Secure XML Querying with Security Views

Wenfei Fan*
University of Edinburgh & Bell Laboratories
wenfei@inf.ed.ac.uk

Chee-Yong Chan
National University of Singapore
chancy@comp.nus.edu.sg

Minos Garofalakis
Bell Laboratories
minos@research.bell-labs.com

Abstract

The prevalent use of XML highlights the need for a generic, flexible access-control mechanism for XML documents that supports efficient and secure query access, without revealing sensitive information to unauthorized users. This paper introduces a novel paradigm for specifying XML security constraints and investigates the enforcement of such constraints during XML query evaluation. Our approach is based on the novel concept of *security views*, which provide for each user group (a) an XML view consisting of all and only the information that the users are authorized to access, and (b) a view DTD that the XML view conforms to. Security views effectively protect sensitive data from access and potential inferences by unauthorized users, and provide authorized users with necessary schema information to facilitate effective query formulation and optimization. We propose an efficient algorithm for deriving security view definitions from security policies (defined on the original document DTD) for different user groups. We also develop novel algorithms for XPath query rewriting and optimization such that queries over security views can be efficiently answered without materializing the views. Our algorithms transform a query over a security view to an equivalent query over the original document, and effectively prune query nodes by exploiting the structural properties of the document DTD in conjunction with approximate XPath containment tests. Our work is the first to study a flexible, DTD-based access-control model for XML and its implications on the XML query-execution engine. Furthermore, it is among the first efforts for query rewriting and optimization in the presence of general DTDs for a rich class of XPath queries. An empirical study based on real-life DTDs verifies the effectiveness of our approach.

1. Introduction

XML is rapidly emerging as the new standard for data representation and exchange on the Internet. As large corporations and organizations increasingly exploit the Internet as a means of improving business-transaction efficiency and productivity, it is increasingly common to find operational data and other business information in XML format. In light of the sensitive nature of such business information, this also raises the important issue of securing XML content and ensuring the selective exposure of information to different classes of users based on their access privileges.

Specifically, for an XML document T there may be multiple user groups who want to query the same document. For these user groups different *access policies* may be imposed, specifying what elements of T the users are granted access to. The problem of *secure XML querying* is to enforce these access policies. More formally, given a user query p, our goal is to ensure that the evaluation of p over T returns only information in T that the user is allowed to access; in other words, we seek to protect sensitive data from direct access or indirect inference through queries by unauthorized users.

Addressing such security concerns mandates the development of generic, flexible *access-control mechanisms* that can effectively support multiple access policies for controlling access to XML content at various levels of granularity (e.g., restricting access to entire subtrees or specific elements in the document tree based on their content or location). Perhaps even more importantly, enforcing such access-control models should not imply any drastic degradation in either performance or functionality for the underlying XML query-execution engine. Furthermore, access control should not inhibit the availability of necessary schema information (i.e., DTDs) specifying the structure of accessible data. This is a very important requirement, given the crucial role of DTDs in XML data exchange and integration, and in XML query formulation and optimization. In other words, XML security policies must be supported by (a) novel *query optimization and processing techniques* that can ensure *efficient* and *secure* query access to large XML documents for a sufficiently rich and powerful query language (i.e., a significant fragment of XPath [7]); and, (b) the ability to effectively derive and publish a number of different *DTD-schemas* characterizing accessible data based on different access policies.

A number of recent research efforts have considered access-control models for XML data [3, 6, 8, 9, 19, 16, 22, 25]. These models, however, suffer from various limitations. For instance, they may reject proper queries and access [16, 25], incur costly runtime security checks for queries [22], require expensive view materialization and maintenance [8, 9], or complicate integrity maintenance by annotating the underlying data [6]. Perhaps a more subtle problem is that none of these earlier models provides users with a DTD characterizing the information that users are allowed to access. Worse still, some models expose the full document DTD to all users, and make it possible to employ (seemingly secure) queries to infer information that the access control policy was meant to protect; such a situation is illustrated in the following example.

Example 1.1: Consider the DTD represented as a graph in Fig. 1. A hospital document conforming to the DTD consists of a list of departments (`dept`*), and each `dept` has children describing clinical trials, patients, and medical staff (doctors and nurses) in the department. A `staff` member can be either a `nurse` or a `doctor`, indicated by dashed edges; similarly for `treatment`. The patient data is organized into two separate groups depending on whether or not the patients are involved in clinical trials.

Suppose that the hospital wants to impose a security policy that authorizes nurses to access all patient data except for information concerning whether a patient is involved in clinical trials. A plausible solution to enforce this policy is allow nurses to access

*Supported in part by NSF Career Award IIS-0093168, NSFC 60228006 and EPSRC GR/S63205/01.

SIGMOD 2004, June 13-18, 2004, Paris, France.

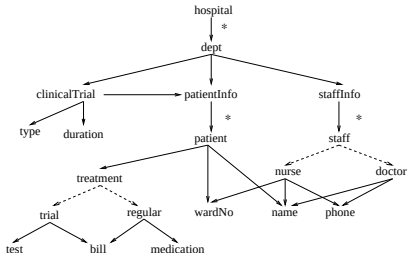


Figure 1: Example Document DTD.

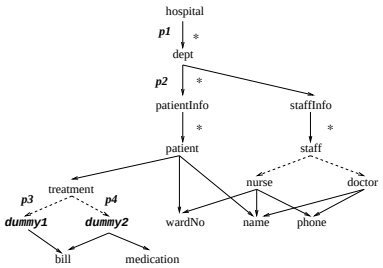


Figure 2: Security view for nurses.

`patientInfo`, while denying their access to `clinicalTrial`, `regular`, `trial`, and `test`; moreover, the document DTD is revealed to nurses so that they can formulate their queries, like [6]. However, even though nurses can not explicitly refer to the `clinicalTrial` element in their queries, it is still possible for them to circumvent this restriction and infer the identity of patients in clinical trials via (multiple) queries by exploiting the DTD structure. Indeed, the "confidential" information can be deduced from the results of the following two permissible queries:

p_1: `//dept//patientInfo/patient/name`,
p_2: `//dept/patientInfo/patient/name`.

The difference between the results of p_1 and p_2 in conjunction with the full DTD tells exactly which patients are in clinical trials. Specifically, while p_2 finds the names of all patients not involved in clinical trials (via the path `hospital/dept/patientInfo`), p_1 actually finds the names of *all patients* (via `hospital/dept/(`$\epsilon \cup$ `clinicalTrial)/patientInfo`). □

Our Contributions. Our first contribution is a novel paradigm for specifying XML security constraints and effectively enforcing such constraints during query processing. Given an XML document T accompanied by a document DTD, we allow multiple access control policies to be declared over T by associating security annotations with element types in the *document DTD*. Our security-specification model supports inheritance and overriding, as well as content-based predicates (in the form of XPath qualifiers). Our approach for enforcing these access control polices is based on the novel notion of *security views*. Abstractly, a security view is a restricted view of the original document and the underlying document DTD that (a) is *automatically derived* based on a given security specification for users, (b) exposes *all and only* necessary schema structure and document content to authorized users, and (c) employs *internal XPath query annotations* in the *view DTD* to describe the access paths to the relevant, accessible parts of the document. While the security view DTD is exposed to authorized users, neither the internal XPath annotations nor the full document DTD is visible. Authorized users can only pose queries over the security view, making use of the exposed view DTD to formulate their queries. Our security-view mechanism guarantees that queries only return data that users are allowed to access, and thus protects sensitive data from access by unauthorized users; furthermore, it also provides *inference* control with respect to a given access policy/security specification, to an extent (inference using constraints and external knowledge is beyond the scope of this paper).

While it is common to enforce secure access via views for traditional databases [5], XML security views introduce new challenges. For instance, it is non-trivial to *construct a sound and complete security view* (i.e., a view that exposes all and only accessible data elements and schema structure) w.r.t. a given security policy. In response to this, our second contribution is a novel quadratic-time algorithm that, given the document DTD and an access-policy specification, *automatically* derives a sound and complete security-view definition (i.e., view DTD plus internal XPath annotations).

Another challenge is that in the presence of multiple access control policies, it is expensive to actually materialize and maintain multiple security views of a large XML document. To avoid the overheads of view materialization and the complexities of view maintenance, our third contribution is a novel *XML query-rewriting algorithm* that transforms an input XPath query p posed over a security view to an equivalent, secure query p_t over the original document. This yields an effective querying mechanism that *completely bypasses view materialization*. Our rewriting algorithm is based on dynamic-programming and runs in low polynomial time in the size of the query and the view DTD; in contrast, earlier rewriting schemes (e.g., based on chasing/unification [26] or direct XPath semantics [13]) require (worst-case) exponential time when dealing with significant XPath fragments [15].

Our fourth contribution is a new *query-optimization* technique that leverages the structural properties of DTDs to optimize the evaluation of the transformed query p_t. Given the inherent difficulty of the problem (simple query containment and equivalence testing in the presence of DTDs ranges from coNP-hard to undecidable [24]), our optimization algorithms are necessarily approximate. Our key technical idea here is to exploit the document DTD in conjunction with an approximate XPath containment test by extending *graph simulation* [20] concepts to our setting. Note that the XPath fragment studied in our work completely subsumes the class of twig queries of [6] and the "purely-conjunctive" XPath queries considered in earlier studies on XPath optimization [2, 27, 18].

Finally, we conduct an experimental study of our secure query evaluation techniques using real-life DTDs. Our results show that our query rewriting and optimization techniques are effective.

Example 1.2: Given the document DTD and access policy of Example 1.1, a security view for nurses can be derived. As depicted in Fig. 2, the view is defined by a view DTD D_v along with XPath annotations p_1 to p_4 (to be elaborated in Example 3.2). The view DTD hides confidential information, e.g., `clinicalTrial`, `trial`, `regular`, `test`. The XPath annotations specify how to extract relevant accessible data from the original document. While the view DTD is provided to the nurses, the XPath annotations are hidden from them. In other words, nurses only see the view DTD D_v and are not aware of the existence of `clinicalTrial`, etc., in the document. (As we discuss later in the paper, the role of the "dummy" nodes in Fig. 2 is to ensure that the semantics of the original DTD is preserved in the exposed view.) The view is not materialized. The nurses may pose a query p on the security view, and the query is efficiently transformed to an equivalent query p_t over the original document via our query rewriting algorithm. The query p_t is further optimized and then executed; the result of p_t consists of only XML elements that the nurses are granted access to. □

In summary, the main contributions of our work include:

- The *first* XML security model based on security views supporting both access control and DTD-schema availability;
- A novel quadratic-time algorithm for deriving a sound and complete security view from a given access control policy;
- A new quadratic-time algorithm for rewriting XPath queries over a view to equivalent queries over the original document;
- Techniques for optimizing queries by taking advantage of DTD constraints and approximate XPath containment; and,
- An empirical study verifying the feasibility and effectiveness of our approach.

