneous impact *on* $\mathsf{ann}_{\mathsf{data}}(d)$ *if there exists an XML instance* T *conforming to a DTD schema* D *such that every instance of type* d *has a set of either outgoing or incoming edges that can be mapped to the set* $\overline{G}(d)$*. We call* $\overline{G}(d)$ *a* set of simultaneous impact *(SSI).*

Example 1. Consider a DTD:

$$A \rightarrow\ (B, C);\ B \rightarrow\ (D, E);\ C \rightarrow\ (D|E);\ D \rightarrow\ (\mathtt{str});\ E \rightarrow\ (\mathtt{str}).$$

[2] Note that an annotation production rule in this case $A \rightarrow\ P_{\mathsf{ann}}(A)$ may be either of a *top-down nature* (i.e. every $B \in\ P_{\mathsf{ann}}(A)$ is a child of A in the DTD schema) or of a *bottom-up nature* (i.e. every $B \in\ P_{\mathsf{ann}}(A)$ is a parent of A in the DTD schema).

Generators (B, D) and (C, D) belong to different SSIs on $\mathsf{ann}_{\mathsf{data}}(D)$ since a node D has either B or C parent in any XML instance. Generators (D, B) and (E, B) belong to the same SSI on $\mathsf{ann}_{\mathsf{data}}(B)$ because any node B has both D and E children in any XML instance. Generators (D, C) and (E, C) belong to different SSIs on $\mathsf{ann}_{\mathsf{data}}(C)$ as long as node C has either a D or an E child node in any XML instance.

Definition 9. *We say that d may obtain a* preliminary full *annotation (PFA) from SSI $\overline{G}(d)$ denoted as $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G}(d)}$, if for every $g \in \overline{G}(d)$, $\mathsf{ann}(g)$ is the same, non empty, and $\mathsf{ann}(g)$ is not a qualifier.*

If $\mathsf{ann}(g)$ is the same for all $g \in \overline{G}(d)$ then $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G}(d)} = \mathsf{ann}(g)$. Otherwise, we use the VC resolution option if it is not "none" [3]. From the analysis of policy options follows that value conflict may arise only in the case of the bottom-up policy class, because every XML instance usually has a node with more than one child.

In Def. 9, we required that $\mathsf{ann}(g) \neq \mathsf{Q}[q]$. Before explaining the case when $\mathsf{ann}(g) = \mathsf{Q}[q]$, we recall the meaning of $\mathsf{ann}(s, d) = \mathsf{Q}[q]$: "a node of type d is visible from node of type s via generator (s, d) if $child(s, d)[q]$ holds", where $child(s, d)$ is a function that for generator (s, d) returns a child element type ch (either s or d) w.r.t. a DTD structure [4]. At the XML instance T conforming to D, it means that condition q may evaluate to *true* for some node instances of type $child(s, d) = ch$, while for the other ch instances, it may evaluate to *false*. In the latter case, a node instance of type d is not visible. Thus, at the schema level, we perform a *splitting* operation for element type d into d_{Y} (i.e. visible d) and d_{N} (i.e. hidden d). Basically, d_{Y} is the initial element d, while d_{N} is its clone. After that, we substitute d in $P(s)$ with $d_{\mathsf{Y}} + d_{\mathsf{N}}$.

The connection of d_{Y} and d_{N} with other (not equal to s) sources/destinations and parents/children of d as follows. (1) d_{Y} (d_{N}) transmits Y (N) to destinations d' of d if $\mathsf{ann}(d, d') = \emptyset$; otherwise, it transmits $\mathsf{ann}(d, d')$. This rule connects d_{Y} and d_{N} with all DTD parents (children) of d in the case of bottom-up (top-down) propagation. (2) d_{Y} (d_{N}) has the same set of children as an element type d had. This rule connects d_{Y} and d_{N} with all children in the case of any kind of propagation. (3) After application of steps (1) and (2), the only connection with parents $p \neq s$ in the case of top-down propagation is not defined. Here, d_{Y} (d_{N}) which is the initial d (the clone of d) should be connected with sources of generators transmitting Y or nothing (N and nothing more).

An algorithmic description of the procedure of removing qualifiers is depicted in Fig. 2. Lines 5, 6, 7 represents the rules for connecting d_{Y} and d_{N} with non-equal to s sources/destinations and parents/children of d.

