Optimizing the Secure Evaluation of Twig Queries

SungRan Cho
Stevens Institute of Technology
scho@attila.stevens-tech.edu

Sihem Amer-Yahia
AT&T Labs–Research
sihem@research.att.com

Laks V.S. Lakshmanan*
University of British Columbia
laks@cs.ubc.ca

Divesh Srivastava
AT&T Labs–Research
divesh@research.att.com

Abstract

The rapid emergence of XML as a standard for data exchange over the Web has led to considerable interest in the problem of securing XML documents. In this context, query evaluation engines need to ensure that user queries only use and return XML data the user is allowed to access. These added access control checks can considerably increase query evaluation time. In this paper, we consider the problem of optimizing the secure evaluation of XML twig queries.

We focus on the simple, but useful, multi-level access control model, where a security level can be either specified at an XML element, or inherited from its parent. For this model, secure query evaluation is possible by rewriting the query to use a recursive function that computes an element's security level. Based on security information in the DTD, we devise efficient algorithms that optimally determine when the recursive check can be eliminated, and when it can be simplified to just a local check on the element's attributes, without violating the access control policy. Finally, we experimentally evaluate the performance benefits of our techniques using a variety of XML data and queries.

*Lakshmanan's research was supported by grants from NSERC and NCE/IRIS.

Proceedings of the 28th VLDB Conference, Hong Kong, China, 2002

1 Introduction

Companies are using the Web as the main means of information dissemination, sparking interest in models and efficient mechanisms for controlled access to information content over the Web. In this respect, securing XML documents is an important step, because XML is rapidly emerging as the standard for data representation and exchange over the Web.

Much of the work on XML access control to date (see, e.g., [4, 5, 2, 10, 9, 3]) has studied models for the specification of XML access control policies, focusing on issues such as granularity of access (e.g., DTD, document, element), propagation options (e.g., local, inherited), and conflict resolution (e.g., most specific, mandatory). Mechanisms for the enforcement of these XML access control policies have been studied for the cases of document access (e.g., [9]), and document browsing and authoring (e.g., [5, 2]). However, despite the importance of query access to XML (see, e.g., [7, 6]), there has been no work on enforcement of access control policies for the case of XML query access.

A naive two-step approach to secure XML query evaluation is: (i) compute the query result using existing XML query processing techniques (see, e.g., [13, 16, 8, 1]), and (ii) filter the query results, using the access control policies, in a post-processing step. While this approach may appear attractive, it is not secure. For example, consider the XML database of an online-seller (the DTD is illustrated in Figure 1), which has information about books and customer accounts. Assume that a specific user is allowed access to the book information but not to any account information. If *only* query results are filtered for accessibility, then the following XQuery path expression:

```
/online_seller[.//customer/name='smith']//book
```

would allow the user to check the existence of customer `smith`, which is clearly not the desired intent of the

access control policy. In general, secure query evaluation requires the evaluation engine to ensure that user queries *only* be permitted to check conditions on, and return, XML data that the user is allowed to access. In the above example, since the user is not allowed to access account information, the query result returned to that user should be empty, whether or not there is a customer by that name.

This paper takes the first steps towards addressing the important problem of efficient, secure XML query evaluation. For this purpose, we focus on tree pattern (i.e., twig) queries that are the basis of many XML query languages (see, e.g., [7, 6]). We consider the simple, but useful, multi-level security model (see, e.g., [11]) for XML data, where: (i) a security level can be specified as an attribute at the granularity of an XML element; and (ii) an element's security level is inherited by its sub-elements unless explicitly overridden (either monotonically or non-monotonically). Users can access elements whose security level is no higher than their own. The DTD identifies which elements must, may, or cannot specify a value for the security level attribute. This security model elegantly captures the key features of multiple granularity access control specification, propagation by inheritance, and overriding that are present in many of the XML access control models proposed in the literature.

Our technical contributions are as follows.

- First, we show that secure query evaluation is possible for a twig query, and the multi-level security model, by simply rewriting the query to use a recursive function that computes an element's security level by identifying the element's nearest ancestor (or self) where a security level attribute is specified.

  Since existing XML query processors can handle such functions, this approach has the advantage that secure query evaluation does not require changes to query evaluation engines.

- In general, these added access control checks can considerably increase query evaluation time. For example, we observed that the query evaluation time of a simple path query on a 30Mb XMark benchmark data set went up from one second to about five seconds! The key to optimizing the secure evaluation is taking advantage of the DTD, which identifies elements that must, may, or cannot specify a value for the security level attribute.

  For our second contribution, we consider DAG-structured DTD graphs, and devise efficient algorithms that *optimally* determine when a recursive check can be eliminated, and when it can be simplified to a local check on the element's attributes, without violating the multi-level security policy.

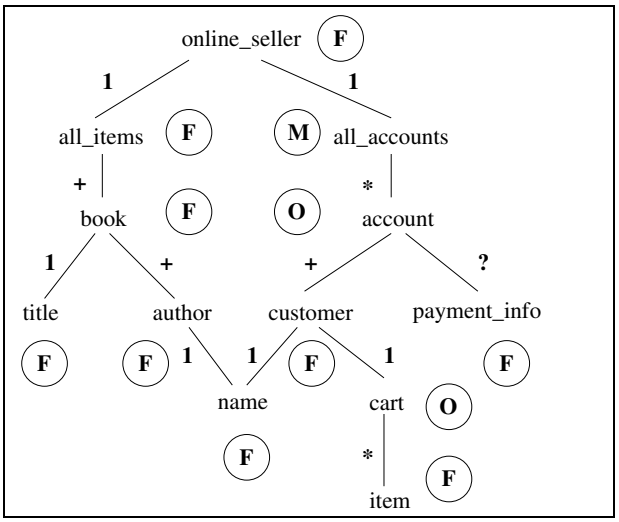


Figure 1: Example DTD Graph

- Finally, we experimentally evaluate the performance benefits of our techniques, using XMark and HL7 data, and a variety of twig queries. Our results demonstrate the significant benefits of optimizing the secure evaluation, while keeping optimization time very low, for both sparse and dense security level annotations in the data and DTD.

2 Motivation

Before we formally define our problems, and develop our solutions, we'd like to present a few examples that capture the essence of optimizing the secure query evaluation of twig queries.

Example Setting:

Consider the example DTD graph of Figure 1, representing the XML database schema of an `online_seller`. The database contains information about multiple (zero or more) `accounts`. Each `account` is associated with an optional `payment_info` (e.g., credit card number), and one or more `customers` (e.g., different members of a family). Each `customer` has one `name` and a `cart` that contains zero or more `items`. The database also contains information about one or more `books`. Each `book` has one `title` and one or more `authors`. Each `author` has one `name`. These various multiplicities are shown as edge labels of the DTD graph (? for optionality, 1 for one, $*$ for zero or more, $+$ for one or more). Note that the DTD graph is a DAG, since the `name` element is a sub-element of both the `customer` element and the `author` element.