Our work is the first to study a flexible, DTD- and XPath-based access-control model for XML and its implications on the XML query-execution engine. Furthermore, it is among the first efforts for query rewriting and optimization in the presence of general DTDs for a rich class of XPath queries.

Related Work. XACML [25] and XACL [16] propose standards for specifying access control policies for general actions not limited to queries. The policies are enforced via a decision procedure that, upon receiving an action request, either grants or denies access. In particular, a user query is rejected if its result contains sensitive data. This enforcement strategy is rather brute-force: in practice, one often wants the query to return the part of the result that the user is authorized to access, instead of being rejected. Instead, [22] proposes to check each element in the result of a query, and return the elements that the user is allowed to access. Also proposed in [22] is a static optimization algorithm for checking whether a query is *safe*, i.e., whether it returns only accessible elements. However, expensive run-time security checks are still required for unsafe queries.

Issues like granularity of access, access-control inheritance, overriding, and conflict resolution have been studied for XML in [3, 8, 9]. In particular, [8, 9] propose to express access policies with XPath queries. The semantics of access control to a user is a specific view of the document determined by the XPath access-control rules. An algorithm is also developed based on tree labeling for computing a particular user view of the data. The enforcement strategy is based on materializing and maintaining views, which can be quite complex and computationally expensive.

A different approach has been explored in [6]. In a nutshell, their access-control model assumes that access annotations are explicitly included in the actual element nodes in the data, whereas nodes in the DTD graph specify "coarse" conditions on the existence of security annotations in corresponding data nodes. Only elements with accessible annotations can appear in the result of a query. Novel optimization algorithms are proposed for finding minimal and safe rewritings of *twig queries* with DTD schema information, in order to minimize redundant security checks and localize the remaining checks at run-time. However, the rewriting and optimization techniques are only applicable to twig queries, which represent only a small fragment of XPath (e.g., no wildcards or union). Furthermore, even though including access annotations directly in the data nodes allows for arbitrarily fine-grained access-control policies, it can also make the definition and maintenance of such policies rather complicated and expensive. As an example, defining a new security policy (e.g., for a new class of users) would mean having to go over the entire database and appropriately annotating all relevant data nodes – this could become a problem, e.g., for massive XML data collections, and worse when updates are considered. Like other previous models, this mechanism does not support schema availability, as illustrated in Example 1.1.

A cryptographic technique has recently been proposed in [19]. It assures that published data is visible to anyone but only understandable to authorized users via enciphering keys. The technique is developed particularly for access control to published XML data, but is not applicable to securing (client) XML queries by a server.

Organization. The remainder of the paper is organized as follows. Section 2 reviews DTDs and XPath. Our access control model and security views are defined in Section 3. Sections 4 and 5 present our query rewriting and optimization algorithms, followed by our experimental study in Section 6. Section 7 concludes the paper.

2. Preliminaries

In this section we briefly review Document Type Definitions (DTDs) and the class of XPath queries considered in this paper.

DTDs. Without loss of generality, we represent a DTD by (Ele, Rg, r), where Ele is a finite set of *element types*; r is a distinguished type in Ele, called the *root type*; Rg defines the element types: for any A in Ele, $Rg(A)$ is a regular expression of the form:

$$\alpha \quad ::= \quad \texttt{str} \mid \epsilon \mid B_1, \ldots, B_n \mid B_1 + \ldots + B_n \mid {B_1}^*$$

where `str` denotes PCDATA, ϵ is the empty word, B_i is a type in Ele (referred to as a *subelement type* of A), and '$+$', ',' and '$*$' denote disjunction, concatenation and the Kleene star, respectively (here we use '$+$' instead of '|' to avoid confusion). We refer to $A \rightarrow Rg(A)$ as the *production* of A. Note that all DTDs can be expressed in this form by introducing new element types (entities).

An XML document (tree) T *conforms to* a DTD D if (1) there is a unique node, the *root*, in T labeled with r; (2) each node in T is labeled 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 they are ordered and their labels are in the regular language defined by $Rg(A)$; and, (4) each text node carries a string value (PCDATA) and is a leaf of the tree. We call T an *instance* of D if T conforms to D.

A DTD D can be represented as a graph, referred to as the *DTD graph* of D. The graph contains a node for each element type A in D, referred to as the A *node*, and the edges depict the parent/child relation. Specifically, for each production $A \rightarrow \alpha$, there is an edge from the A node to the B node for each element type B in α. If $\alpha = B^*$, then the edge has a '$*$' as a label indicating that zero or more B elements can be immediately nested within an A element. If α is a disjunction, then the edges are indicated by dashed lines to distinguish from the case of a concatenation. When it is clear from the context, we shall use the DTD and its graph interchangeably, both referred to as D; similarly for A element type and A node.

For example, Fig. 1 depicts a DTD graph of the hospital DTD. Note that a DTD graph can be a DAG (directed acyclic graph); it may even have cycles if the DTD is *recursive*, i.e., when some A is defined in terms of itself directly or indirectly. In contrast to [6], we consider general DTDs with disjunction and recursion. Attributes are not considered here, but they can be easily incorporated.

XPath queries. We consider a class of XPath [7] queries, referred to as $\mathcal{C}$ and defined as follows:

$$p \quad ::= \quad \epsilon \mid l \mid * \mid p/p \mid //p \mid p \cup p \mid p[q],$$

where ϵ, l and $*$ denote the empty path, a label (in Ele) and a wildcard, respectively; '$\cup$', '$/$' and '$//$' stand for union, *child*-axis and *descendant-or-self*-axis, respectively; and finally, q in $p[q]$ is called a *qualifier* and defined by:

$$q \quad ::= \quad p \mid p = c \mid q \wedge q \mid q \vee q \mid \neg q,$$

where c is a constant, p is as defined above, and '$\wedge$', '$\vee$' and '$\neg$' denote conjunction, disjunction and negation. For $p = p_1/p_2$, if p_2 is $//p_2'$, we write p as $p_1//p_2'$. We also use $\emptyset$ to denote a special query, which returns the empty set over all XML trees, with $\emptyset \cup p$ equivalent to p and $p/\emptyset/p'$ equivalent to $\emptyset$.

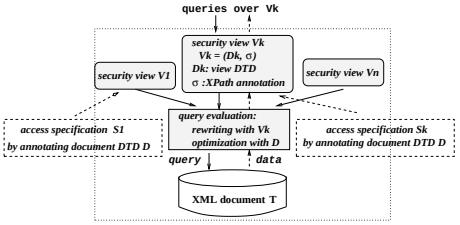


Figure 3: Framework for querying XML via security views.

An XPath query p is evaluated at a *context node* v in an XML tree T, and its result is the set of nodes (or `str` data) of T reachable via p from v, denoted by $v[\![p]\!]$. Qualifiers are interpreted as follows: at a context node v, the atomic predicate $[p]$ holds iff $v[\![p]\!]$ is nonempty, i.e., there exists a node reachable via p from v; and $[p = c]$ is true iff $v[\![p]\!]$ contains a text node whose string value equals the constant c. The boolean operations are self-explanatory.

Our class $\mathcal{C}$ of XPath queries properly contains the twig queries studied in [6]: $\mathcal{C}$ supports wildcard, union, and richer qualifiers with disjunction and negation. As observed by [18, 24], optimization of XPath queries with these operators is far more intriguing than twig queries; among other things, the complexity of XPath containment with the addition of these operators goes up to coNP-hard from quadratic time [27] for twig queries.

3. Access Control with Security Views

In this section we present our view-based security model. We define the concepts of access specifications and security views, and propose an efficient algorithm for automatically deriving a sound and complete security view definition from an access specification.

3.1 Overview of our Access Control Model

Consider an XML document T with a document DTD D. Multiple access control policies are possibly declared over T at the same time, each specifying, for a class of users, what elements in T the users are granted, denied, or conditionally granted access to. We define a language for specifying fine-grained access control policies. An *access specification* S expressed in the language is a simple extension of the document DTD D associating element types with security annotations (XPath qualifiers), which specify structure- and content-based accessibility of the corresponding elements of these types in T. Since we are primarily concerned with querying XML data, i.e., we focus on query as the operation, our specification language adopts a simple syntax instead of the conventional (subject, object, operation) syntax [5, 16, 25].

An access specification S is enforced through an automatically-derived *security view* $V = (D_v, \sigma)$, where D_v is a *view DTD* and σ is a function defined via XPath queries. The view DTD D_v exposes only *accessible* data w.r.t. S, and is provided to users authorized by S so that they can formulate their queries *over the view*. The function σ is transparent to authorized users, and is used to extract accessible data from T. The only structural information about T that the users are aware of is D_v, and no information beyond the view can be inferred from user queries. Thus, our security views support both access/inference control and schema availability. We provide an efficient algorithm that, given a specification S, derives a sound and complete security view definition V, i.e., V characterizing *all* and *only* those accessible elements of T w.r.t. S.

In summary, we propose an access control model based on security views for XML, as depicted in Fig. 3. For each access control policy, a security administrator (or DBA) defines a specification S by annotating the document DTD D (e.g., through a simple GUI tool). For each specification S, a sound and complete security view definition $V = (D_v, \sigma)$ is *automatically derived* by our view-derivation algorithm. The security-view DTD D_v is exposed to users authorized by S so that they can formulate and pose their queries over V. The security view is *virtual*, and a query p over V is efficiently rewritten to an equivalent query p_t over the original document T by incorporating XPath queries in σ; furthermore, our algorithms optimize p_t by exploiting the document DTD D. Finally, the optimized query p_t is executed over T and its result is returned to the users. Note that S, σ, and D are invisible to users. Security issues are handled at the query-rewriting level and are completely hidden from users of the view. This yields a flexible, secure framework for querying XML data that overcomes the limitations of earlier proposals.

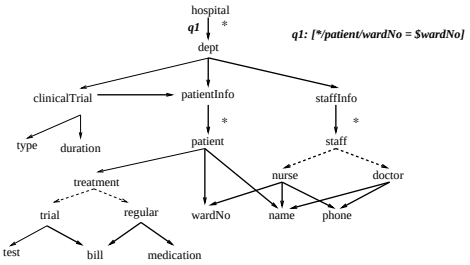


Figure 4: Security specification for nurse

The remainder of this section focuses on access specifications, security views, and our view-derivation algorithm. We study query rewriting and optimization in Sections 4 and 5.