Having removed qualifiers, we can define SSIs. Obviously, for the top-down propagation, SSI contains only one generator (parent-child DTD edge), and the number of SSIs is equal to the number of parents in DTD graph. However, the

[3] Otherwise, the policy is inconsistent

[4] In the same way we may introduce function $parent(s, d)$ that returns parent element type w.r.t. DTD structure for a pair (s, d)

Algorithm: QUALIFIER REMOVING
Input: Partially annotated DTD with qualifiers
Output: Partially annotated DTD without qualifiers

1: **for** every generator (s, d) such that $\mathsf{ann}(s, d) = \mathsf{Q}[q]$ **do**
2: Create element types d_{Y} and d_{N};
3: In $s \rightarrow P(s)$, substitute d for $d_{\mathsf{Y}} + d_{\mathsf{N}}$;
4: Set
$\sigma(parent(s, d_{\mathsf{Y}}), child(s, d_{\mathsf{Y}})) = child(s, d)[q]; \sigma(parent(s, d_{\mathsf{N}}), child(s, d_{\mathsf{N}})) = child(s, d)[\neg q];$
$\mathsf{ann}(s, d_{\mathsf{Y}}) = \mathsf{Y};\ \mathsf{ann}(s, d_{\mathsf{N}}) = \mathsf{N};$
5: Connect d_{Y} and d_{N} with all destinations d' of d:
$\sigma(parent(d', d_{\mathsf{Y}}), child(d', d_{\mathsf{Y}})) = \sigma(parent(d', d), child(d', d)) = \sigma(parent(d', d_{\mathsf{N}}), child(d', d_{\mathsf{N}}))$
$\mathsf{ann}(d_{\mathsf{Y}}, d') = \mathsf{ann}(d, d') = \mathsf{ann}(d, d'),\ if\ \mathsf{ann}(d, d') \neq \emptyset;$
$\mathsf{ann}(d_{\mathsf{Y}}, d') = \mathsf{Y}; \mathsf{ann}(d_{\mathsf{N}}, d') = \mathsf{N},\ if\ \mathsf{ann}(d, d') = \emptyset;$

// After step 5, the next step has the meaning only for *bottom-up* policy class
6: Connect d_{Y} and d_{N} with all DTD children $ch \neq s$ of d setting:
$\sigma(d_{\mathsf{Y}}, ch) = \sigma(d, ch) = \sigma(d_{\mathsf{N}}, ch);$
$\mathsf{ann}(ch, d_{\mathsf{Y}}) = \mathsf{ann}(ch, d) = \mathsf{ann}(ch, d_{\mathsf{N}});$

// After step 5, the next step has the meaning only for *top-down* policy class
7: Connect d_{Y} (d_{N}) with other parents $p \neq s$ of d such that $\mathsf{ann}(p, d) = \mathsf{Y}|\emptyset$ $(\mathsf{ann}(p, d) = \mathsf{N})$ setting:
$\sigma(p, d_{\mathsf{Y}}) = \sigma(p, d)\ (\sigma(p, d_{\mathsf{N}}) = \sigma(p, d));$
$\mathsf{ann}(p, d_{\mathsf{Y}}) = \mathsf{ann}(p, d)\ (\mathsf{ann}(p, d_{\mathsf{N}}) = \mathsf{ann}(p, d));$

Fig. 2. Algorithm QUALIFIER REMOVING

situation is more complicated for the bottom-up policy. First of all, every destination element type d may have several children transmitting their security labels to d. Secondly, the number of SSIs and their components depend on the presence of choices $(\alpha + \alpha)$ in $P(A)$ (see Def. 3). The intuition is the following: we present every sequence (α, α) of $P(A)$ as a conjunction $(\alpha \wedge \alpha)$, and every choice $(\alpha + \alpha)$ of $P(A)$ as disjunction $(\alpha \vee \alpha)$ in parenthesis. From the introduced logical expression, we construct formula Δ by removing parenthesis. The number of SSIs and their configuration is, respectively, the number of disjuncts and configuration of conjuncts in every disjunct in Δ. For example, logical representation of production rule $A \rightarrow ((B|C), D)$ is $A = (B \vee C) \wedge D$ which has the following view after parenthesis removing: $B \wedge D \vee C \wedge D$. Therefore, in the case of bottom-up propagation, A has two SSIs: $\{B, D\}$ and $\{C, D\}$.

Next, for every SSI, we calculate the PFA using VC option if necessary.

Definition 10. *We say that* $\mathsf{ann}_{\mathsf{data}}(d)$ *is* steady *if for every* $\overline{G}(d)$*,* $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G}(d)}$ *are the same and not empty. Otherwise,* $\mathsf{ann}_{\mathsf{data}}(d)$ *is* alternating.

An alternating annotation means that d may obtain different annotations depending on the SSI at the XML level, while a steady annotation for d means that d always has the same label wherever d occurs in XML document. To deal with alternating annotations, we split node as in QUALIFIER REMOVING connecting d_{Y} (d_{N}) with SSI of generators transmitting Y (N).