Each element in the DTD can specify whether its instances must specify a value for the `SecurityLevel` attribute (i.e., it is mandatory), may specify a value for this attribute (i.e., it is optional), or cannot specify a value for this attribute (i.e., it is forbidden). These

are depicted in the DTD graph of Figure 1, as circles containing a boldface **M**, **O**, and **F**, respectively, associated with the element nodes. For example, specifying a security level is mandatory at the `all_accounts` node, optional at the `account` and `cart` nodes, and forbidden elsewhere.

The actual values of the `SecurityLevel` attributes are specified in the XML data items themselves, and may vary from instance to instance. In our multi-level security model, these values are from a partially ordered domain, though our examples will use a totally ordered domain (of integers in the range $[1..n]$ for simplicity of exposition). For example, in an instance of this DTD, the (mandatory) `SecurityLevel` of the `all_accounts` node may be 2 (medium security). Some of the `accounts` may explicitly specify a higher security level, say 3 (high security), while others inherit the security level of 2 specified at the `all_accounts` node. Assume that some high security `customers` are willing to let their `cart` nodes have medium security (say, to permit statistical analysis), while still preserving high security of their other nodes. These `cart` nodes would need to override the inherited security level, by explicitly specifying a `SecurityLevel` of 2. This natural example of non-monotonic overriding (from 2 to 3, and back to 2, for nodes deeper in the XML tree) illustrates that overriding of security levels need not always be monotone.

Twig Queries:

Some example path and twig queries consistent with this DTD graph are illustrated in Figure 2. Nodes are labeled with element tags or string values; edges are either parent-child (single edge) or ancestor-descendant (double edge); solid edges are between elements, while a dashed edge is between an element and a string value. Additionally, a node may be labeled with an asterisk, indicating that it is in the query projection list. For now, ignore the other node labels.

Each of the queries in Figure 2 can be expressed as a single XPath expression. For example, the path query in Figure 2(a) can be expressed as:

```
//customer[./cart/item = 'labyrinths']
```

Secure Query Evaluation:

Each twig query can be evaluated securely, by additionally testing a predicate at *each* query element node that compares the security level of that node (computed by identifying the element's nearest ancestor (or self) where a `SecurityLevel` attribute is specified) with that of the user. We refer to this as the *recursive check*, or `RC`. Thus, annotating each query element node with an `RC` ensures secure evaluation.

While annotating each query element node with an `RC` may sometimes be unavoidable, often one can eliminate checks at some nodes (i.e., a *no check*, or `NC`), while at other nodes one may be able to simplify the check to merely examining the value of the `SecurityLevel` attribute at that node (i.e., a *local check*, or `LC`). We illustrate this behavior through a series of examples, mainly for the case where the relevant part of the DTD graph is a tree structure, and finally one example where the DAG structure of the DTD graph plays a role.

It is important to keep in mind that these optimizations (like all query optimizations) are performed *without* examining the actual database instance. Only the DTD graph and the query are examined, and hence our reasoning needs to allow for any possible database instance that is valid with respect to the DTD.

Optimizing Parent-Child Edges:

Consider the path query in Figure 2(a). Since security levels are inherited in a top-down fashion, a natural strategy is to examine the query nodes in a top-down fashion as well. One can optimize this query in a top-down fashion as follows. At the `customer` node, one needs to retain the `RC` annotation label, since a `customer` node inherits its security level (either from an `account` node, or the `all_accounts` node, depending on the instance). Once the `customer` node is determined to be accessible, the only way its child `cart` node may be inaccessible is if it explicitly overrides the inherited security level. This possibility can be checked using the local check, `LC`; an `RC` is not needed! Finally, an `item` node is not permitted (by the DTD) to explicitly specify its own security level, hence it always inherits it from its parent `cart` node. Thus, the check at the `item` node can be eliminated, modifying the annotation label to `NC`. It is easy to see that this is *optimal*, in that no `RC` annotation label can be converted to an `LC` or an `NC`, without the possibility of the security policy being violated in *some* XML database instance.

The query in Figure 2(b) is an example where all the `RC` annotation labels can be converted to `NC`, i.e., no check needs to be performed. To see this, observe that all ancestors of the `book` node in the DTD graph are forbidden to specify a value for the `SecurityLevel` attribute. Similarly for the `author` node, and the `name` child of the `author` node. Again, the query can be optimized in a top-down fashion.

A similar reasoning as the one applied for the query in Figure 2(a) can be employed for the query in Figure 2(c), using a top-down strategy, examining each parent-child edge, one at a time. More generally, if the DTD node corresponding to a child query node can specify the `SecurityLevel` attribute (either mandatory or optional), the annotation label at the child query node can be simplified to `LC`; if the corresponding DTD node is forbidden to have the `SecurityLevel` attribute, the check can be eliminated (equivalently, the annotation label becomes `NC`).

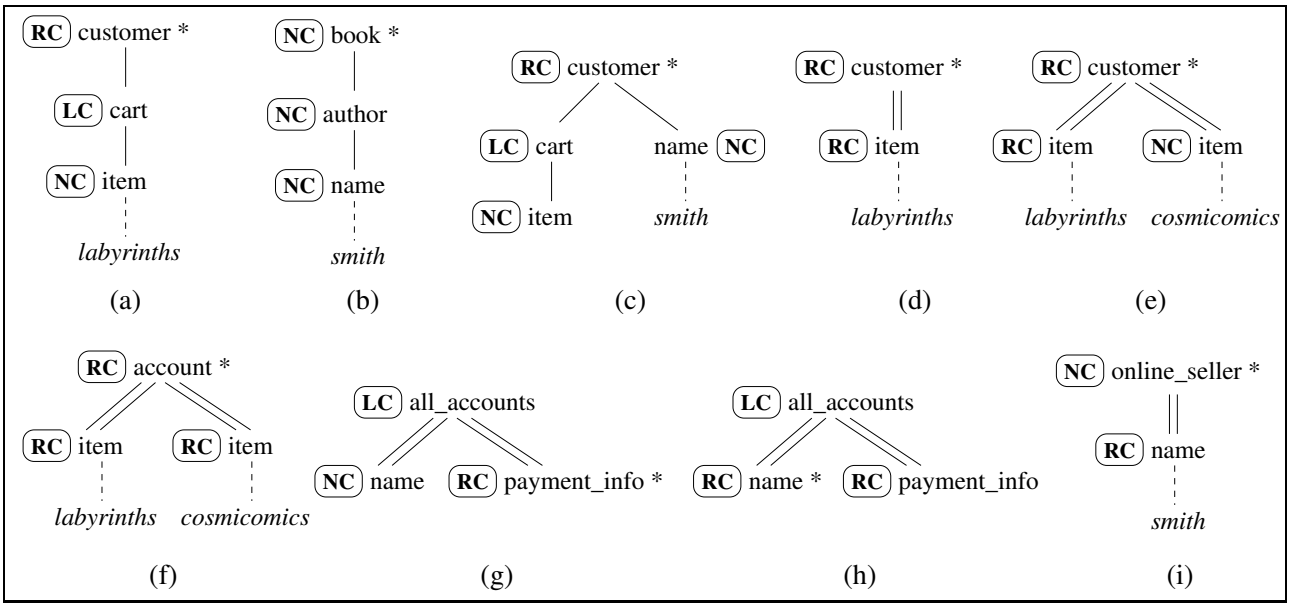


Figure 2: Example Path and Twig Queries

Optimizing Ancestor-Descendant Edges:

Consider the path query in Figure 2(d). Again, let us examine this query in a top-down fashion. As with the path query in Figure 2(a), one needs to retain the `RC` annotation label at the `customer` node. Now, while an `item` node is not permitted to specify a value for the `SecurityLevel` attribute, its security level may not be the same as the `customer` node. This may have been overridden (either by increasing or decreasing the security level) at the (missing) `cart` node. To check for this possibility, one needs to retain `RC` annotation label at the `item` node. Again, this query is *optimal*, in that the annotations identified by our technique are necessary to ensure a secure evaluation.

Insufficiency of Top-down Analysis:

The twig query in Figure 2(e), with ancestor-descendant edges, is our first example that illustrates the insufficiency of a simple top-down analysis. A top-down analysis would determine, as in the case of Figure 2(d), that each of the two `item` nodes should retain their `RC` annotation labels. However, this is sub-optimal! To see this, note that an `item` node has to inherit its security level from its (parent) `cart` node. Since a `customer` node has only one `cart` node, both `item` nodes are guaranteed to have the same security level, independent of its actual value. For this reason, one of the two `RC`s can be eliminated, as depicted in Figure 2(e).

We will show later that a simple *forward pass*, where query nodes are examined in a topological order and state information is maintained about visited query nodes, suffices to infer the optimal security check annotations for the secure query evaluation.

Note that while the query in Figure 2(f) appears similar to the one in Figure 2(e), there is a crucial difference. The two `item` nodes here may inherit their security levels through *different* `cart` nodes, each of which has the ability to independently specify its own value for the `SecurityLevel` attribute. This forces both `item` nodes to retain their `RC` annotation labels.

Dealing with Existential Checks:

Consider the twig query in Figure 2(g). This can be expressed as the XQuery expression:

```
//all_accounts[.//name]//payment_info
```

Intuitively, this query returns all accessible `payment_info` nodes, provided that its ancestor `all_accounts` node is accessible, and this `all_accounts` node has an accessible `name` descendant. Since `payment_info` and `name` nodes both inherit their security levels through `account` nodes, one may be tempted to infer that both nodes need to retain their `RC` annotation labels, as was the case in Figure 2(f). However, this would be sub-optimal. The key observation is that for any given accessible `payment_info` node, the DTD guarantees that its (parent) `account` node will have at least one (descendant) `name` node (the path in the DTD graph from `account` to `name` has edge labels from the set $\{1, +\}$), and this `name` node will have the same security level as the `payment_info`. As a result, replacing the `RC` check at `name` by an `NC` does not violate security.

Now, consider the symmetric case in Figure 2(h), where the query returns all accessible `name` nodes instead. In this case, however, one cannot eliminate or simplify either of the `RC` checks. Essentially, the DTD

graph does not guarantee (because of the ? edge label between `account` and `payment_info`) that there is a `payment_info` node with the same security level as an accessible `name` node.

Dealing with a DAG DTD:

Finally, we hint at the kinds of issues created when dealing with DTD graphs that are not trees, but DAGs. Consider the path query in Figure 2(i). It simply checks if there is an accessible `name` node in the database with content `smith`. Such a `name` node, if it exists, might be the name of an author, or the name of a customer. By not performing an `RC` check, one might mistakenly determine that `name` nodes that are descendants of inaccessible `account` nodes are accessible. For this reason, the `RC` check needs to be retained at `name`.

If, however, the query in Figure 2(i) did not test the `content` of the `name` node, but merely checked for the presence of an accessible `name` node, its existence and accessibility are guaranteed by the DTD (since the online supplier has at least one book, which has at least one author, which has one name, all of which are accessible).

In the sequel, we shall use the intuitions developed in this section to present efficient algorithms for determining the optimal security check annotations for any twig query, for the case of tree-structured and DAG-structured DTD graphs.

3 Security Model and Semantics

In this section, we describe the security model, the class of queries we consider, security check annotations, and the semantics of secure query evaluation. We assume the reader is familiar with XML.

3.1 DTDs and Queries

Essentially, a DTD is an extended context-free grammar describing the structure of elements in terms of required and possible subelements, and attributes that may be attached to the element. In this paper, we focus on DTDs that can be represented as a (directed) graph defined as follows.

Definition 3.1 [DTD Graph] A *DTD graph* is a finite node-labeled and edge-labeled directed graph $D = (V, E)$, such that: (i) each node in V is labeled by an element tag, and (ii) each edge has a label from the set $\{+, *, ?, 1\}$. □

It should be clear that the set of edges leaving a node labeled `A` and entering, say `B, C, D`, define element `A` as composed of subelements `B, C, D`. Furthermore, edge labels indicate the frequency of subelements to be expected: $+, *, ?, 1$ meaning, respectively "one or more", "zero or more", "zero or one", and "exactly one". DTDs also permit a description of attributes that may be attached to an element's opening tag. Attributes are easily accommodated in the above definition of a DTD graph. *For convenience, we assume an attribute is accessible to a user whenever its parent element is.* This assumption can be easily relaxed by treating attributes like subelements.

The next question is what kind of queries? While various query languages have been proposed for XML and, of those, XQuery is essentially emerging as a standard, the basic paradigm in XML query evaluation is finding all matches for a given pattern tree. Indeed, XPath expressions (restricted to `child` and `descendant` axes), which are borrowed into XQuery, essentially correspond to (somewhat restricted) pattern trees. We call these kinds of pattern tree queries as *twig queries*. We gave several examples of twig queries in Section 2. We define them formally next.

Definition 3.2 [Twig Query] A *twig query* is a rooted node-labeled and edge-labeled tree such that (i) its nodes are labeled by element tags, with the exception that its leaves may be labeled by tags or values, (ii) its edges are labeled pc (for parent-child) or ad (for ancestor-descendant), and (iii) a subset of nodes is marked distinguished. □

In our examples, we use double lines for ad-edges and single lines for pc-edges. Leaf labels could be string values such as "cosmicomics" or values of any other base types such as `int`, etc. Distinguished nodes (DNs) correspond to elements returned as query answer. XPath allows exactly one DN in a query, but we don't make this restriction. We refer to the list of DNs of a twig query as its *projection list*.