3.2 Security Specifications

An *access specification* S is an extension of a document DTD D associating security annotations with productions of D. Specifically, S is defined to be (D, ann), where ann is a partial mapping such that, for each production $A \to \alpha$ and each element type B in α, $\mathsf{ann}(A, B)$, if explicitly defined, is an annotation of the form:

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

where $[q]$ is a qualifier in our fragment $\mathcal{C}$ of XPath (Section 2). Intuitively, a value of Y, $[q]$, or N for $\mathsf{ann}(A, B)$ indicates that the B children of A elements in an instantiation of D are accessible, conditionally accessible, and inaccessible, respectively. 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 A. The root of D is annotated Y by default.

Example 3.1: An access control policy for nurses over the hospital document of Example 1.1 can be specified as follows:

```
hospital → dept*     /* production */
  ann(hospital,dept) = [*/patient/wardNo=$wardNo]   /*q1*/
dept → clinicalTrial, patientInfo, staffInfo
  ann(dept, clinicalTrial) = N
clinicalTrial → patientInfo
  ann(clinicalTrial, patientInfo) = Y
treatment → trial + regular
  ann(treatment, trial) = N
  ann(treatment, regular) = N
trial → bill
  ann(trial, bill) = Y
regular → bill, medication
  ann(regular, bill) = Y
  ann(regular, medication) = Y
```

This specification is depicted in Fig. 4, where bold edges represent 'Y' or '$[q]$' annotations, while normal edges represent 'N' annotations. Thus, nurses can only access the patient and staff information in a `dept` having a certain ward (restricted by the qualifier q_1), and moreover, that they are not authorized to know which patients are involved in clinical trials as well as the form of `treatment`,

except for `bill` and `medication` information. Observe that `$wardNo` is treated as a constant parameter; i.e., when a concrete value, e.g., 6, is substituted for `$wardNo`, the specification defines the access control policy for nurses working in ward number 6.

Observe that ann(`dept,patientInfo`) is not explicitly defined, which means that the `patientInfo` children of `dept` nodes inherit the accessibility (Y) of `dept`; similarly for `staffInfo` nodes and their descendants. On the other hand, ann(`dept,clinicalTrial`) is defined to be N, which overrides the accessibility (Y) of `dept` and indicates that the `clinicalTrial` children of `dept` nodes are not accessible; similarly for ann(`clinicalTrial,patientInfo`), etc. We omit annotations of productions in which the parent and all its children have the same accessibility, e.g., the one for `staff`. □

For an XML instance T of a DTD D, an access specification $S = (D, \mathsf{ann})$ can be easily defined, e.g., using a simple GUI tool over D's DTD graph. Furthermore, S unambiguously defines the accessibility of document nodes in T. To see this, note that DTD D must be unambiguous by the XML standard [4]. Since T is an instance of D, this implies that each B element v of T has a unique parent A element and a unique production that "parses" the A subtree; thus, we can define v's accessibility $\mathsf{ann}(v)$ to be exactly the $\mathsf{ann}(A, B)$ associated with the production for A. We say that v is *accessible* w.r.t. S if and only if either (1) $\mathsf{ann}(v)$ is Y or $\mathsf{ann}(v)$ is $[q]$ and $[q]$ is true at v, and, moreover, for all ancestors v' of v such that $\mathsf{ann}(v') = [q']$, the qualifier $[q']$ is true at v'; or, (2) $\mathsf{ann}(v)$ is not explicitly defined but the parent of v is accessible w.r.t. S. Note that we require that for v to be accessible, the qualifiers associated with all ancestors of v must be true. Referring to Example 3.1, for a nurse to access the information of a department d, the qualifier q_1 associated with `dept` must be true at d, so that the nurse is prevented from unauthorized access to information of different departments. The accessibility of document nodes is well defined:

Proposition 3.1: *For any specification $S = (D, \mathsf{ann})$ and any XML instance T of D, the accessibility of each node in T w.r.t. S is uniquely defined, i.e., it is either accessible or inaccessible.* □

Observe the following. First, as shown by Example 3.1, our access specifications support *inheritance and overriding*. Indeed, if a node does not have an annotation but its parent is accessible (inaccessible), then the node is accessible (resp., inaccessible); that is, the node's accessibility is inherited from that parent. On the other hand, an explicit annotation at a node may override the accessibility of its parent. Second, *content-based access privileges* are supported via XPath qualifiers. Third, the accessibility of elements in a document is *context-sensitive*: it is determined by the paths from the root to these elements in the document. For example, the `bill` information of a patient is accessible only if the patient is in a `dept` that satisfies the qualifier $[q_1]$. Thus, elements of the same type in a document may have different accessibility. Compared to the access control model of [6], our security-specification model also allows for fine-grained access-control, but is significantly more flexible and easier to specify/modify, since security constraints are defined over DTD-schemas rather than on XML documents.

3.3 Security Views

Syntax. Abstractly, a security view defines a mapping from instances of a document DTD D to instances of a view DTD D_v that is, once again, automatically derived from a given access specification. Let $S = (D, \mathsf{ann})$ be an access specification. A *security view definition* (or simply a *security view*) V from S to a *view DTD* D_v, denoted by $V : S \rightarrow D_v$, is defined as a pair $V = (D_v, \sigma)$, where σ defines XPath query annotations used to extract accessible data from an instance T of D. Specifically, for each production $A \rightarrow \alpha$ in D_v and each element type B in α, $\sigma(A, B)$ is an XPath query (in our class $\mathcal{C}$) defined over document instances of D such that, given an A element, $\sigma(A, B)$ generates its B subelements in the view by extracting data from the document. A special case is the unary parameter usage with $\sigma(r_v) = r$, where r_v is the root type of D_v and r is the root of D, i.e., σ maps the root of T to the root of its view. The view is said to be *recursive* if D_v is recursive.

Example 3.2: Figure 2 depicts a security view V from the access specification of Example 3.1 to a view DTD D_v for nurses. The view DTD removes information about `clinicalTrial`, and introduces "dummy" labels `dummy1,dummy2` to hide the label information of `regular` and `trial`, while retaining the disjunctive semantics at the accessible `treatment` node. Specifically, $V = (D_v, \sigma)$, where σ (computed by our view-derivation algorithm in Fig. 5) is as follows.

production: `hospital` $\rightarrow$ `dept`*
  σ(`hospital,dept`) = `dept`*[*/patient/wardNo=$wardNo]* /*$p_1$*/
production: `dept` $\rightarrow$ `patientInfo`
  σ(`dept,patientInfo`) = p_2
    where p_2 = (`clinicalTrial` $\cup$ ϵ)/`patientInfo`
production: `treatment` $\rightarrow$ `dummy1 + dummy2`
  σ(`treatment, dummy1`) = `trial` /*p_3*/
  σ(`treatment, dummy2`) = `regular` /*p_4*/
production: $A \rightarrow \alpha$ /*for all other productions*/
  $\sigma(A, B) = B$ /*for all $B \in \alpha$ */
    /* e.g., σ(dummy1, bill) = bill */

Recall that ϵ denotes the empty path. The view DTD is provided to the nurses, while the XPath mapping σ is not visible to them. Since the nurses can not see the document DTD, they have no knowledge about what the dummies stand for. □

Semantics. We give the semantics of a security view definition $V : S \rightarrow D_v$ by presenting a materialization strategy for V. We should once again stress that security views are *never materialized* in our access-control framework; the sole purpose of the materialization algorithm described below is to illustrate the semantics of security views. Given an instance T of the document DTD, we build a view of T, denoted by T_v, that conforms to the view DTD D_v and consists of all and only accessible nodes of T w.r.t. S. The computation is top-down: first extract the root of T and treat it as the root of T_v, and then iteratively expand the partial tree by generating the children of current leaf nodes. Specifically, in each iteration, we inspect each leaf v. Assume that the element type of v is A and that the A production in D_v is $P(A) = A \rightarrow \alpha$. The children of v are generated by extracting nodes from T via the XPath annotation $\sigma(A, B)$ for each child type B in α. The computation is based on the structure of production $P(A)$ as follows.

(1) Nothing needs to be done when $P(A)$ is $A \rightarrow \epsilon$.

(2) $P(A) = A \rightarrow$ `str`. Then, the query p defined in σ(`A,str`) is evaluated at context node v in T. If $v[\![p]\!]$ returns a single text node in T that is accessible w.r.t. S, then the text node is treated as the only child of v; otherwise, the computation aborts.

(3) $P(A) = A \rightarrow B_1, \ldots, B_n$. Then, for each $i \in [1, n]$, the query $p_i = \sigma(A, B_i)$ is evaluated at context node v in T. If for all $i \in [1, n]$, $v[\![p_i]\!]$ returns a single node v_i accessible w.r.t. S, then v_i is treated as the B_i child of v; otherwise, the computation aborts.

(4) $P(A) = A \rightarrow B_1 + \ldots + B_n$. Then, for each $i \in [1, n]$, the XPath query $p_i = \sigma(A, B_i)$ is evaluated at context node v in T. If there exists one and only one $i \in [1, n]$ such that $v[\![p_i]\!]$ returns a single node accessible w.r.t. S, then the node is treated as the single child of v; otherwise, the computation aborts.

(5) $P(A) = A \rightarrow B^*$. Then, the query $p = \sigma(A, B)$ is evaluated at context node v in T. All the nodes in $v[\![p]\!]$ accessible w.r.t. S are treated as the B children of v, ordered by the document order of T. Note that, if $v[\![p]\!]$ is empty, no children of v are created.

The construction proceeds until either no leaf node can be further expanded (successful termination), or abortion takes place.

Example 3.3: Given a document T of the hospital DTD shown in Fig. 1, a nurse view T_v of Fig. 2 can be materialized as follows. The root `hospital` of T_v is first created. Then, the annotation p_1 is executed over T, which extracts only the `dept` d that possesses the particular ward. The `staffInfo` subtree of d is copied from T to T_v, while both the `patientInfo` child of d and the `patientInfo` under `clinicalTrial` of d are extracted from T via p_2 at d (note that `clinicalTrial` of T is not copied to T_v). Similarly, the `patient` children of `patientInfo` are generated, and so are the children of each `patient`. At the `treatment` child of each patient in T_v, queries p_3 and p_4 are evaluated to extract either the `bill` child of a `trial` or both `bill` and `medication` children of a `regular`, whereas `trial` and `regular` are mapped to `dummy1` and `dummy2` respectively. Note that the document DTD ensures that either `trial` or `regular` is under the `treatment` in T, but not both. The construction terminates successfully after each `patient` subtree is completed. □

A security view $V = (D_v, \sigma)$ is said to be *sound* and *complete* w.r.t. an access specification $S = (D, \mathsf{ann})$ if for all instances T of the document DTD D, T_v is an XML document that both conforms to the view DTD D_v and consists of all and only those nodes in T that are accessible w.r.t. S.