Definition 11. *We say that destination type d, such that* $\mathsf{ann}_{\mathsf{data}}(d) \neq \emptyset$*, is* closed *if for every destination type* $d' \in P_{\mathsf{ann}}(d)$*,* $\mathsf{ann}(d, d') \neq \emptyset$*. Otherwise, d is* open *and for* $\forall d' \in P_{\mathsf{ann}}(d)$ *such that* $\mathsf{ann}(d, d') = \emptyset$*, d retransmits annotation* $\mathsf{ann}_{\mathsf{data}}(d)$ *to d' via generator (d, d'). Thus, we rename d as s and d' as d.*

We assume that every initially annotated DTD element type e (e.g., *root* or all leaves for bottom-up propagation) automatically retransmits its annotation to all generators $g = (e, d')$ such that $\mathsf{ann}(g) = \emptyset$.

Algorithm: Split
Input: DTD element type d having generators with different annotations
Output: d_{N}
1: Create element types d_{Y} and d_{N};
2: **for** every SSI $\overline{G_k}(d)(k = \overline{1,n})$ having sources $\{s_1, \ldots, s_{m_k}\}$ and resulting in a PFA Y (N) of d **do**
3: Connect source s_i of every generator $g_i \in \overline{G_k}(d), i = \overline{1, m_k}$ with d_{Y} (d_{N}) setting:
$\sigma(parent(s_i, d_{\mathsf{Y}}), child(s_i, d_{\mathsf{Y}})) = child(s_i, d) = \sigma(parent(s_i, d_{\mathsf{N}}), child(s_i, d_{\mathsf{N}}))$
$\mathsf{ann}(s_i, d_{\mathsf{Y}}) = \mathsf{ann}(s_i, d)(= \mathsf{Y}); \mathsf{ann}(s_i, d_{\mathsf{N}}) = \mathsf{ann}(s_i, d)(= \mathsf{N})$;
4: **for** every generator $g' = (d, d')$ where d is a source **do**
5: Connect d_{Y} and d_{N} with d' setting:
$\sigma(parent(d', d_{\mathsf{Y}}), child(d', d_{\mathsf{Y}})) = child(d', d) = \sigma(parent(d', d_{\mathsf{N}}), child(d', d_{\mathsf{N}}))$
$\mathsf{ann}(d_{\mathsf{Y}}, d') = \mathsf{ann}(d, d') = \mathsf{ann}(d_{\mathsf{N}}, d')$;
6: return d_{N};

Fig. 3. Algorithm Split

Algorithm: Annotate View
Input: Partially annotated annotated DTD D
Output: Fully annotated DTD
1: Preprocessing;
2: Qualifier Removing;
3: Create empty *queue*, initialize it with all DTD element types;
4: **while** *queue* is not empty **do**
5: d :=Dequeue($queue$);
6: **if** $\mathsf{ann}_{\mathsf{data}}(d) = \emptyset$ **then**
7: **if** d is not *expecting* **then**
8: Calculate SSIs $\left\{\overline{G_1}(d), \overline{G_2}(d), \ldots, \overline{G_n}(d)\right\}$;
9: **for** every $\overline{G_i}(d)$ **do**
10: Calculate $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G_i}(d)}$ (applying *value conflict resolution* policy option if not for all $g \in \overline{G_i}(d)$ $\mathsf{ann}(g)$ is the same);
11: **if** $\mathsf{ann}_{\mathsf{data}}(d)$ is *steady* **then**
12: Assign any $\mathsf{ann}_{\mathsf{data}}(d)_{\overline{G_i}(d)}$ to $\mathsf{ann}_{\mathsf{data}}(d)$;
13: **else if** $\mathsf{ann}(d)$ is *alternating* **then**
14: d_{clone} :=Split(d);
15: Enqueue($queue, d_{clone}$);
16: **if** d is not splitted and d is *open* **then**
17: For every $d' \in P_{\mathsf{ann}}(d)$ such that $\mathsf{ann}(d, d') = \emptyset$, set $\mathsf{ann}(d, d') = \mathsf{ann}_{\mathsf{data}}(d)$;
18: **else**
19: Enqueue($queue, d$);

Fig. 4. Algorithm Annotate View

The generic algorithm Annotate View is shown in Fig 4. It starts with a *preprocessing* procedure which is needed only for the local policy to define and apply a default labeling for non-annotated generators. After the preprocessing and qualifier removing steps, we invoke labeling iterations via *queue* [10]: if the next considered element type d has a full annotation $\mathsf{ann}_{\mathsf{data}}(d)$, there is no need to process it; otherwise, the **if** clause at line 2. If all generators of d have a defined annotation, then $\mathsf{ann}_{\mathsf{data}}(d)$ is defined. If not, place d back to *queue* (step 19), thus delaying definition of a full annotation of d (i.e. d is *expecting*).