3.2 Multi-level Security and Authorization

We assume a set of *security levels*, which may be any finite partially ordered set. For simplicity of exposition, we use total orders in all our examples. For instance, in a military environment, it is common to use the security levels $S = \{$unclassified, confidential, secret, top secret$\}$, with the ordering unclassified $<$ confidential $<$ secret $<$ top secret. The security levels used in a commercial application may vary, and depend on the organization's policy.

The basic idea behind multi-level security is that each resource/object (e.g., XML element) is potentially assigned a security level, as is each user/subject that is authorized to access objects in the space (e.g., a collection of XML documents). Here, "user" might include an application or a human. A subject may only access an object if the security level assigned to the subject is no less than that assigned to the object. In general, separate security levels may be associated depending on the kind of access action (e.g., read or write). Since our focus is query processing, we only

consider read actions, and thus leave them implicit. Thus, a subject whose security level is "secret" can access (i.e., read) objects with security level "secret", "confidential", or "unclassified".

In the context of XML data trees, security levels are assigned to objects (elements) by associating a separate attribute called, say `SecurityLevel`. Thus, a DTD needs to permit such attributes in addition to the usual content of elements. Since XML is tree structured, the security level of an element, if not specified, can be inherited from its closest ancestor. On the other hand, security level defined at an element always overrides the inherited one. This overriding can be *monotone*, in that security levels defined at subelements are never less than those defined at their parent, or *non-monotone* where no such restriction is imposed. Clearly, non-monotone overriding is more expressive in that it permits a broader class of applications. For this reason and for brevity, in the sequel, we only consider non-monotone overriding.

3.3 Security Model

It can be tedious, and even unnecessary, to have to specify a value for the `SecurityLevel` attribute at every element's opening tag. To alleviate this burden, and to offer some flexibility, we assume the DTD distinguishes elements (i.e., nodes of DTD graph) into three types:

- **mandatory**: all instances of such an element in a data tree must specify their security level. In practice, the DTD may specify a default value, which an instance is free to override, an issue that need not concern us.
- **optional**: instances of such an element in a data tree may (but don't have to) specify their security level.
- **forbidden**: instances of such an element in a data tree are not allowed to specify their own security level, i.e., it must be inherited, or if it cannot be inherited (e.g., it is the root of a data tree), then it will be assumed to be accessible to every subject.

In Figure 1, we gave an example of a DTD where nodes are classified into one of these three types. We refer to a DTD as *dense* if many of its nodes belong to either the mandatory or optional class and *sparse* if many belong to the forbidden class. The purpose of this classification is to experimentally explore and understand for what kind of DTDs, optimizing secure query evaluation will result in most savings.

Before leaving this section, we point out that while XML documents are trees, the DTD graphs we consider may be DAGs. Thus, at the level of a DTD, an element may inherit its security level from multiple ancestors. This is the case for the element `name` in Figure 1.[1]

4 Secure Query Evaluations and Our Problem

Given a twig query against a collection of XML documents with assigned security levels, how can we evaluate the query so that exactly those answers that the user is supposed to see are returned? In Section 2, we introduced the notion of checking additional predicates potentially at every data tree node that matches a twig query node, by examining the value of the `SecurityLevel` attribute in the data. These are `RC` (recursive check), `LC` (local check), and `NC` (no check). We call these *security check annotation labels* (SC annotation labels). In this section, we make them more precise and suggest an implementation of these checks using XQuery. Finally, we give a formal statement of the problem studied in this paper.

`LC` merely amounts to checking the value of the `SecurityLevel` attribute at a given node, while `NC` is a no-op. `RC`, on the other hand, involves recursively checking the value of this attribute at every node starting from the given node until a nearest ancestor is reached where the attribute is defined. Consider the following function definition in XQuery:

```
DEFINE FUNCTION
self-or-nearestAncestor(element $e)
       RETURNS integer{
   IF $e/@SecurityLevel THEN
      RETURN $e/@SecurityLevel
   ELSE RETURN self-or-nearestAncestor($e/..)
}
```

This function computes the value of the `SecurityLevel` attribute at the nearest ancestor of the current node where it is defined. We can now implement `LC` at a node by adding the predicate [`@SecurityLevel` $\leq$ `$usl OR NOT @SecurityLevel`] to the label of the node. Notice that in a twig query, each node label, i.e., an element tag t, or value v, really stands for the predicate `tag` $= t$, or `content` $= v$. By adding the additional predicate corresponding to `SecurityLevel` above, we really are taking a conjunction of the two predicates. In the same way, we can implement `RC` at a node by adding the predicate [`self-or-nearestAncestor(.)` $\leq$ `$usl`]. As an example, the query of Figure 2(g), with SC annotation labels incorporated, corresponds to the XPath expression `all_accounts[@SecurityLevel` $\leq$ `$usl OR NOT @SecurityLevel][//name] //payment_info[self-or-nearestAncestor(.)` $\leq$ `$usl].` Thus, every twig query with SC annotation labels added can be expressed as a query with function calls added for local and recursive checks

[1] It turns out all ancestors on one of the paths from root to `name` are forbidden to specify their security level, but that is an aside.

and evaluated using the same evaluation engine. To formulate the optimization problem, we define the following:

Definition 4.1 [SC Annotated Query] Let Q be a twig query. A *security check annotation* of Q is essentially Q with each of its nodes associated with one of the SC annotation labels `RC, LC, NC`. Define `NC` $<$ `LC` $<$ `RC`, reflecting the complexity of performing these checks in general. Let Q be a twig query and Q_a, Q_b any two security check annotations of Q. Then define $Q_a \leq Q_b$ provided for every node u of Q, the SC annotation label associated with u by Q_a is $\leq$ that associated by Q_b. □

The intuition behind the above definition is that a security check annotation Q_a is dominated by (i.e., is no more expensive to evaluate than) another security check annotation Q_b provided on a node by node basis, the SC annotation label associated by Q_a is dominated by that associated by Q_b.

Before optimizing security check annotations, we need to characterize what is a safe evaluation of a query. Suppose Q is a twig query with nodes $(u_1, ..., u_k)$, posed by a user with user security level *usl*. Let DB be a database of XML documents with security levels assigned to their elements. Answers to twig queries are obtained by finding *matchings*. A matching is a function that maps Q's nodes to data tree nodes such that all node predicates are satisfied and further whenever nodes u and v in Q are related by a pc-edge (resp., ad-edge), their images are related by a parent/child (resp., ancestor/descendant) relationship. A matching $\eta : Q \rightarrow DB$ binds each node u_i to a data tree node x_i. We call the resulting tuple of bindings $(x_1, ..., x_k)$ a binding tuple. A binding tuple $(x_1, ..., x_k)$ is *safe* provided the security level of every component x_i is no more than *usl*. The *safe answer* to the above query Q is obtained from the set of safe binding tuples and projecting that set onto the set of nodes appearing in the projection list. More precisely, let S be the set of safe binding tuples of query Q, and let $\{u_{i_1}, ..., u_{i_m}\} \subseteq \{u_1, ..., u_k\}$ be the projection list of Q. Then the set of safe answers to Q is the set $\pi_{u_{i_1}, ..., u_{i_m}}(S)$.