3.4 Security-View Derivation Algorithm

We now present a novel algorithm (termed derive) that, given an access specification $S = (D, \mathsf{ann})$, automatically computes a security view definition $V = (D_v, \sigma)$ w.r.t. S such that, for any instance T of the document DTD, if the computation of T_v terminates (i.e., does not abort), it comprises all and only accessible elements of T w.r.t. S. In particular, V is sound and complete w.r.t. S if and only if such a view definition exists for S.

Algorithm derive is shown in Fig. 5. In a nutshell, when building $V = (D_v, \sigma)$, the algorithm hides inaccessible nodes in the document DTD D by either *short-cutting* them, or *renaming* them using dummy labels. It uses two procedures, Proc_Acc(S, A) and Proc_InAcc(S, A), to deal with accessible and inaccessible element types A of D, respectively. It traverses the document DTD D top-down by invoking Proc_Acc(S, r), where r is the root element type of D. For each *accessible* element type A encountered, Proc_Acc(S, A) constructs a production $P_v(A) = A \rightarrow \alpha$ in the view DTD D_v, and computes appropriate XPath queries $\sigma(A, B) = p_B$ for each type B in α, based on the A-production in the document DTD D (cases 1 – 4). More specifically, (a) if B is accessible, then p_B is simply 'B' (steps 6,7); (b) if B is conditionally accessible (i.e., $\mathsf{ann}(A, B) = [q]$), then p_B is '$B[q]$', i.e., qualifiers in S are preserved (steps 8,9); and, (c) if B is inaccessible, then derive either prunes the entire inaccessible subgraph below B if B does not have any accessible descendants (step 11), or "shortcuts" B by treating the accessible descendants of B as children of A if this does not violate the DTD-schema form of Section 2 (steps 12–15), or renames B to a "dummy" label to hide the label B while retaining the DTD structure and semantics (steps 16–20). Children of the B node are then processed in the same manner. Again, the key intuition behind the above process is that we want the view DTD D_v to preserve the structure and semantics of the relevant and accessible parts of the original document DTD.

Procedure Proc_Acc(S, A)
Input: specification $S = (D, \mathsf{ann}())$ and an accessible type A in D.
Output: security view $V = (D_v, \sigma)$ for A and its descendants in D.

1. if visited[A, acc] then return else visited[A, acc] := `true`;
2. case the A-production $A \rightarrow \alpha$ in the document DTD D of
3. (1) $A \rightarrow B_1, \ldots, B_n$:
4. $P_v(A) := A \rightarrow B_1, \ldots, B_n$; /*production in the view DTD D_v */
5. for i from 1 to n do
6. if $\mathsf{ann}(A, B_i) = \mathsf{Y}$
7. then $\sigma(A, B_i) := B_i$; Proc_Acc(S, B_i);
8. else if $\mathsf{ann}(A, B_i) = [q]$
9. then $\sigma(A, B_i) = B[q]$; Proc_Acc(S, B_i);
10. else Proc_InAcc(S, B_i);
11. if $\mathsf{reg}(B_i) = \emptyset$ then remove B_i from $P_v(A)$;
12. if $\mathsf{reg}(B_i) = C_1, \ldots, C_k$
13. then replace B_i with $\mathsf{reg}(B_i)$ in $P_v(A)$;
14. for j from 1 to k do
15. $\sigma(A, C_j) := B_i/\mathsf{path}[B_i, C_j]$;
16. else replace B_i with a distinct new label X in $P_v(A)$;
17. add production $X \rightarrow \mathsf{reg}(B_i)$ to the view DTD D_v;
18. $\sigma(A, X) := B_i$;
19. for j from 1 to k do
20. $\sigma(X, C_j)$: = $\mathsf{path}[B_i, C_j]$;
21. (2) $A \rightarrow B_1 + \ldots + B_n$:
22. /* similar to (1), except that for an inaccessible B_i, if $\mathsf{reg}(B_i)$
23. is $C_1 + \ldots + C_k$, then replace B_i with $\mathsf{reg}(B_i)$ in $P_v(A)$ */
24. (3) $A \rightarrow B^*$:
25. /* similar to (1), except that for an inaccessible B_i, if $\mathsf{reg}(B_i)$
26. is C or C^*, then replace B_i with $\mathsf{reg}(B_i)$ in $P_v(A)$ */
27. (4) $A \rightarrow$ `str`:
28. /* similar to (1), except that if $\mathsf{ann}(A,$ `str`$) = \mathsf{N}$, then
29. $P_v(A) := A \rightarrow \epsilon$; */
30. return;

Procedure Proc_InAcc(S, A)
Input: specification $S = (D, \mathsf{ann}())$ and an inaccessible type A in D.
Output: regular expression $\mathsf{reg}(A)$ and $\mathsf{path}[A, C]$ for each C in $\mathsf{reg}(A)$.

1. if visited[A, inacc] then return else visited[A, inacc] := `true`;
2. case the A-production $A \rightarrow \alpha$ in the document DTD D of
3. (1) $A \rightarrow B_1, \ldots, B_n$:
4. $\mathsf{reg}(A) := B_1, \ldots, B_n$; /* analogous to A-production in D_v */
5. for i from 1 to n do
6. if $\mathsf{ann}(A, B_i) = \mathsf{Y}$
7. then $\mathsf{path}[A, B_i] := B_i$; Proc_Acc$(S, B_i)$;
8. else if $\mathsf{ann}(A, B_i) = [q]$
9. then $\mathsf{path}[A, B_i] = B[q]$; Proc_Acc$(S, B_i)$;
10. else Proc_InAcc(S, B_i);
11. if $\mathsf{reg}(B_i) = \emptyset$ then remove B_i from $\mathsf{reg}(A)$;
12. if $\mathsf{reg}(B_i) = C_1, \ldots, C_k$
13. then replace B_i with $\mathsf{reg}(B_i)$ in $\mathsf{reg}(A)$;
14. for j from 1 to k do
15. $\mathsf{path}[A, C_j] := B_i/\mathsf{path}[B_i, C_j]$;
16. else replace B_i with a distinct new label X in $\mathsf{reg}(A)$;
17. add production $X \rightarrow \mathsf{reg}(B_i)$ to the view DTD D_v;
18. $\sigma(A, X) := B_i$;
19. for j from 1 to k do
20. $\sigma(X, C_j)$: = $\mathsf{path}[B_i, C_j]$;
/* similar for productions of other forms */

Figure 5: Algorithm derive

The procedure Proc_InAcc(S, A) processes an inaccessible node A in a similar manner. One difference is that it computes (1) $\mathsf{reg}(A)$ instead of α in the A-production $A \rightarrow \alpha$ in the view DTD D_v, and (2) $\mathsf{path}[A, B]$ for each element type B in $\mathsf{reg}(A)$ rather than $\sigma(A, B)$. Intuitively, $\mathsf{reg}(B)$ is a regular expression identifying all the closest *accessible descendants* of B in D, and $\mathsf{path}[A, B]$ stores the XPath query that captures the paths from A to B in the document DTD. Another difference concerns the treatment of *recursive node* A. If an inaccessible A is encountered again in the computation of Proc_InAcc(S, A), then A is renamed to a

dummy label and retained in the regular expression returned. Retaining the recursive structure of the document DTD is essential to simplifying query rewriting. To simplify the discussion, we omit from Fig. 5 this treatment of recursive inaccessible nodes.

To efficiently compute V, Algorithm derive associates two boolean variables visited[A, acc] and visited[A, inacc] (initially `false`) with each element type A in the document DTD D. These variables indicate whether A has already been processed as an accessible or inaccessible node, respectively, to ensure that each element type of D is processed only once in each case. In light of this, the algorithm takes at most $O(|D|^2)$ time, where $|D|$ is the size of the document DTD.[1] Moreover, one can verify its correctness:

Theorem 3.2: *Given an access specification $S = (D, \mathsf{ann})$, algorithm* derive *computes a sound and complete security view w.r.t. S in quadratic time if and only if such a view exists.* □

Example 3.4: Given the access specification S of Fig. 4, Algorithm derive computes the security view (D_v, σ) of Example 3.2 as follows. It starts by invoking Proc_Acc(S, `hospital`), and finds that `dept` is accessible w.r.t. S and is annotated q_1. Thus it adds production `hospital` $\rightarrow$ `dept`* to the view DTD D_v, and defines σ(`hospital`, `dept`) to be `dept`$[q_1]$. It then proceeds to process `dept`, and encounters inaccessible `clinicalTrial`. Now it invokes Proc_InAcc(S, `clinicalTrial`), which yields reg(`clinicalTrial`) = {`patientInfo`} as well as path[`clinicalTrial`,`patientInfo`] = `patientInfo`. Given these, it shortcuts the inaccessible node and defines the view DTD production `dept` $\rightarrow$ `patientInfo`^{1}, `patientInfo`^{2}, `staffInfo`, where `patientInfo`^{1} comes from the inaccessible `clinicalTrial` with XPath query p_1^1 = `clinicalTrial/patientInfo`, while `patientInfo`^{2} is the accessible child of `dept` with query p_1^2 = `patientInfo`. A more compact form of this production is `dept` $\rightarrow$ `patientInfo`*, `staffInfo`, where `patientInfo` is annotated by $p_1 = p_1^1 \cup p_1^2$ (see Example 3.2). The algorithm then processes `staffInfo` and `patientInfo`. The former is simple: the productions remain unchanged from the document DTD for each element type involved. The latter is similar, until the inaccessible `trial` and `regular` are encountered. Since `treatment` is defined with a disjunction, while reg(`trial`) and reg(`regular`) are concatenations, the inaccessible nodes are renamed, and productions `dummy1` $\rightarrow$ `bill` and `dummy2` $\rightarrow$ `bill,medication` are added to D_v. The algorithm yields the security view of Example 3.2 upon its termination. □

4. Query Rewriting

We next study querying XML via security views. Consider a security view $V : S \rightarrow D_v$, where $S = (D, \mathsf{ann})$ and $V = (D_v, \sigma)$. Recall that users of V are provided with the view DTD D_v and are allowed to pose queries over V. A naive way to evaluate such queries is: given an instance T of the document DTD D, compute and store T_v, and evaluate queries directly over T_v. However, this introduces the overhead of materialization and the difficulties of view maintenance. These problems are more evident when multiple views of a large document are materialized at the same time.

We take a different approach, based on *query rewriting*: given an XPath query p over the security view, we automatically transform p to another XPath query p_t over the document DTD D such that, for any instance T of D, p over T_v and p_t over T yield the same answer. In other words, p over the view is *equivalent* to p_t over the original document (i.e., $p_t(T) = p(T_v)$). This eliminates the need for materializing views and its associated problems.