Finally, we remove the N-labeled nodes from the fully annotated DTD. This algorithm is identical to that in [1].

Example 2. The left part of Fig. 5 represents an initial annotation of a DTD schema. We use top-down propagation to obtain a full annotation which is shown on the central part of Fig. 5. In particular, solid and dashed lines are generators

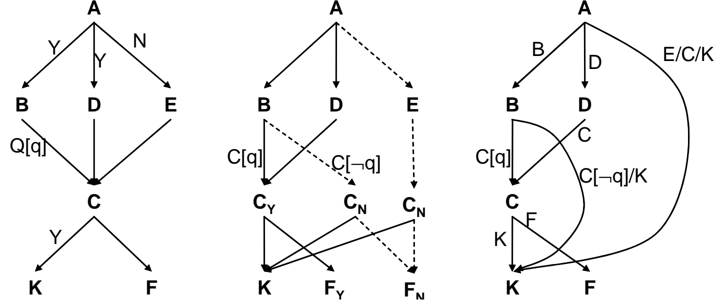

Fig. 5. View construction example

transmitting Y and N respectively; labels on (B, C_{Y}) and (B, C_{N}) generators are corresponding σ-functions; for the other generators, $\sigma(x, y) = y$. Finally, the right part of Fig. 5 is the DTD after deletion of N-labeled nodes. Labels on edges represent corresponding σ-functions.

4 Query Rewriting Algorithm Description

In this section we show the algorithm for query rewriting. The query language is that of the CoreXPath of Gottlob et al. [11] augmented with the union operator and atomic tests and which is denoted by Benedict et al. [12] as $\mathcal{X}$.

Definition 12. *An XPath expression in $\mathcal{X}$ is defined by the following grammar:*

$$\begin{aligned}
\langle path \rangle &::= \langle step \rangle \, (\text{`}/\text{`} \, \langle remaining\ path \rangle)? \\
\langle remaining\ path \rangle &::= \langle path \rangle \\
\langle step \rangle &::= \theta(\text{`}[\text{`} \, \langle qual \rangle \, \text{`}]\text{`})* \ \mid \ \langle path \rangle \, \text{`} \cup \text{`} \, \langle path \rangle \\
\langle qual \rangle &::= A \ \mid \ \text{`} * \text{`} \ \mid \ op\ c \ \mid \\
&\quad\ \ \langle qual \rangle \ and \ \langle qual \rangle \ \mid \ \langle qual \rangle \ or \ \langle qual \rangle \ \mid \\
&\quad\ \ not \ \langle qual \rangle \ \mid \ \text{`}(\text{`} \, \langle qual \rangle \, \text{`})\text{`} \ \mid \ \langle path \rangle
\end{aligned}$$

where θ stands for an XPath step specification (axis :: label, where label is either label A or symbol $*$), c is a `str` *constant, op stands for one of $=$, $\neq$, $<$, $>$, $\leq$, $\geq$; qual is called* qualifier *(or* filter*) and is denoted by q.*

The algorithm for query rewriting has two phases: query parsing and further translation of the parsed query into σ-functions. Query parsing phase implies that user query is represented as a tree of subqueries (*parse tree*) according to the grammar that we have shown in Def. 12.

The translation of the parsed query starts from the leaves of the parse tree and moves up to the root $\langle path \rangle$. In particular, for each subquery p and an element A, the algorithm calculates $QR_{(p,A)}$ using Query Rewrite(p_i, B_j), where p_i is a direct subquery (child in a parse tree) of p and B_j is a node reachable from A via p_i in D_v. At the same time, the algorithm calculates $reach(p, A)$ representing the set of nodes reachable from node A via the path p. To obtain a rewriting of the initial user query q, we invoke Query Rewrite(q, *root*).

Algorithm: QUERY REWRITE
Input: a subquery q (as a parsed XPath expression), a node A for which query rewriting is carried
Output: rewritten subquery q w.r.t. A node