For a query Q, we say an SC annotation Q_a of Q is *correct* provided the set of answers returned by evaluating Q_a normally (i.e., as a regular twig query) is equal to the set of safe answers to Q defined above. First, note that there is a trivial way to make sure we get exactly the safe answers: annotate all query nodes with `RC`. This guarantees the evaluation of Q never accesses nodes the user with security level *usl* is not allowed to. However, this can be very expensive and we seek to optimize SC annotations. More precisely, given a twig query, we seek to find an optimal SC annotation of Q, i.e., an SC annotation Q_o such that: (i) Q_o is correct, and (ii) there is no correct SC annotation Q'_o such that $Q'_o < Q_o$. This is the central problem we address in this paper.

5 Tree-Structured DTD Graph

In this section, we consider optimizing SC annotations of twig queries for the case the DTD graph is a tree. In the next section, we briefly deal with DAG-structured DTD graphs. We approach the problem of optimization in the following stages. First, we consider queries containing only pc-edges and then consider queries containing both pc- and ad-edges.

5.1 Parent-Child Queries

Call a twig query a pc-query provided it contains no ad-edges. Intuitively, it should be simple to optimize SC annotations on these queries, since it does not involve examining arbitrarily long paths in the DTD. First, we settle the optimal annotation label for the query root and then determine how the remaining nodes should be optimally annotated. It turns out this kind of separation still allows us to preserve the global optimality of the annotation of the query as a whole.

Root Annotation Label:

Let n be the query root and n' be the corresponding DTD node (which need not be the DTD root). If node n' is classified as mandatory by the DTD, then clearly n's annotation label can be `LC`. Suppose this is not the case. Then n''s status is optional or forbidden. In this case, examine every path from the DTD root to node n'. If there is any path p such that there is a node v' $(\neq n')$ on it whose status is mandatory or optional, then we have no choice but to perform a recursive check at node n, i.e., its annotation label must be `RC` in any correct query annotation. This is because in a data tree, n may or may not have its `SecurityLevel` defined and hence it may need to be inherited from a nearest ancestor where it is defined. Suppose there is no such path p satisfying the above condition, i.e., all nodes on such paths have a forbidden status in the DTD. In this case, the status of n' determines the optimal annotation label for n: if n''s status is optional, then it is `LC`, while if n''s status is forbidden, it is `NC`. The correctness of these assignments should be self-evident. We deal with other nodes in the query tree next. Procedure rootSCA in Figure 3 is the pseudocode for the procedure just described.

Annotation Labels for Other Nodes:

Let x be any node in the query tree, other than the root. Let x' be the corresponding node in the DTD. Whenever the status of x' in the DTD is optional or mandatory, we set the annotation label of x to be `LC`; otherwise we set it to be `NC`.

```
procedure rootSCA
Input:  query root n
Output:  node n with its SC annotation
let n' be the DTDgraph node corresponding to n;
if (mandatory(n')) { set LC(n); return };
if (∃ a DTDpath p from DTDroot to n', ∃ node v'(≠ n')
      on p s.t. (mandatory(v') or optional(v'))) set RC(n);
else {
   if (optional(n')) set LC(n);
   if (forbidden(n')) set NC(n); }
end procedure
```

Figure 3: Procedure `rootSCA`

5.2 General Queries

In this section, we develop an algorithm for optimizing SC annotations of arbitrary twig queries, for the case of tree-structured DTD graphs. The reasoning involved in determining optimal SC annotations for this case is quite non-trivial, even though we assume the DTD is tree-structured. First, we observe that the optimal SC annotation of the query root can be done in a way identical to what we did for PC queries, i.e., call procedure rootSCA (Figure 3).

Nodes of the twig query Q are traversed top-down in some topological order. If the current node n_2 is the query root, it is dealt with as above. Otherwise, let n_2' be the node in the DTD graph corresponding to n_2 (i.e., it has the same tag). Let n_1 be the parent of n_2 in Q and let n_1' be the corresponding DTD graph node.[2] If there is no path p from n_1' to n_2' containing a node v_1', different from n_1', n_2', whose status is mandatory or optional, then n_2 cannot inherit its security level from an ancestor, so if the status of n_2' is either mandatory or optional, we can set its annotation to `LC`, and otherwise to `NC`. The more difficult case is when there is such a path p. In this case, if the status of n_2' is mandatory, again, we can set its annotation to `LC`. If it is optional, then depending on whether there is a unique node in the DTD that specifies the security levels for all data tree instances of n_2' under the data tree instance of n_1', we set the annotation of n_2 to a tentative label, which may subsequently change. These intricacies are best illustrated with examples.

First, consider Figure 1(d). It is easy to see the root's annotation label must be `RC`. Consider the `item` child of the root. Since the DTD path from `customer` to `item` contains `cart` which is optional, the current node must get an `RC`. This is an optimal annotation. Next, consider the only slightly different Figure 1(e). Suppose the topological order used is the root `customer`, followed by the right `item`, followed by the left. The annotation labels of the root and its two children are determined exactly as before, and are set to `RC`. However, when the second `item` child is processed, we notice that it must inherit its security level from an intermediate `cart` node in the DTD path from `customer` to `item`, and all edges on the path segment from `customer` to `cart` (there is only one!) are *not* labeled * or +, so all `item` children must inherit from a unique `cart` node. This means the previously processed `item` node can act as an existential witness for the second `item` node, whose annotation label can thus be replaced by an `NC`. By contrast, for the query of Figure 1(f), which looks identical in structure to that of Figure 1(e), we cannot do this, since the DTD path from `account` to `cart` (the node with a non-forbidden status), has one edge labeled +, which destroys the uniqueness property mentioned above.

To illustrate yet another intricacy and subtlety, consider yet another identical looking query of Figure 1(g). Initially, all three nodes get an `RC`. In particular, suppose we visit node `name` before node `payment_info`, there is no way to know the final annotation label. However, we can reason that in the DTD, the `name` node, being of forbidden status, must inherit its security level from the ancestor `account`. So, if the `payment_info` node in the query has an `RC` on it (as it does), it would ensure an accessible `account` ancestor. Since the DTD path from the `account` node to `name` node is free from edges labeled ? or *, we know there is guaranteed to be a `name` descendant, which must be accessible, since its security level is inherited from the `account` ancestor, which is accessible. So, in the query, the `name` node's annotation label can be changed to an `NC`. We leave it to the reader as a simple exercise to see why in yet another similar query, Figure 1(h), none of the child labels can be changed from an `RC`.