We present a query-rewriting algorithm for our class $\mathcal{C}$ of XPath queries, which, given a query p, computes p_t in $O(|p|*|D_v|^2)$ time. Our algorithm is not only useful in the security context, but is also interesting in its own right. First, it is among the first efforts for XPath query rewriting. Second, based on dynamic programming, our algorithm can be generalized to handle XPath queries beyond $\mathcal{C}$, without a significant increase in complexity.

Below we first present our query-rewriting algorithm for non-recursive views, i.e, views with a non-recursive (i.e., DAG) view DTD; we then generalize our algorithm to handle recursive views.

4.1 Query Rewriting for Non-recursive Views

Given a query p over the view DTD D_v, our rewriting algorithm "evaluates" p over the DTD graph D_v. For each node A reached via p from the root r of D_v, we rewrite every label path leading to A from r by incorporating the security-view XPath annotations σ along the path. As σ maps view nodes to document nodes, this yields a query p_t over the document DTD D.

To implement this idea, our algorithm works over the hierarchical, parse-tree representation[2] of the view query p and uses the following set of variables. For any sub-query p' of p and each node A in D_v, we use $\mathsf{rw}(p', A)$ to denote the *local translation* of p' at A, i.e., a query over D that is equivalent to p' when p' is evaluated at a context node A. Thus, $\mathsf{rw}(p, r) = p_t$ is what our algorithm needs to compute. We also use $\mathsf{reach}(p', A)$ to denote the nodes in D_v that are *reachable* from A via p'. Finally, we use N to denote the list of all the nodes in D_v, and Q to denote the list of all sub-queries of p in "ascending" order, such that all sub-queries of p' (i.e., its descendants in p's parse tree) precede p' in Q.

Given these, we present our rewrite algorithm in Fig. 6. The algorithm is based on *dynamic programming*: for each sub-query p' of p and node A in D_v, rewrite computes the local translation $\mathsf{rw}(p', A)$. To do this, rewrite first computes $\mathsf{rw}(p_i, B_i)$ for each (immediate) sub-query p_i of p' at each possible view DTD node B_i under A; then, it combines these $\mathsf{rw}(p_i, B_i)$'s to get $\mathsf{rw}(p', A)$. The details of this combination are, of course, determined based on the formation of p' from its immediate sub-queries p_i, if any (cases 1-12). The computation is carried out bottom-up via a nested iteration over the lists of sub-queries Q and DTD nodes N. Each step of the iteration computes $\mathsf{rw}(p', A)$ for some p' and A, starting from the "smallest" sub-queries of p. At the end of the iteration $p_t = \mathsf{rw}(p, r)$ is obtained.

For example, the algorithm rewrites '$*$' (case 3) at a view DTD node A to the query $\mathsf{rw}(*, A)$ over the original document; the rewritten query is the union of $\sigma(A, l)$'s for all child node l of A in the view DTD. As another example, it rewrites 'p_1/p_2' (case 4) to $\mathsf{rw}(p_1, A)/qq$, where qq is the union of $\mathsf{rw}(p_2, B)$'s for all node B in the view DTD reachable from A via p_1 (i.e., $B \in \mathsf{reach}(p_1, A)$).

A special case concerns the fixed query '$//$' (case 5). In order to reduce the overall processing costs, we assume that $\mathsf{reach}(//, A)$ and $\mathsf{rw}(//, A)$ have been precomputed by a procedure recProc (see Fig. 6) for each D_v node A, once and for all, and are made available to rewrite. Furthermore, for each B in $\mathsf{reach}(//, A)$, procedure recProc also returns an XPath query $\mathsf{recrw}(A, B)$ that captures all the paths from A to B, and translates it to an equivalent query over the document DTD D. Note that simply enumerating all the paths from A to a descendant B may lead to an exponential explosion; to avoid this, our recProc procedure employs symbolic variables

[1] We implicitly assume here that qualifiers in the specification can be copied to the result security view in constant time.

[2] As an example, for the simple query p = "//patient", its parse tree is a binary tree with root node representing p; its left and right (single-node) subtrees represent the sub-queries // and $patient$, respectively.

Algorithm rewrite

Input: a security view $V : S \rightarrow D_v$, a query p in $\mathcal{C}$ over view DTD D_v.
Output: an equivalent $\mathcal{C}$ query p_t over the document DTD D of S.

1. compute the ascending list Q of sub-queries of p;
2. compute the list N of all the nodes in D_v;
3. for each p' in Q do
4. for each A in N do
5. $\mathsf{rw}(p', A) := \emptyset$; $\mathsf{reach}(p', A) := \emptyset$;
6. for each p' in the order of Q do
7. for each A in N do
8. case p' of
9. (1) ϵ: $\mathsf{rw}(p', A) := \epsilon$; $\mathsf{reach}(p', A) := \{A\}$;
10. (2) l: if l is a child type of A
11. then $\mathsf{rw}(p', A) := \sigma(A, l)$; $\mathsf{reach}(p', A) := \{l\}$;
/* XPath annotation $\sigma(A, l)$ is given in V */
12. else $\mathsf{rw}(p', A) := \emptyset$; $\mathsf{reach}(p', A) := \emptyset$;
13. (3) $*$: for each child type v of A in D_v do
14. $\mathsf{rw}(p', A) := \mathsf{rw}(p', A) \cup \sigma(A, v)$;
15. $\mathsf{reach}(p', A) := \mathsf{reach}(p', A) \cup \{v\}$;
16. (4) p_1/p_2: if $\mathsf{rw}(p_1, A) = \emptyset$
17. then $\mathsf{rw}(p', A) := \emptyset$; $\mathsf{reach}(p', A) := \emptyset$;
18, else $qq := \emptyset$;
19. for each v in $\mathsf{reach}(p_1, A)$ do
20. $qq := qq \cup \mathsf{rw}(p_2, v)$;
21. $\mathsf{reach}(p', A) :=$
$\mathsf{reach}(p', A) \cup \mathsf{reach}(p_2, v)$;
22. if $qq \neq \emptyset$
23. then $\mathsf{rw}(p', A) := \mathsf{rw}(p_1, A)/qq$;
24. else $\mathsf{reach}(p', A) := \emptyset$; $\mathsf{rw}(p', A) := \emptyset$;
25. (5) $//p_1$: /* $\mathsf{reach}(//, A)$, $\mathsf{recrw}(A, B)$ are precomputed */
26. for each B in $\mathsf{reach}(//, A)$ do
27. if $\mathsf{rw}(p_1, B) \neq \emptyset$
28. then $\mathsf{rw}(p', A) := \mathsf{rw}(p', A) \cup \mathsf{recrw}(A, B)/\mathsf{rw}(p_1, B)$;
29. $\mathsf{reach}(p', A) := \mathsf{reach}(p', A) \cup \mathsf{reach}(B, p_1)$;
30. (6) $p_1 \cup p_2$: $\mathsf{rw}(p', A) := \mathsf{rw}(p_1, A) \cup \mathsf{rw}(p_2, A)$;
31. $\mathsf{reach}(p', A) := \mathsf{reach}(p_1, A) \cup \mathsf{reach}(p_2, A)$;
32. (7) $\epsilon[q]$: $\mathsf{rw}(p', A) := \epsilon[\mathsf{rw}(q, A)]$; $\mathsf{reach}(p', A) := \{A\}$;
33. (8) $[p_1]$: $\mathsf{rw}(p', A) := [\mathsf{rw}(p_1, A)]$;
34. (9) $[p_1 = c]$: $\mathsf{rw}(p', A) := [\mathsf{rw}([p_1]) = c]$;
35. (10) $[p_1 \wedge p_2]$: $\mathsf{rw}(p', A) := [\mathsf{rw}([p_1], A) \wedge \mathsf{rw}([p_2], A)]$;
36. (11) $[p_1 \vee p_2]$: $\mathsf{rw}(p', A) := [\mathsf{rw}([p_1], A) \vee \mathsf{rw}([p_2], A)]$;
37. (12) $[\neg p_1]$: $\mathsf{rw}(p', A) := [\neg \mathsf{rw}([p_1], A)]$;
38. return $\mathsf{rw}(p, r)$; /* r is the root of D_v */

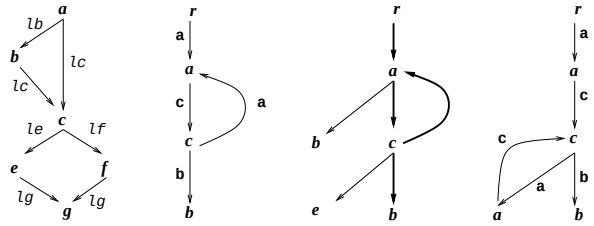
procedure recProc(A) /* static precomputation for $//$ */

Input: a node A in D_v.
Output: $\mathsf{reach}(//, A)$, and for each B in $\mathsf{reach}(//, A)$, $\mathsf{recrw}(A, B)$.

1. for each B in N do
2. $\mathsf{recrw}(A, B) := \emptyset$; $\mathsf{visited}(B) :=$ `false`;
3. traverse(A);
4. sort $\mathsf{reach}(//, A)$ such that x precedes y if 'Z_x' is in $\mathsf{recrw}(A, y)$
5. for each y in the topological list do
6. substitute each $\mathsf{recrw}(A, x)$ for 'Z_x', for each x
7. return ($\mathsf{reach}(//, A)$, recrw);

procedure traverse(x); /* invoked by procedure recProc */
1. for each child type y of x do
2. $\mathsf{recrw}(A, y) := \mathsf{recrw}(A, y) \cup$ 'Z_x'$/\sigma(x, y)$;
/* $\sigma(x, y)$ is the query associated with y in the x production */
3. if not $\mathsf{visited}(y)$
4. then $\mathsf{visited}(y) :=$ `true`; $\mathsf{reach}(//, A) := \mathsf{reach}(//, A) \cup \{y\}$;
5. traverse(y);
6. return;

Figure 6: An algorithm for XPath query rewriting



(a) query $//$ (b) view (c) spec. S (d) unfold

Figure 7: Query rewriting in the presence of recursion

('Z_x, Z_y') to ensure that, for each intermediate node C between A and B, any path from A to C (or, from C to B) is included in $\mathsf{recrw}(A, B)$ *only once*. Since D_v is a DAG, this guarantees that the size of $\mathsf{recrw}(A, B)$ is bounded by $|D_v|$. As an example, consider the view DTD shown in Fig. 7 (a), in which each element type X is associated with an XPath query lx. Then the query $\mathsf{recrw}(a, g)$ computed by procedure recProc(a) is $(l_b \cup \epsilon)/l_c/(l_e \cup l_f)/l_g$. It is easy to verify that recProc(A) takes $O(|D_v| * log|D_v|)$ time (due to the topological sort in step 4), and thus the precomputation takes $O(|D_v|^2 * log|D_v|)$ time for all D_v nodes.