1: **if** q is $\langle path \rangle$ **then**
   // $q = q_1/q_2$ where $q_1 = firstStep, q_2 = remainingSteps$
2: $QR_{(q,A)}$:=QUERY REWRITE(q_1, A)/$\bigcup_{v \in reach(q_1,A)}$QUERY REWRITE($q_2, v$);
3: $reach(q_1/q_2, A) := reach(q_1/q_2, A) \cup \bigcup_{v \in reach(q_1,A)} reach(q_2, v)$;
4: **else if** q is a union of paths $\langle path \rangle$ **then**
   // $q = p_1 \cup p_2 \cup \ldots \cup p_n$
5: $QR_{(q,A)} := \bigcup_{p_i}$ QUERY REWRITE(p_i, A);
6: $reach(q, A) = reach(q, A) \cup reach(p_i, A)$;
7: **else if** q is $\theta[\langle qual \rangle]$ **then**
   // $q = q_0[filter_1] \ldots [filter_n]$ where q_0 is $nodeTest$
8: $QR_{(q,A)}$:=QUERY REWRITE(q_0, A) $\bigcup_{filter_i} [\bigcup_{v \in reach(q_0,A)}$QUERY REWRITE($filter_i, v$)];
9: $reach(q, A) := reach(q_0, A)$;
10: **else if** q is $\langle qual \rangle$ **then**
   // $q = \langle qual \rangle$ from Def. 12
11: **if** q has no operands **then**
12: $QR_{(q,A)}$:=QUERY REWRITE(q, A); // q is $\langle path \rangle$
13: **else if** q has one (*not*) operand **then**
14: $QR_{(q,A)}$ = *not* QUERY REWRITE(q_0, A); // where q_0 is the operand;
15: **else if** q has two operands **then**
   // q_1 is the first operand, q_2 is the second operand; op_2 is one of *and*, *or*, =, ≠, ≥, ≤;
16: $QR_{(q,A)}$:=QUERY REWRITE(q_1, A) op_2 QUERY REWRITE(q_2, A);
17: **else if** q is θ **then**
   // θ=$axis$ :: $label$
18: **if** $axis$ is '*child*' or '*parent*' **then**
19: $QR_{(q,A)}$:=processChildParent($label$,$axis$,A);// (Fig. 8)
20: **else if** $axis$ is '*descendant-or-self*' or '*ancestor-or-self*' **then**
21: $QR_{(q,A)}$:=processDescendAncest($label$,$axis$,A);//(Fig. 7)
22: **else if** $axis$ is '*attribute*' **then**
23: **if** A has attribute $label$ **then**
24: $QR_{(q,A)} := q$;
25: **else if** (q is literal) or (q is number) **then**
26: $QR_{(q,A)} = q$;
27: return $QR_{(q,A)}$;

Fig. 6. Algorithm QUERY REWRITE

The algorithm presented in Fig. 6 shows the translation procedure. Lines 1, 4, 7, 10, 17 distinguish whether the subexpression is $\langle path \rangle$, union of steps, node with qualifiers $\theta[\langle qual \rangle]$, qualifier $\langle qual \rangle$, and node test θ respectively.

The translation of $\langle path \rangle$, union of steps, $\theta[\langle qual \rangle]$, and $\langle qual \rangle$ is quite straightforward, and we concentrate on a processing of node test θ. Since θ has a general form $axis :: label[filter_1] \ldots [filter_n]$, we have the following possibilities for rewriting:

1. *label* is a child of A in D_v: $rewrite(q_1, A) = \sigma(A, label)$;
2. *label* is a parent of A: $rewrite(q_1, A) = \sigma^{-1}(label, A)$ (the definition of $\sigma^{-1}(B, A)$ goes below);
3. *label* is a descendant (ancestor) of A: all the paths from A to *label* (from *label* to A) in D_v should be rewritten w.r.t. σ (σ^{-1})-function.

We introduce two auxiliary functions: *processChildParent* (that captures possibilities 1 and 2) in Fig. 8 and *processDescendAncest* (handling possibility 3) in Fig. 7. The symbol $\downarrow *$ ($\uparrow *$) is used to denote the subquery q = *descendant-or-self* (*ancestor-or-self*).

Algorithm: processDescendAncest
Input: node *label* $\in \{str, *\}$, node *axis* $\in$ {*descendant-or-self, ancestor-or-self*}, element node A of D_v
// $p = axis::label$; $reach(\downarrow *, A)$ ($reach(\uparrow *, A)$) contain all descendants (ancestors) of A;
1: **if** $axis$ = *descendant-or-self* **then**
2: q :='↓∗';
3: **else if** $axis$ = *ancestor-or-self* **then**
4: q :='↑∗';
5: $res := \bigcup_{label \in reach(q,A)} preRewrite(q, A, label)$;
6: $reach(p, A) := reach(p, A) \cup \{label \in reach(q, A)\}$
7: return res;

Fig. 7. Algorithm processDescendAncest

Algorithm: processChildParent
Input: node *label* $\in \{str, *\}$, node *axis* $\in$ {*child, parent*}, node A of D_v
// $q = axis::label$
1: $reach(q, A)$ is a set of nodes that are in relation $axis$ with A;
2: $res := \bigcup_{label \in reach(q,A)} \max \{\sigma(A, label), \sigma^{-1}(A, label)\}$;
3: return res;

Fig. 8. Algorithm processChildParent

For rewriting of descendant/ancestor relations, we use the data of the statically precomputed table *preRewrite* which contains all the rewritten paths from A to all B descendants/ancestors. We use a simple deep-first-search algorithm to find the union u of all paths p_i from A to any B. After that, every p_i of u is rewritten by a call of Query Rewrite(p_i, A). Thus, $preRewrite(\downarrow*(\uparrow*), A, B)$ is a union of rewritten paths p_i.