Algorithm `ForwardPassTree` of Figure 4 presents pseudo-code for optimizing SC annotations of arbitrary XML twig queries, for the case of tree-structured DTD graphs. By using case analysis, we can show the following result.

Theorem 5.1 (Optimality of `ForwardPassTree`) *Let Q be an arbitrary XML twig query, and D be any tree-structured DTD graph. Algorithm* `ForwardPassTree` *correctly optimizes the SC annotations of Q on D. Further, it is* optimal *in that if Q_a is the SC annotation of Q computed by Algorithm* `ForwardPassTree`*, then $\nexists$ SC annotation $Q_b (\neq Q_a)$ of Q such that $Q_b \leq Q_a$.*

6 DAG-structured DTD-graphs

In extending our `ForwardPass` algorithm to deal with DAG-structured DTDs, we need to allow for the fact that an element in the DTD graph may inherit its security level from any one of a number of different elements, because multiple DTDpaths are possible between the same pair of nodes. For example, the DTD graph of Figure 1 shows that there are two paths between `online_seller` and `name` elements in the DTD. If the `book` elements, for example, were permitted to

[2] Note that the edge (n_1, n_2) might be an ad-edge.

```
Algorithm ForwardPassTree
Input:  query Q
Output:  query Q with SC annotation
for (n ∈ topological order of nodes(Q)) {
   if (queryroot(n)) rootSCA(n);
   else nonrootSCA(n);
}
end Algorithm

procedure nonrootSCA
Input:  non-query root node n2
Output:  node n2 with its SC annotation
let n2' be the DTDgraph node corresponding to n2;
let n1 = parent(n2) in Q, and n1' be its DTDgraph node;
   /* edge between n1 and n2 could be anc-desc */
if (∃ a DTDpath p from n1' to n2', ∃ node v1'(≠ n1', n2')
      on p s.t. (mandatory(v1') or optional(v1'))) {
   if (mandatory(n2')) set LC(n2);
   else if (⟨?, setIh(n2')⟩ is not in ListIh) {
      set RC(n2); /* this may change later */
      if (forbidden(n2') and (uniqueIhNode(n1', n2') or
            (existentialWitness(n2)))) {
         add ⟨n2, n1⟩ to canWit; }
      else if (optional(n2') and uniqueIhNode(n1', n2')) {
         add ⟨n2, n1⟩ to canWit; } }
   else {
      if (uniqueIhNode(n1', n2')) {
         if (forbidden(n2')) set NC(n2);
         if (optional(n2')) set LC(n2);
         set simplified(n2); }
      else if (existentialWitness(n2)) {
         for (⟨m3', setIh(n2')⟩ in ListIh) {
            if (forbidden(n2') and isWitness(m3', n2')) {
               set NC(n2); set simplified(n2); }
         } }
      else set RC(n2); }
} else {
   if (mandatory(n2') or optional(n2')) set LC(n2);
   if (forbidden(n2')) set NC(n2); }
/* update realIh and ListIh */
if (forbidden(n2')) set realIh(n2') = setIh(n2');
else set realIh(n2') = {n2'};
add ⟨n2', realIh(n2')⟩ to ListIh;
/* check candidates for modification to SC annotation */
if (!(simplified(n2))) {
   for (⟨m2, m1⟩ ∈ CanWit) {
      if (uniqueIhNode(m1', m2') and
            (setIh(m2') = realIh(n2'))) {
         if (optional(m2')) set LC(m2);
         if (forbidden(m2')) set NC(m2);
         remove ⟨m2, m1⟩ from CanWit; }
      else if ((existentialWitness(m2)) and
            forbidden(m2') and isWitness(n2', m2')) {
         set NC(m2);
         remove ⟨m2, m1⟩ from CanWit; } }
}
end procedure

function uniqueIhNode
Input:  DTD nodes n1', n2'
Output:  checks if there is a unique node that specifies
   security levels for all n2 under a given n1
for (v' ∈ setIh(n2')) {
   if (∃ a DTDpath p1 from n1' to v', and ∃ edge e on p1,
         s.t. (label(e) = '+' or label(e) = '*')) {
      return FALSE; } }
return TRUE;
end function

function isWitness
Input:  DTD nodes n2', m2'
Output:  checks if m2' is a witness, given an n2'
if (∀v1' ∈ realIh(n2'), ∃ a DTDpath p1 from v1' to m2', s.t.
   ((∀ nodes v2' (≠ m2') on p1, forbidden(v2')) and
([define v3' on path p1 as the lowest node from which
n2' and m2' branch] (all edges on the path from v3'
to m2' satisfy (label(e) = '+' or
label(e) = '1'))))) { return TRUE; }
return FALSE;
end function
```

Figure 4: Algorithm `ForwardPassTree`

have security levels, different `name` elements in the database may inherit their security levels from either `book` elements or `account` elements.

In this section, we highlight the key differences between dealing with tree-structured and DAG-structured DTD-graphs. First, since the inheritance sets, `setIh`, of different elements may partially overlap, it does not suffice to check (in Procedure `nonrootSCA`) whether or not an element n's `setIh`(n') is present in `ListIh`; one needs to use containment and overlap tests instead. Second, it is possible that a query element n's SC annotation can be optimized when a set of other elements are present in the query, but cannot be optimized in the presence of any subset of these elements. This happens, for example, when each member of the inheritance set of n' is a unique node; in this case n's SC annotation can be simplified if, for example, each of the nodes in this inheritance set is matched in the query. Third, for a query element n to be an existential witness to an element n_2, n would need to be an existential witness to n_2 for *each* of the possible DTD paths along which n_2 may be instantiated.

The task of incorporating these changes into the `ForwardPass` algorithm is simplified because the basic logic of the various procedures and functions in Figure 4 are already specified in terms of members of `setIh`, and conditions that need to hold along *some* path and nodes/edges along this path, or along *all* paths. We omit a detailed presentation of the algorithm for reasons of space.

7 Experimental Results

7.1 Experimental Setup

We ran our experiments over the `XMark` benchmark dataset and the `HL7` clinical dataset. The DTDs of `XMark` and `HL7` are found at `http://monetdb.cwi.nl/xml/` and `http://www.hl7.org`, respectively. We constructed two very different DTD access specifications for `XMark`: one in which only 3 element types are permitted to have a security level (we call this a *sparse DTD*); in the other DTD, half of the element types are permitted to have a security level (a *dense DTD*); all are optional, none are mandatory.