Example 4.1: Given the XPath query `//patient//bill` over the `nurse` view of Fig. 2, Algorithm rewrite generates an equivalent query $p_1/p_2/p_3$, where

p_1 = `hospital/dept[*/patient/wardNo=$wardNo]`
p_2 = `(clinicalTrial` $\cup$ ϵ`)/patientInfo/patient`
p_3 = `treatment/(trial` $\cup$ `regular)/bill`

It is defined over the `hospital` document DTD of Fig. 1. □

Observe that each step in the iteration takes at most $O(|D_v|)$ time. Since the list Q is linear in the size of p, one can verify:

Theorem 4.1: *Given a non-recursive security view $V : S \rightarrow D_v$ and an XPath query p (in C) over D_v, Algorithm* rewrite *computes an equivalent query p_t over the original document in at most $O(|p| * |D_v|^2)$ time.* □

Algorithm rewrite and security views provide a general access control mechanism; no further treatment for security constraints is needed for a query generated by the algorithm. Furthermore, the algorithm can be generalized to deal with larger XPath fragments with upward and sideways axes, without significantly increasing its complexity. Indeed, recently [15] presented an algorithm for evaluating queries of the full XPath language in low polynomial time, which is essentially based on dynamic programming. Although the query rewriting problem is different from query evaluation, Algorithm rewrite can be generalized to deal with the full XPath language along the same lines as [15]. In contrast, query rewriting based on direct implementation of XPath semantics may, as pointed out by [15], take exponential time in the size of queries.

4.2 Coping with Recursive Views

Query rewriting becomes more intriguing when the view DTD is recursive. For example, consider the view DTD shown in Fig. 7 (b), which is derived from the specification S of Fig. 7 (c) (where, as in Fig. 4, normal edges point to inaccessible nodes). Consider query `//b` over the view. Although the view DTD is merely a sub-graph of the document DTD D for S, this query cannot be evaluated directly over instances of D since it returns the inaccessible b child of a. Algorithm rewrite no longer works here since a direct translation of '`//`' leads to *infinitely many paths*. Although the query is equivalent to the regular expression $(a/c)^*/b$, such regular expressions are beyond the expressive power of the XPath standard; thus, it is not always possible to rewrite an XPath query over a recursive view to an equivalent *XPath query* over a document DTD.

A solution to this problem is by *unfolding* recursive nodes. By unfolding a recursive DTD node A we mean creating distinct children for A following the A production. Referring to Fig. 7 (b), unfolding c by one level means creating a distinct a child for c instead of referring to the existing a node, as shown in Fig. 7 (d). Remember that a security view $V : S \rightarrow D_v$ is defined over a concrete XML document T. Since the height of T is known, one can determine by how many levels recursive nodes need to be unfolded, and such an unfolding yields a non-recursive (DAG) view DTD that the document is guaranteed to conform to. This allows us to use algorithm rewrite as before. Unfolding D_v to a DAG is possible since, as long as D_v is *consistent* (i.e., there exist documents conforming to it), each recursive A must have a non-recursive rule. For example, $a \rightarrow b$ is the non-recursive rule for $a \rightarrow a|b$, and $a \rightarrow b, \epsilon$ is the non-recursive rule for $a \rightarrow b, a^*$. Thus, for a fixed T, one can determine the unfolding levels and apply the non-recursive rules at certain stages. Note that when T is updated, the adjustment to the DTD unfolding is rather mild and does not introduce any serious overhead. It is worth mentioning that while access-control specifications, security views and their derivation are all conducted at the schema-level (i.e., on DTDs only), query rewriting over *recursive* security views needs the height information of the concrete XML tree over which the queries are evaluated.

5. Evaluation Optimization

The rewriting algorithm given in Section 4 transforms an XPath query over a security view to an equivalent XPath query over the original document. However, the rewritten query may not be efficient. This motivates us to consider XPath query optimization in the presence of a DTD D: given an XPath query p, find another query p_o such that over any instance T of D, (1) p and p_o are *equivalent*, i.e., $p(T) = p_o(T)$; and (2) p_o is *more efficient* than p, i.e., $p_o(T)$ takes less time/space to compute than $p(T)$. This is not only important in our access control model where queries generated by Algorithm rewrite are optimized using the document DTD, but is also useful for XPath query evaluation beyond the security context.

Ideally, one might want to find an optimal p_o with the least cost among all the queries equivalent to p over D. Unfortunately, one cannot find p_o efficiently: XPath optimization involves containment test, i.e., for two XPath queries p_1, p_2, to check whether p_1 is contained in p_2; recent study has shown that containment is coNP-hard even for simple non-recursive DTDs and for a small XPath fragment with union '$\cup$' but without '$//, *$' and qualifiers [24]. It is even undecidable for some more complex cases.

In light of these negative results, we develop our optimization algorithm based on approximate XPath containment tests and the structural properties of DTDs. We first present our approximate test and then develop our optimization algorithm.

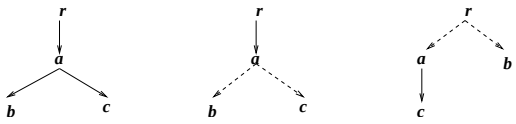
5.1 Approximate XPath Containment

To simplify the discussion, we consider a fragment $\mathcal{C}^-$ of the class $\mathcal{C}$ of XPath queries defined in Section 2. Queries of $\mathcal{C}^-$ allow '$//, /, *, \cup$', but restrict qualifiers to be conjunctive like [2, 27]:

$$q ::= \rho \mid q \wedge q \mid \rho := l \mid * \mid //\rho \mid \rho/\rho \mid \epsilon[q].$$

It should be mentioned that our approximate containment test can be generalized to deal with qualifiers with other boolean operators by using approximate satisfiability checking, a well studied technique (see, e.g., [28]).

We optimize $\mathcal{C}^-$ queries p using a DTD D with two techniques. First, we exploit the structural properties of DTD D to prune redundant sub-queries of p. The idea is to "evaluate" p over the DTD graph of D. When a sub-query p_1 is evaluated at a node A of D, we inspect p_1 and the DTD constraints imposed by the production of A, and decide whether p_1 can be reduced to $\emptyset$ and/or its qualifiers are equivalent to `true` or `false` at A, as illustrated below.



(a) co-existence (b) exclusive (c) non-existence

Figure 8: Query optimization in the presence of DTDs

Example 5.1: First, the query $//a[b \wedge c]$ over any instance of the DTD shown in Fig. 8 (a) is equivalent to $//a$, since the production of a in the DTD is defined with a concatenation, which imposes a *co-existence* constraint: b and c children must exist at the same time under a. Second, $//a[b \wedge c]$ is equivalent to $\emptyset$ over the DTD of Fig. 8 (b), because the production of a in the DTD is a disjunction, which asserts an *exclusive* constraint: either b or c is under a, but not both. Third, $(a \cup b)/c$ can be reduced to a/c over the DTD of Fig. 8 (c), because the production for b asserts a *non-existence* constraint: b cannot have a c child. □

Except for co-existence constraints (called child/descendant constraints [2, 27]), these DTD constraints have not been explored for optimizing XPath queries. Graph schema has been studied for regular path query optimization over semistructured data [14], but graph schema does not impose DTD constraints considered above.

Our second optimization technique is based on approximate containment test for XPath queries. We say that p_1 is *contained in* p_2 at a DTD node A if for any instance T of D and at any A-element n in T, $n[\![p_1]\!] \subseteq n[\![p_2]\!]$. If p_1 is contained in p_2, we can reduce $p_1 \cup p_2$ to p_2 at A, i.e., p_1 is redundant and is thus removed. Similarly, if $[q_1]$ is contained in $[q_2]$, i.e., $n[\![\epsilon[q_1]]\!] \subseteq n[\![\epsilon[q_2]]\!]$, then $[q_1 \wedge q_2]$ can be reduced to $[q_1]$ since $[q_1]$ is implied by $[q_2]$ at A

To implement these techniques we need a notion of image graphs. Below we first introduce the notion, and then outline our algorithms for approximate XPath containment test and redundant subquery pruning based on image graphs.

Image graphs. The image graph of p at a DTD node A, denoted by $\mathsf{image}(p, A)$, is a graph that is rooted at A and consists of all the nodes reached from A via p in the DTD graph, along with the paths leading to them. Specifically, $\mathsf{image}(p, A)$ is computed as follows. To simplify the discussion we consider non-recursive DTDs (DAGs); recursive DTDs is handled as described in Section 4.2.

(1) p is l. If A has an l child, then $\mathsf{image}(p, A)$ consists of an A-node, an l-node and an edge from A to l; otherwise it is empty.

(2) p is $*$. If A does not have any child, then $\mathsf{image}(p, A)$ is empty. Otherwise it consists of an A-node, all children of A and edges from A to these nodes.

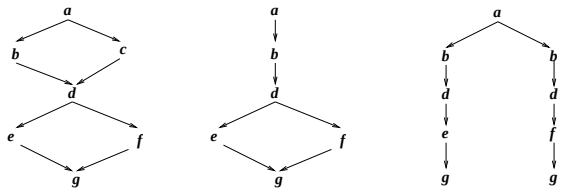
(3) p is p_1/p_2. Combine $\mathsf{image}(p_1, A)$ and $\mathsf{image}(p_2, B)$ to create $\mathsf{image}(p, A)$ for each node B reached from A via p_1 if $\mathsf{image}(p_2, B)$ is not empty, by merging the nodes representing B in these graphs. If $\mathsf{image}(p_1, A)$ is empty or if $\mathsf{image}(p_2, B)$ is empty for each B reachable from A via p_1, then so is $\mathsf{image}(p, A)$.

(4) p is $//p_1$. The treatment is similar to (3).

(5) p is $p_1 \cup p_2$. The graph $\mathsf{image}(p, A)$ is a combination of $\mathsf{image}(p_1, A)$ and $\mathsf{image}(p_2, A)$ by merging the nodes representing A in these graphs.

(6) p is $\epsilon[q]$. If $[q]$ is `true`, then $\mathsf{image}(p, A)$ consists of a single node A. If $[q]$ is `false`, then it is empty. Otherwise $\mathsf{image}(p, A)$ contains an A-node with an edge from A to the root of $\mathsf{image}([q], A)$, which is labeled '[ ]'.

For a qualifier $[q]$, $\mathsf{image}([q], A)$ is defined similarly, except that



(a) DTD (b) image(p_2, a) (c) image(p_3, a)

Figure 9: Image graphs of XPath queries

we attempt to evaluate $[q]$ at A to `true`/`false` by exploiting the co-existence, exclusive and non-existence constraints of the DTD D, as illustrated in Example 5.1. Specifically, we inspect the production of A and the structure of $[q]$ to determine the truth value of $[q]$ at A, denoted by bool$([q], A)$. If bool$([q], A)$ can be determined, then image$([q], A)$ is empty. The graph is constructed only when bool$([q], A)$ is not fixed, and it has the same structure as described above except that it has a distinguished root labeled '[ ]'. Due to the lack of space, we only define image$([q], A)$ and bool$([q], A)$ for some cases of $[q]$ as follows. Let the production of A be $A \rightarrow \alpha$.