Now, consider the meaning of $\sigma^{-1}(B, A)$. $\sigma(B, A)$ is a collection of paths from B to A in the initial DTD D such that B is a parent of A in D_v: $\sigma(B, A) := \cup_{i=1}^{k} p_i$ where each

$$p_i = c_{i_1}[f_{i_1}]/c_{i_2}[f_{i_2}]/\ldots/c_{i_{n_i-1}}[f_{i_{n_i-1}}]/c_{i_{n_i}}[f_{i_{n_i}}]$$

with c_{i_1} as the child of B (since p_i is applied to B), $c_{i_{n_i}}$ is A, each c_{i_j} is a child of $c_{i_{j-1}}$, f_{i_j} is a filter expression for the node c_{i_j}. Then $\sigma^{-1}(B, A)$ is defined as follows:

Definition 13. *The reversed representation* $\sigma^{-1}(B, A)$ *of non empty* $\sigma(B, A)$ *is* $\sigma^{-1}(B, A) := \cup_{i=1}^{k} {p_i}^{-1}$ *where each*

$$\begin{aligned} {p_i}^{-1} = &\ self :: c_{i_{n_i}}[f_{i_{n_i}}]/parent :: c_{i_{n_i-1}}[f_{i_{n_i-1}}]/\ldots \\ &/parent :: c_{i_1}[f_{i_1}]/parent :: B[preRewrite(\uparrow *, B, root)] \end{aligned}$$

is applied to the node A.

An expression $self::c_{i_{n_i}}[f_{i_{n_i}}]$ ensures that $\sigma^{-1}(B, A)$ is applied to a *permitted* A node. Analogously, $parent::B$ is filtered by $[preRewrite(\uparrow*, B, root)]$ to guarantee that the user cannot reveal any additional information like in the following example:

Example 3. Suppose, D_v contains a fragment $B \to (C, A), C \to (D, A)$; the user has an access to B and A, but C and D are visible only under some condition

Q_C. The query $//A/parent::C/D$ may leak sensitive information if we do not restrict C with an expression $[Q_C]$.

The algorithm of $\sigma^{-1}(B, A)$ calculation directly follows from Def. 13. We do not show it here for the lack of the space.

The algorithm *processChildParent* is presented in Fig. 8. The expression $\max\{\sigma(A,B), \sigma^{-1}(B,A)\}$ in line 2 selects the non-empty element from $\sigma(A, B)$ and $\sigma^{-1}(B, A)$ (one of them is always empty, while the other is not).

Example 4. We use the security view of Example 2 to demonstrate our query rewriting algorithm.

Path rewriting : $A/B/K \to A/\sigma(A,B)/\sigma(B,K) \to A/B/C[\neg q]/K$;
Descendant rewriting : $A//C \to A/preRewrite(\downarrow *, A, C) \to A/(B/C \cup D/C) \to$
$\to A/(\sigma(A,B)/\sigma(B,C) \cup \sigma(A,D)/\sigma(D,C)) \to A/(B/C[q] \cup D/C)$;
Filter rewriting : $A[B]/K \to A[\sigma(A,B)]/\sigma(A,K) \to A[B]/E/C/K$;
Usage of σ^{-1} : $A/B//K/parent::C \to A/\sigma(A,B)/prerewrite(\downarrow *, B, K)/\sigma^{-1}(C,K) \to$
$\to A/B/(C[q]/K \cup C[\neg q]/K)/self::K/parent::C[preRewrite(\uparrow *, C, A)] \to$
the latter filter is rewritten as follows
$preRewrite(\uparrow *, C, A) \to$ Query Rewrite$(parent::B/parent::A \cup parent::D/parent::A, C) \to$
$\to \sigma^{-1}(B,C)/\sigma^{-1}(A,B) \cup \sigma^{-1}(D,C)/\sigma^{-1}(A,D)$
The last expression is calculated according to Def. 13.

The closest approach to query rewriting is presented by Fan et al. in [8]. The main differences are: the algorithm derives a security view without any dummy element types which may be a source of sensitive information leakage. Therefore, the σ-function used in our query rewriting has different semantics. An extended XPath fragment has *parent* and *descendant-or-self* axes. Finally, Fan et al. use dynamic programming so that $QR_{(q,A)}$ is calculated for *every* DTD element type A; while we perform a rewriting of q w.r.t. to a *subset of relevant element types A of DTD* in a recursive manner.