Two `XMark` synthetic datasets were created based on these DTDs; all elements with an optional security level were assigned values for their `SecurityLevel` attribute. The `HL7` dataset was designed so that 25% of the `patient` elements have a "secure" security level, and the other 75% are "unsecure". For these unsecure patients, the `observation` elements are divided equally into "secure" and "unsecure" security levels. The user is designated as "unsecure". For query evaluation we used `XALAN`, found at `http://xml.apache.org`, because of its support for

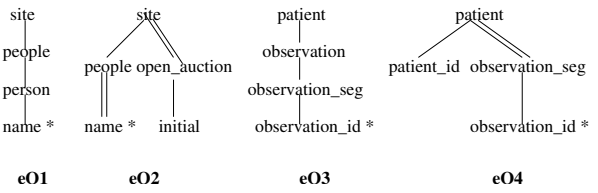


Figure 5: Queries Used in the Experiments

external functions, which is needed for evaluating our security check annotations. Our optimizer was implemented on top of `XALAN`.

Our experiments were run on a lightly loaded Unix machine with 128 MBytes of memory. In all experiments, time is reported in seconds.

7.2 Queries

We used the queries of Figure 5. The query eQ_1 and eQ_2 are provided by `XMark`. The synthetic queries, eQ_3 and eQ_4 for `HL7` are created. eQ_1 and eQ_3 are path queries with only parent-child edges, and eQ_2 and eQ_4 are branching queries with both parent-child and ancestor-descendant edges.

In the case of `XMark`, the `dense DTD` specifies that, of the elements mentioned in eQ_1 and eQ_2, the elements `people`, `person` and `open_auction` can have values for the `SecurityLevel` attribute, while the `sparse DTD` specifies that only the element `people` can have such values. When the dense DTD is used, the optimal eQ_1 is that `people` and `person` are annotated `LC` and others are annotated `NC`, and the optimal eQ_2 is that `people` and `open_auction` are annotated `LC`, and others are annotated `NC`. With the sparse DTD, in the optimal query eQ_1, `people` is annotated `LC` and others are annotated `NC`, and the optimal query eQ_2 is that `people` is annotated `LC` and others are `NC`.

In the case of `HL7`, the elements `patient` and `observation` (both mentioned in the queries) can have values for the `SecurityLevel` attribute. The optimal query eQ_3 is that `patient` and `observation` are `LC` and others are `NC`, and in the optimal eQ_4, `patient` is `LC`, `observation_seg` is `RC`, and others are `NC`.

7.3 Impact of Our Optimization

We varied the size of datasets from 5MB to 30MB. The size of the secure answer for each query is the same over varying datasets.

Figures 6(a), (b) and 7(a) report query evaluation times. The graphs of Figures 6(a) and (b) are the results of `XMark` and report four curves each: the optimal eQ_1 (resp., eQ_2) for the dense DTD and the sparse DTD and non-optimal (with `RC` annotations at all nodes) eQ_1 (resp., eQ_2) for the dense DTD and the sparse DTD. Figure7(a) is the result of `HL7` and reports two curves: the optimal and non-optimal (with `RC` annotations at all nodes) eQ_3. The result of eQ_4 was very similar, and we omit the results for reasons of space.

There are two significant points worth noting. Consider Figure 6(a), i.e., evaluation times for eQ_1. First, the benefits of our optimization are substantial. For example, for the 30Mb dataset, the unoptimized secure evaluation (i.e., with RCs at each node) for the sparse DTD took 5s, whereas the optimized secure evaluation took only 1.25s, a savings of 75%! Further, since query evaluation without performing *any* security checks took 1s, the optimized secure evaluation did not add too much overhead. Second, the benefits of our optimization are more for the case of sparse DTDs than for the case of dense DTDs. The reason for this is as follows. The unoptimized secure evaluation is more expensive when the data is sparsely annotated than when it is densely annotated, since (in the sparse case) each recursive check has to traverse more parent pointers, on the average, to reach a node at which the `SecurityLevel` attribute specifies a value. However, the optimized secure evaluation is cheaper for sparse DTDs, since our optimization algorithm infers that many more nodes can be annotated with `NC` and `LC` than in the dense case. A similar behavior is also observed for the twig query eQ_2 in Figure 6(b).

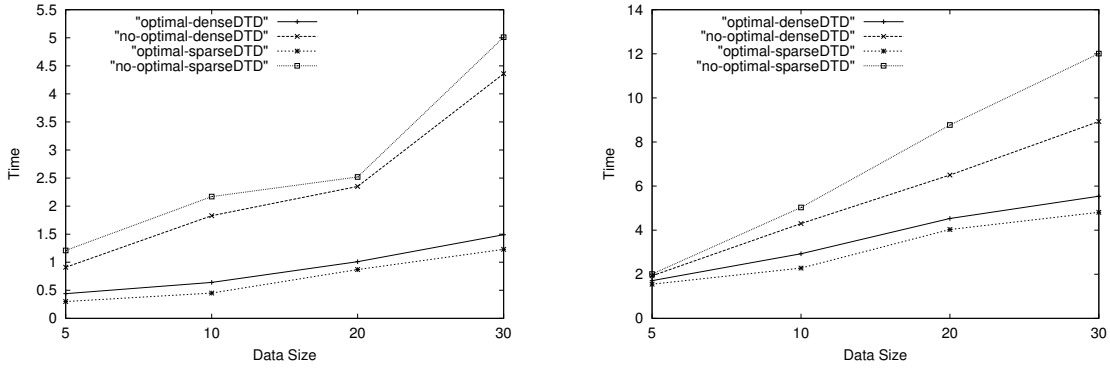
The dataset of `HL7` contains many branch instances, compared to `XMark` dataset, resulting in the higher overall evaluation times. However, the trends in terms of the benefits due to our optimization technique are similar to the `XMark` dataset.

It is important to note that we used the *unmodified* `XALAN` query evaluation, for our experiments, relying solely on query rewriting, outside the evaluation engine to implement our optimization strategy.

7.4 Optimization Overhead

The second experiment measures the actual optimization time. We ran the two queries eQ_1 and eQ_2, additionally varying the number of nodes from 10 to 30 in the queries. We ran these queries over the sparse and dense `XMark` DTDs. Figure 7(b) shows the optimization times in these cases.

The first thing to note is that the optimization time for the eQ_1 family of queries, involving just parent-child edges, depends only on the number of nodes in the query, and is independent of the type of the DTD, since all decisions about SC annotations are purely local. For this reason, we plot just a single curve for this parent-child case in Figure 7(b). For the case of the eQ_2 family, which has ancestor-descendant edges, the DTD type matters. The sparse DTD took a bit more time than the dense DTD, essentially because our query optimization algorithm had to traverse more edges in the DTD graph to locate the first node that had the `SecurityLevel` attribute defined. (In the case of paths where the attribute is not defined, our algorithm had to touch each edge on the path.) More im-



(a) eQ1 for XMark

(b) eQ2 for XMark

Figure 6: Evaluation Times

portantly, though, the optimization times are a very small fraction of the query evaluation time. Even for queries with 20-30 nodes, the optimization time was under 0.00025s.