(7) q is $*$. If α is ϵ or `str`, then bool$([*], A)$ is `false`. If α is a concatenation or disjunction, then bool$([*], A)$ is `true`; otherwise bool$([*], A)$ is undefined and image$(*, A)$ is created as in case (2).

(8) q is $q_1 \wedge q_2$. We consider the following cases.

- If either bool$([q_1], A)$ or bool$([q_2], A)$ is `false`, then so is bool$([q], A)$. If bool$([q_1], A)$ is `true`, then bool$([q], A)$ and image$([q], A)$ are bool$([q_2], A)$ and image$([q_2], A)$, respectively; similarly if bool$([q_2], A)$ is `true`.
- If α is a disjunction with B_1, B_2 in it, and in image(q_1, A) and/or image(q_2, A), the roots have B_1, B_2 children, then bool$([q], A)$ is `false`, due to the exclusive constraint.
- If q_1 is contained in q_2 at A (via an approximate algorithm for XPath containment test given below), then image$([q], A)$ is image$([q_1], A)$; similarly if q_2 is contained in q_1.
- Otherwise, we combine image$([q_1], A)$ and image$([q_2], A)$ to create image$([q], A)$ as in case (5) above.

One can verify that the size of image(p, A) is bounded by $|D| * |p|$, and it can be constructed in $O(|D| * |p|)$ time.

Example 5.2: At the node a of the DTD graph shown in Fig. 9 (a), consider queries $p_1 = a[b]/*/d/*/g$, $p_2 = a[b]/(b \cup c)/d/(e \cup f)/g$, and $p_3 = a[b]/b/d/e/g \cup a/b/d/f/g$. The image graph image$(p_1, a)$ is the same as Fig. 9 (a), while image(p_2, a) and image(p_3, a) are given in Fig. 9 (b) and Fig. 9 (c), respectively. The qualifier $[b]$ is evaluated to `true` at a and is thus removed. □

Qualifier optimization based on DTD constraints. Taking advantage of image graphs, a procedure, evaluate$([q], A)$, can be implemented to rewrite a qualifier $[q]$ at A to an equivalent yet simplified qualifier, denoted by opt$([q], A)$. Specifically, if bool$([q], A)$ is defined then opt$([q], A)$ = bool$([q], A)$. Otherwise, opt$([q], A)$ is a qualifier that is easily constructed based on image$([q], A)$.

Approximate containment test for XPath queries. Our approximate test is based on a simulation relation on image graphs, defined as follows. Given two image graphs G_1, G_2, a node v_1 in G_1 is *simulated by* a node v_2 in G_2, denoted by `simu`(v_1, v_2), if (1) v_1 and v_2 have the same label; (2) for any child x of v_1, if x is not labeled '[ ]' (i.e. x is not a qualifier), then v_2 has a child such that `simu`(x, y) returns `true`; and (3) for any child y of v_2 that is labeled '[ ]', v_1 has a '[ ]'-labeled child x such that `simu`(y, x) holds. Intuitively, all the 'non-qualifier' children of v_1 are simulated by some children of v_2; on the other hand, if v_2 has a qualifier y then v_1 must have a qualifier x such that x implies y, i.e., the subgraph rooted at y in G_2 is a subgraph of the subgraph rooted at x in G_1. In contrast to the conventional simulation relation on labeled graphs (see, e.g., [1]), our simulation relation alters direction when qualifiers are encountered. We say that G_1 is simulated by G_2 if the root of G_1 is simulated by the root of G_2.

Our simulation relation leads to a sound algorithm for testing containment of XPath queries in the presence of DTDs:

Proposition 5.1: *If graph* image(p_1, A) *is simulated by graph* image(p_2, A), *then* p_1 *is contained in* p_2 *at* A. □

Note that the simulation-based test is approximate since the other direction of the proposition may not hold. Similar approximation has also been used in, e.g., indexing for semistructured data [21].

Example 5.3: Referring to Example 5.2 and Fig. 9, observe that p_2 and p_3 are contained in p_1 at a while image(p_2, a) and image(p_3, a) are simulated by image(p_1, a). Similarly, p_3 is contained in p_2 at a and image(p_3, a) is simulated by image(p_2, a). However, image(p_2, a) is not simulated by image(p_3, a) although p_2 is indeed contained in p_3 at a. □

An algorithm for testing whether image(p_1, A) is simulated by image(p_2, A), Algorithm simulate(image(p_1, A), image(p_2, A)), can be easily developed as a mild extension of the well-known quadratic time algorithm for testing conventional graph simulation. The algorithm takes at most $O(|n_1| \, |n_2|)$ time, where n_1 and n_2 are the sizes of image(p_1, A) and image(p_2, A), respectively.

The containment test given above is quite different from the simulation-based technique proposed by [27]. Our simulation test is conducted on image *graphs* derived from a DTD graph w.r.t. a query, while the simulation of [27] is defined for *tree pattern* queries. Previous XPath optimization algorithms [2, 27, 18] do not support union, and at best consider simple DTD constraints only, e.g., the existence of child or descendant. There have also been optimization techniques based on classical integrity constraints [10, 11], which are complementary to our optimization technique.

5.2 Optimization Algorithm

Based on the approximate XPath containment test and structural constraints of DTDs, we present an optimization algorithm, Algorithm optimize, in Fig. 10. Given a DTD D and a $\mathcal{C}^-$ query p, Algorithm optimize(D, r, p) rewrites p to an equivalent yet more efficient p_o, where r is the root of D. The algorithm uses the following variables. (1) For each sub-query p' of p and each type A in the DTD D, opt(p', A) denotes optimized p' at A, i.e., a query equivalent to but more efficient than p' when being evaluated at an A element. The variable is initially '$\perp$' indicating that opt(p', A) is not yet defined, which ensures that each sub-query is processed at each DTD node at most once. (2) reach(p', A) is the set of nodes in D reachable from A via p', with an initial value $\emptyset$. (3) image(p', A) is the image graph of p' at A. The algorithm also invokes the following procedures. (1) recProc(A, B) is a mild variation of the version given in Fig. 6. It precomputes reach$(//, A)$ and moreover, for each B in reach$(//, A)$, derives an XPath query recrw(A, B) that captures all the paths from A to B. It differs from the one of Fig. 6 in that there is no need to substitute XPath annotations for a node label. (2) simulate(image(p_1, A), image(p_2, A)) checks whether image(p_1, A) is simulated by image(p_2, A), as described earlier. (3) evaluate$([q], A)$ evaluates a qualifier q at A by exploiting the DTD constraints, as given earlier.

Given these, Algorithm optimize(D, A, p) rewrites query p at A elements based on the structures of p and A (cases 1–7). It recursively prunes redundant sub-queries of p by exploiting the structural constraints of the DTD D. For example, given a query

Algorithm optimize (D, A, p)

Input: an XPath query p in $\mathcal{C}^*$, a DTD D and a node A in D.
Output: an optimized $\mathcal{C}^*$ query over D, equivalent to p at A elements.

1. opt$(p, A) := \emptyset$;
2. case p of
3. (1) ϵ: opt$(p, A) := \epsilon$; reach$(p, A) := \{A\}$;
4. (2) l: if A has an l child v
5. then opt$(p, A) := l$; reach$(p, A) := \{v\}$;
6. else opt$(p, A) := \emptyset$; reach$(p, A) := \emptyset$;
7. (3) $*$: for each child B of A in D do
8. opt$(p, A) :=$ opt$(p, A) \cup B$;
9. reach$(p, A) :=$ reach$(p, A) \cup \{B\}$;
10. (4) p_1/p_2:
11. if opt$(p_1, A) = \perp$
12. then optimize(D, A, p_1);
13. for each B in reach(p_1, A) do
14. if opt$(p_2, B) = \perp$
15. then optimize(D, B, p_2);
16. if opt$(p_2, B) \neq \emptyset$
17. then opt$(p, A) :=$ opt$(p, A) \cup$ opt$(p_1, A)/$opt(p_2, B);
18. reach$(p, A) :=$ reach$(p, A) \cup$ reach(p_2, B);
19. (5) $//p_1$: /* reach$(//, A)$ and recrw(A, B) have been */
precomputed by procedure recProc(A) given in Fig. 6 */
20. for each B in reach$(//, A)$ do
21. if opt$(p_1, B) = \perp$
22. then optimize(D, B, p_1);
23. if opt$(p_1, B) \neq \emptyset$
24. then opt$(p, A) :=$ opt$(p, A) \cup$ recrw$(A, B)/$opt(B, p_1);
25. reach$(p, A) :=$ reach$(p, A) \cup$ reach(B, p_1);
26. (6) $p_1 \cup p_2$: if opt$(p_1, A) = \perp$
27. then optimize(D, A, p_1);
28. if opt$(p_2, A) = \perp$
29. then optimize(D, A, p_2);
30. if simulate(image(p_1, A), image(p_2, A))
31. then opt$(p, A) :=$ opt(p_2, A); reach$(p, A) :=$ reach(p_2, A);
32. else if simulate(image(p_2, A), image(p_1, A))
33. then opt$(p, A) :=$ opt(p_1, A); reach$(p, A) :=$ reach(p_1, A);
34. else opt$(p, A) :=$ opt$(p_1, A) \cup$ opt(p_2, A);
35. reach$(p, A) :=$ reach$(p_1, A) \cup$ reach(p_2, A);
36. (7) $\epsilon[q]$: if opt$([q], A) = \perp$
37. then evaluate$(A, [q])$;
38. if opt$([q], A) =$ `true`
39. then opt$(p, A) := \epsilon$; reach$(p, A) := \{A\}$;
40. else if opt$([q], A) =$ `false`
41. then opt$(p, A) := \emptyset$; reach$(p, A) := \emptyset$;
42. else opt$(p, A) := \epsilon$[opt$([q], A)$]; reach$(p, A) := \{A\}$;
43. return opt(p, A);

Figure 10: An algorithm for XPath query optimization

$p_1 \cup p_2$ and a DTD node A (case 6), the algorithm first optimizes p_1 and p_2 at A (i.e., it computes opt(D, p_1, A) and opt(D, p_2, A)). It then checks whether p_1 is (approximately) contained in p_2 by testing whether image(p_1, A) is simulated by image(p_1, A). If so, it rewrites $p_1 \cup p_2$ to p_2; similarly if p_2 is contained in p_1. Along the same lines, case 7 may remove a qualifier $[q]$ if it is `true`, reduce $\epsilon[q]$ to an empty set if $[q]$ is `false`, or simplifies $[q]$.