5 Related Work and Conclusion

The mapping between existing policy frameworks and our proposal is summarized in Table 2. Note that our Y(N) label corresponds to "grant" ("deny"), + (−) of other models. The comparison of some other access control parameters is shown in Table 3.

The provisional access control model for XML documents [6] considers both top-down and bottom-up propagation. Since the rule *most specific takes precedence* (MSTP) is not used, arising conflicts are resolved by the VC option. In the case of the presence of unresolved conflicts or unlabeled nodes, a special *default* option (Y or N) is applied. The policy is evaluated at the stage of query answering (we marked it in Table 3 as "policy evaluation" (PE)). If access to the requested node is permitted, the user receives "weak" XML view, where N-labeled nodes having Y-children are revealed without their attributes.

Table 2. Existing policy frameworks

Method	HP	LP	SC	VC	MSTP	default
Kudo et al. [6]	$*$	none	hf	$*$	No	Yes ($\neq$none)
Murata et al. [13]	td ind.	$*$ ind.	none	dtp	No	Yes (closed)
Gabillon et al. [5]	td ind.	$*$ ind.	none	priority, order	No	Yes ($\neq$none)
Damiani et al. [3]	td ind.	$*$ ind.	none	dtp	Yes	Yes (closed)
Bertino et al. [4]	td ind.	$*$ ind.	none	dtp	Yes	Yes (closed)
Cho et al. [14]	td ind.	$*$ ind.	none	dtp	No	Yes (open)
Fan et al. [8]	td	none	$\neq$lf	none	Yes	No
Our method	$*$	$*$	$*$	$*$	Yes	No

Table 3. Existing XML access control frameworks

Method	XML view	DTD view	Query asking over	Query answering by
[6]	Yes (weak)	No	initial XML	policy evaluation (PE)
[13]	No	No	initial XML/DTD	rewriting, PE
[5]	Yes (weak)	No	XML view	XPath evaluation
[3]	Yes (weak)	Yes (loosened)	XML/DTD view	XPath evaluation
[4]	Yes	No	XML view	XPath evaluation
[14]	No	No	initial XML/DTD	rewriting, PE
Stoica et al. [7]	No	Yes	-	-
[8]	No	Yes	DTD view	safe rewriting
Our method	Yes	Yes	DTD/XML view	safe rewriting

The proposal of Murata et al. [13] defines for every XML node an individual label which is propagated either in hierarchical (td) or in local manner (in the Table 2, it is marked as "ind."). An additional *denial downward consistency* (DDC) policy is introduced: a subtree rooted at N-labeled node has only N-labeled descendants. The method rewrites (using DTD if exists) the query q posed over the initial XML/DTD so that to minimize the need in PE during query answering.

Gabilon et al. [5] differs from [13] in that the VC resolution is based on a *priority* associated with every access control rule. In the case of multiple rules with the same priority, the last rule in *XML Authorization Sheet* (i.e., the list of access rules) is elected. In this method, PE is used to construct an XML view. Thus, the query evaluation is simply an XPath evaluation over the XML view.

Damiani's et al. [3] proposal partially annotates the DTD schema or an XML instance by Y|N labels which are, then, propagated. The notions of *soft* and *hard* authorizations define the precedence of DTD label over XML label and viceversa. However, this subtlety can be captured by qualifiers of our proposal. Next, the fully annotated schema/instance is pruned. The method considers the XML view in "weak" form and the DTD view in "loosened" form, i.e., every forbidden element has cardinality "optional".

A similar policy enforcement mechanism is used in Author-X system [4]. The differences with the previous approach are (i) the absolute precedence of DTD labels over XML labels, (ii) the presence of an additional option ONE_LEVEL of the HP (td), and (iii) the construction of an XML view without any N-labeled nodes. However, the DTD view is not available for user.

In [14], the DTD/XML is annotated by *mandatory* (i.e., all instances of the element in XML tree *must* specify their security level, DTD may specify a default value), *optional* (instances *may* specify their security level) and *forbidden* (instance labeling is inherited or Y) labels. We treat this framework in the following way: mandatory (optional) specification is Y|N ($*$, respectively) label with local propagation, forbidden specification is a label defined via HP (td) propagation, and the default policy is "open". Like in [13], the approach minimizes the need in policy evaluation during query answering by a special query rewriting.

The method of Stoica et al. [7] takes as an input a fully annotated DTD from which the DTD view is derived. There is no discussion of policy propagation and conflict resolution in [7], thus we do not introduce this method in Table 2. In addition, neither XML view nor query evaluation is considered.