8 Related Work

Several ideas for supporting access control on the Web have been recently proposed. Research on XML security is mainly divided into two complementary branches: (i) modeling and the development of sophisticated access control models and mechanisms in order to support different security requirements and multiple policies; (ii) lower level features such as encryption and digital signatures. At the same time, the security community is tending toward representing authorization-based or role-based access control using XML syntax and DTD by making use of the expressive power of XML for representing complex tree-structured and heterogeneous data.

Recent work by [10, 9, 2, 3] has proposed authorization-based access control, and all models assume that the system either authorizes the access request or denies it. In particular, Damiani et al. [10, 9] proposed an access control model for XML documents and schema, which is an extension of their early work that defines positive and negative authorization and authorization propagation on object-oriented database concepts. In particular, they develop an approach for expressing access control policies using XML syntax. The semantics of access control to a user is a particular view of the documents determined by the relevant access control rules. They provide a novel algorithm for computing the view using tree labeling. However, these view definitions can often be quite complex and expensive to compute and maintain. The work in [2, 3] is similar, and is implemented in the Author-X prototype.

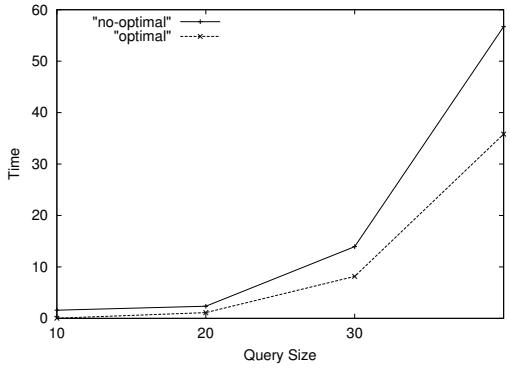
In the same framework, [12] stressed richer authorization decisions by incorporating the notion of provisional authorization model into traditional authorization semantics. The provisional model provides XML with a sophisticated access control mechanism. It presents an XML access control language (XACL) that integrates security features such as authorization, non-repudiation, confidentiality, and an audit trail for XML documents.

Our focus in this paper is finding optimal and safe rewritings of queries that ensure secure evaluation given a DTD graph of the database of XML documents. The rewriting is accomplished by means of adorning query tree nodes with security check annotation labels that infer the security level applicable to a given node. Thus, our contributions are different from, but complementary to the body of work on access control models and policies above.

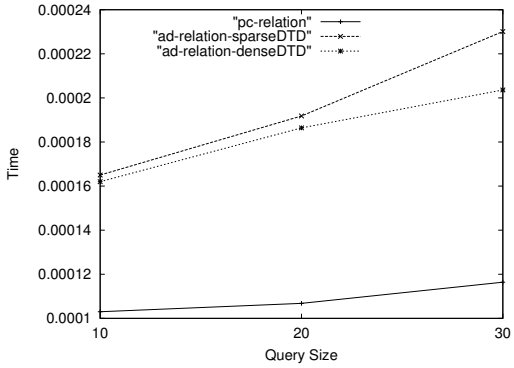
In [14, 15], the authors proposed query rewriting in XML or semi-structured data using a DTD. In particular, [14] presented an algorithm, based on generalizing containment mappings, the chase, and unification, from a given semistructured query and a set of views. The goal of these papers was to find minimal equivalent queries by eliminating query tree nodes. By contrast, the goal of query rewriting in this paper is to minimize the number of and kinds of security checks that need to be performed at evaluation time thus optimizing secure query evaluation.

9 Conclusions

In this paper, we have considered the problem of secure evaluation of XML twig queries, for the simple, but useful, multi-level security model. Our main contribution is an efficient algorithm that determines an optimal set of security check annotations with twig query nodes, by analyzing the subtle interactions between inheritance of security levels and the paths in the DTD graph. We experimentally validated our algorithm, and demonstrated the performance benefits to optimizing the secure evaluation of twig queries.



(a) eQ3 for HL7



(b) Optimization Time

Figure 7: Evaluation and Optimization Times

Ours is the first work in this important area of secure evaluation of XML twig queries, and establishes the foundation for additional work in this area.

References

[1] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, and Y. Wu. Structural joins: A primitive for efficient XML query pattern matching. In *ICDE*, 2002.

[2] E. Bertino, M. Braun, S. Castano, E. Ferrari, and M. Mesiti. AuthorX: A Java-based system for XML data protection. In *Proc. of the 14th Annual IFIP WG 11.3 Working Conference on Database Security*, Schoorl, The Netherlands, Aug. 2000.

[3] E. Bertino, S. Castano, and E. Ferrari. Securing XML documents with Author-X. *IEEE Internet Computing*, 5(3):21–31, 2001.

[4] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Controlled access and dissemination of XML documents. In *Workshop on Web Information and Data Management*, pages 22–27, 1999.

[5] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Specifying and enforcing access control policies for XML document sources. *World Wide Web Journal (Baltzer Publ.)*, 3(3), 2000.

[6] S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu. XQuery 1.0: An XML query language. W3C Working Draft. Available from http://www.w3.org/TR/xquery, Dec. 2001.

[7] D. D. Chamberlin, J. Robie, and D. Florescu. Quilt: An XML query language for heterogeneous data sources. In *WebDB (Informal Proceedings)*, pages 53–62, 2000.

[8] B. F. Cooper, N. Sample, M. J. Franklin, G. R. Hjaltason, and M. Shadmon. A fast index for semistructured data. In *VLDB*, 2001.

[9] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Design and implementation of an access control processor for XML documents. *Computer Networks*, 33(1–6):59–75, 2000. Also in WWW9.

[10] E. Damiani, S. D. C. di Vimercati, S. Paraboschi, and P. Samarati. Securing XML documents. In *Proceedings of the International Conference on Extending Database Technology*, 2000.

[11] S. Jajodia and R. Sandhu. Toward a multilevel secure relational data model. In *PODS*, 1991.

[12] M. Kudo and S. Hada. XML document security based on provisional authorization. In *ACM Conf. Computer and Communications Security*, 2000.

[13] J. McHugh and J. Widom. Query optimization for XML. In *VLDB*, 1999.

[14] Y. Papakonstantinou and V. Vassalos. Query rewriting for semi-structured data. In *SIGMOD*, 1999.

[15] P. T. Wood. Optimizing web queries using document type definitions. In *ACM CIKM'99 2nd International Workshop on Web Information and Data Management (WIDM'99)*, 1999.

[16] C. Zhang, J. Naughton, D. Dewitt, Q. Luo, and G. Lohman. On supporting containment queries in relational database management systems. In *SIGMOD*, 2001.