Example 5.4: Consider a query $p = p_1 \cup p_2$ over the hospital DTD shown in Fig. 1, where p_1 is `//patient` and p_2 is `//(patient` $\cup$ `staff)[//medication]`. The algorithm optimize converts p to a query p_{o1}/p_{o2}, where p_{o1} = `hospital/dept`, and p_{o2} = `(clinicalTrial` $\cup$ ϵ`)/patientInfo/patient`. Specifically, it first transforms p_1 to $p_1' = p_{o1}/p_{o2}$ and p_2 to $p_2' = p_{o1}/p_{o2}$[`treatment/regular/medication`]; then, it finds that the image graph of p_2' is simulated by the image graph of p_1'. Now it concludes that p_2' is contained in p_1' and thus returns p_1' as the result of optimizing p. □

One can verify that the algorithm takes $O(|D|^3 * |q|^3)$ time in the worse case. This is because opt(p', A) is computed at most once for each sub-query p' and DTD node A. Each step takes at most $O(|D|^2 * |q|^2)$ time, which comes from invoking the procedure simulate for approximate XPath containment test, while the image graphs involved in procedure simulate are bounded by $|D| * |q|$ in size. Observe that $|D|$ and $|q|$ are small in practice.

6. Experimental Results

To verify the effectiveness of our approach and optimization algorithms, we have conducted a performance study using real-life DTDs and various XPath queries. Our experimental results clearly demonstrate both the efficiency of our query rewriting approach over a straightforward query rewriting approach (that is based on element-level security annotations) as well as the benefits of our optimization techniques, particularly for large documents. Specifically, our query rewriting approach can achieve an improvement by up to a factor of 40 over naive query rewriting, which can be further improved by up to factor of 2 using our optimization algorithm.

XML Documents. Our data sets are generated with the real-life Adex DTD [23], which is a standard proposed by the Newspaper Association of America for electronic exchange of classified advertisements. We generated XML documents using IBM's XML Generator tool [12] by varying the maximum branching factor parameter to obtain four documents: $D1(3.2MB)$, $D2(16.7MB)$, $D3(51.55MB)$, and $D4(77.0MB)$.

Security Views. For the Adex DTD, we created a security view for a user where he is permitted to access only data related to real estate advertisements and data related to buyers. This security view is created by simply annotating the children of the root element `adex` as "N" and both the `real-estate` and `buyer-info` descendants as "Y" in the Adex DTD.

XPath Queries. We consider the following four XPath queries on the Adex security view:

```
Q1: //buyer-info/contact-info
Q2: //house/r-e.warranty | //apartment/r-e.warranty
Q3: //buyer-info[company-id and contact-info]
Q4: //house[//r-e.asking-price and //r-e.unit-type]
```

$Q1$ simply retrieves the contact information of all buyers. $Q2$ retrieves the real estate warranty information for houses and apartments. $Q3$ retrieves information of buyers who have both company-id and contact-info subelements. $Q4$ retrieves houses that have both asking price and unit type information.

Approaches. We compared three different approaches (naive, rewrite, optimize) in our experiments, all of which are based on the use of security views for querying. The first ("naive") approach, which does not use DTD for query rewriting, requires the data documents to be annotated with additional element accessibility information and works as follows. A new attribute called *accessibility* is defined for each element in the XML document which is used to store the accessibility value of that element. The naive approach uses two simple rules to rewrite an input query to ensure that (a) it accesses only authorized elements and (b) it is converted to a query over the document. The first rule adds the qualifier $[@accessibility = "1"]$ to the last step of the query to ensure (a). The second rule replaces each child axis in the query with the descendant axis to ensure (b)[3]. The second rule is necessary since an edge in a security view DTD can represent some path in the document DTD. Thus, the naive approach represents a simple rewriting approach that relies on element-level annotations instead

[3]This rule is sound so long as the DTD has unique element names.

Query	Data Set	Naive	Rewrite	Optimize
Q1	D1	4.12	0.44	-
	D2	39.75	2.69	-
	D3	416.85	13.09	-
	D4	917.64	22.53	-
Q2	D1	8.49	0.54	-
	D2	72.41	2.81	-
	D3	916.15	11.42	-
	D4	1406.56	19.16	-
Q3	D1	4.1	0.54	0.50
	D2	41.20	2.92	2.67
	D3	464.66	11.39	8.15
	D4	1128.12	36.07	15.89
Q4	D1	3.89	0.51	0
	D2	40.58	3.17	0
	D3	466.61	11.31	0
	D4	1021.55	38.03	0

Table 1: Performance Comparison

of DTD for query rewriting. The second ("rewrite") approach is our proposed method of rewriting queries using DTD. The third ("optimize") approach is an enhancement of the second approach that further optimizes the rewritten queries using our proposed optimizations. To compare the performance of the three approaches, we used a state-of-the-art XPath evaluation implementation [17] that has been shown to be more efficient and scalable than several existing XPath evaluators [15]. Our experiments were conducted on a 2.4 GHz Intel Pentium IV machine with 512 MB of main memory running Microsoft Windows XP.

Experimental Results. The experimental results are shown in Table 1, where each row compares the query evaluation time (in seconds) of naive, rewrite, and optimize approaches for a given document and query. For queries that can not be further improved by the optimize approach, we indicate this with a "-" value under the optimize column.

For Q1, the naive approach evaluates it as *//buyer-info//contact-info[@accessibility="1"],* while the rewrite approach utilizes the DTD to expand Q1 into a more precise query */adex/head/buyer-info/contact-info.*

The naive approach rewrites $Q2$ to *//house//r-e.warranty [@accessibility="1"]* | *//apartment//r-e.warranty [@accessibility="1"]* while the rewrite approach expands the query to */adex/body/ad-instance/real-estate/house/r-e.warranty.* Note that the rewrite approach has simplified the second sub-expression to empty since the $r\text{-}e.warranty$ element is not a sub-element of $apartment$.

For Q3, the naive approach evaluates the query as *//buyer-info[//company-id and //contact-info][@accessibility="1"],* while the rewrite approach expands the query to */adex/head/buyer-info[company-id and contact-info].* The optimize approach further exploits the co-existence constraint that each *buyer-info* element has both *company-id* and *contact-info* sub-elements to simplify the rewritten query to */adex/head/buyer-info.*

The query Q4 shows the benefit of exploiting the exclusive constraint. The rewrite approach expands the query to */adex/body/ad-instance/real-estate [house/r-e.asking-price and apartment/r-e.unit-type],* which is further refined by the optimize approach to an empty query since the *real-estate* element can not have both *house* and *apartment* sub-elements; thus the evaluation of Q4 can be avoided.

Overall, our experimental results have demonstrated the effectiveness of our proposed query rewriting technique for processing secured XML queries. Our results have also emphasized the importance of using DTD constraints to optimize the evaluation of XPath queries on large XML documents.

7. Conclusions

We have proposed a new paradigm for securing XML data (based on the novel notion of security views) and thoroughly studied its implications on the XML query-execution engine. To enable the efficient evaluation of user queries over security views, we have introduced novel query rewriting and optimization algorithms for a significant fragment of XPath. This yields the first XML security model that provides both access/inference control and schema availability. Our query rewriting and optimization techniques are not only important for XML access control but also useful for XML view processing in general. Our experimental results show that these techniques yield substantial reductions in processing time.

We plan to extend our rewriting and optimization techniques to handle larger fragments of XPath and other XML query languages such as XSLT and XQuery. We are also studying extensions of our notion of security views based on XML Schema instead of DTD.

8. References

[1] S. Abiteboul, P. Buneman, and D. Suciu. *Data on the Web. From Relations to Semistructured Data and XML*. Morgan Kaufman, 2000.

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

[3] E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *TISSEC*, 5(3):290–331, 2002.

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

[5] S. Castano, M. G. Fugini, G. Martella, and P. Samarati. *"Database Security"*. Addison-Wesley, 1995.

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

[7] J. Clark and S. DeRose. XML Path Language (XPath). W3C Working Draft, Nov. 1999.

[8] E. Damiani, S. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *EDBT*, 2000.

[9] E. Damiani, S. di Vimercati, S. Paraboschi, and P. Samarati. A fine-grained access control system for XML documents. *TISSEC*, 5(2):169–202, 2002.

[10] A. Deutsch, L. Popa, and V. Tannen. Physical data independence, constraints, and optimization with universal plans. In *VLDB*, 1999.

[11] A. Deutsch and V. Tannen. Reformulation of XML queries and constraints. In *ICDT*, 2003.

[12] A. Diaz and D. Lovell. XML generator, 1999.

[13] M. F. Fernandez, Y. Kadiyska, D. S. A. Morishima, and W. Tan. SilkRoute: A framework for publishing relational data in XML. *TODS*, 27(4):438–493, 2002.

[14] M. F. Fernandez and D. Suciu. Optimizing regular path expressions using graph schemas. In *ICDE*, 1998.

[15] G. Gottlob, C. Koch, and R. Pichler. Efficient algorithms for processing XPath queries. In *VLDB*, 2002.

[16] S. Hada and M. Kudo. XML access control language: Provisional authorization for XML documents. `http://www.trl.ibm.com/projects/xml/xacl/xacl-spec.html`.

[17] C. Koch. XML Task Force, 2003.

[18] G. Miklau and D. Suciu. Containment and equivalence of XPath expressions. In *PODS*, 2002.

[19] G. Miklau and D. Suciu. Controlling access to published data using cryptography. In *VLDB*, 2003.

[20] R. Milner. *"Communication and Concurrency"*. Prentice Hall (Intl. Series in Computer Science), 1989.

[21] T. Milo and D. Suciu. Index structures for path expressions. In *ICDT*, 1999.

[22] M. Murata, A. Tozawa, M. Kudo, and S. Hada. XML access control using static analysis. In *CCS*, 2003.

[23] NAA classified advertising standards task force. Adex DTD, 1999.

[24] F. Neven and T. Schwentick. XPath containment in the presence of disjunction, DTDs, and variables. In *ICDT*, 2003.

[25] Oasis. eXtensible Access Control Markup Language (XACML). `http://www.oasis-open.org/committees/xcaml`.

[26] Y. Papakonstantinou and V. Vassalos. Query rewriting for semistructured data. In *SIGMOD*, 1999.

[27] P. Ramanan. Efficient algorithms for minimizing tree pattern queries. In *SIGMOD*, 2002.

[28] T., Asano, K. Hori, T. Ono, and T. Hirata. A theoretical framework of hybrid approaches to MAX SAT. In *Proc. 8th Ann. Int. Symp. on Algorithms and Computation*, 1997.