The proposal of Fan et al. [8] has the similar notion of the initial DTD annotation as in this paper. However, the range of policies is restricted to the *top-down* policy. Moreover, the paper does not consider XML view construction, but discusses a *safe query rewriting* when the query over the DTD view is translated into the equivalent query over the initial DTD and the rewritten query is evaluated over the initial XML data. Hence, policy evaluation is used for DTD view construction instead of query answering. Furthermore, the safe rewriting of queries excludes system answers "access denied" which are presented, for example, in [6], [13], [14]. Thus, the information leakage is diminished.

As it may be seen from Table 3, our proposal comprises both the XML view (not "weak") and the DTD view (not "loosened"). Moreover, we use safe query rewriting to eliminate denial of service. In addition, we provide an extended range of policy classes (see Table 2). However, there are directions for future work. First of all, we plan to investigate the compatibility of our proposal with others. For example, one of the closest policy framework is that of Kudo et al. [6]. Hence, we plan to investigate whether XML security views can be integrated into Provisional Authorization Architecture. Secondly, XML access control models of [13], [5], [3], [4] showed a possibility of an individual policy configuration, i.e. for every node. Next, in this paper, we have introduced a notion of generator, i.e., a DTD edge which, in essence, may by generalized to a path in an undirected graph isomorphic to a given DTD graph. However, we leave this generalization for future work as well. Finally, an extended experimental evaluation is required.

Acknowledgments

I would like to thank Gabriel Kuper and Fabio Massacci for encouragement and many useful discussions, and Gabriel Kuper, in particular, for checking my English.

References

1. Kuper, G., Massacci, F., Rassadko, N.: Generalized XML security views. In: SACMAT '05: Proceedings of the tenth ACM symposium on Access control models and technologies, New York, NY, USA, ACM Press (2005) 77–84
2. Lunt, T.F., Denning, D.E., Schell, R.R., Heckman, M., Shockley, W.R.: The SeaView security model. IEEE Trans. Softw. Eng. **16**(6) (1990) 593–607
3. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: A fine-grained access control system for xml documents. ACM Trans. Inf. Syst. Secur. **5**(2) (2002) 169–202
4. Bertino, E., Braun, M., Castano, S., Ferrari, E., Mesiti, M.: Author-X: A Java-based system for XML data protection. In: Proceedings of the IFIP TC11/WG11.3 Fourteenth Annual Working Conference on Database and Application Security, Deventer, The Netherlands, The Netherlands, Kluwer, B.V. (2001) 15–26
5. Gabillon, A., Bruno, E.: Regulating access to XML documents. In: Proceedings of the IFIP TC11/WG11.3 fifteenth annual working conference on Database and application security, Norwell, MA, USA, Kluwer Academic Publishers (2002) 299–314
6. Kudo, M., Hada, S.: XML document security based on provisional authorization. In: CCS '00: Proceedings of the 7th ACM conference on Computer and communications security, New York, NY, USA, ACM Press (2000) 87–96
7. Stoica, A., Farkas, C.: Secure XML views. In: Proceedings of the IFIP TC11/WG11.3 Sixteenth International Conference on Data and Applications Security. Volume 256., Kluwer (2003) 133–146
8. Fan, W., Chan, C.Y., Garofalakis, M.: Secure xml querying with security views. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data, New York, NY, USA, ACM Press (2004) 587–598
9. Samarati, P., De Capitani di Vimercati, S.: Access control: Policies, models, and mechanisms. In: FOSAD '00: Revised versions of lectures given during the IFIP WG 1.7 International School on Foundations of Security Analysis and Design on Foundations of Security Analysis and Design, London, UK, Springer-Verlag (2001) 137–196
10. Cormen, T.H., Stein, C., Rivest, R.L., Leiserson, C.E.: Introduction to Algorithms. McGraw-Hill Higher Education (2001)
11. Gottlob, G., Koch, C., Pichler, R.: Efficient algorithms for processing XPath queries. ACM Trans. Database Syst. **30**(2) (2005) 444–491
12. Benedikt, M., Fan, W., Kuper, G.M.: Structural properties of XPath fragments. In: ICDT '03: Proceedings of the 9th International Conference on Database Theory, London, UK, Springer-Verlag (2002) 79–95
13. Murata, M., Tozawa, A., Kudo, M., Hada, S.: XML access control using static analysis. In: CCS '03: Proceedings of the 10th ACM conference on Computer and communications security, New York, NY, USA, ACM Press (2003) 73–84
14. Cho, S., Amer-Yahia, S., Lakshmanan, L., Srivastava, D.: Optimizing the secure evaluation of twig queries. In: VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases. (2002) 490–501