Compact Access Control Labeling for Efficient Secure XML Query Evaluation

Huaxin Zhang Ning Zhang Kenneth Salem Donghui Zhuo

University of Waterloo, {h7zhang,nzhang,kmsalem,dhzhuo}@cs.uwaterloo.ca

Abstract

Fine-grained access controls for XML define access privileges at the granularity of individual XML nodes. In this paper, we present a fine-grained access control mechanism for XML data. This mechanism exploits the structural locality of access rights as well as correlations among the access rights of different users to produce a compact physical encoding of the access control data. This encoding can be constructed using a single pass over a labeled XML database. It is block-oriented and suitable for use in secondary storage. We show how this access control mechanism can be integrated with a next-of-kin (NoK) XML query processor to provide efficient, secure query evaluation. The key idea is that the structural information of the nodes and their encoded access controls are stored together, allowing the access privileges to be checked efficiently. Our evaluation shows that the access control mechanism introduces little overhead into the query evaluation process.

Key words: Access Control, XML, Query Evaluation, Optimization, Multiuser

1 Introduction

Access controls in relational databases are usually defined at a coarse granularity, e.g. on entire tables. In contrast, much of the work on XML access controls assumes a fine-grained model in which access controls can be specified for individual nodes. Fine-grained access control for XML gives rise to several challenges. First, the access rights specification is potentially very large: proportional to both the size of the database itself and the number of database system users. Thus, there must be a reasonably simple mechanism for specifying these rights and there must be a compact way to store them. Second, access controls must be implemented efficiently, since processing even a single query may involve many access right checks. This is not normally an issue when access controls are coarse-grained. For example, in relational database systems, access controls are normally checked once before a query is processed.

If all of the database objects (e.g. relations) on which a query depends are accessible, then the query is processed, otherwise it is rejected. In contrast, most fine-grained access control mechanisms never reject a query. Each query is answered using that portion of the database that is accessible to the user that submits the query. As a result, access controls and query evaluation are very closely related.

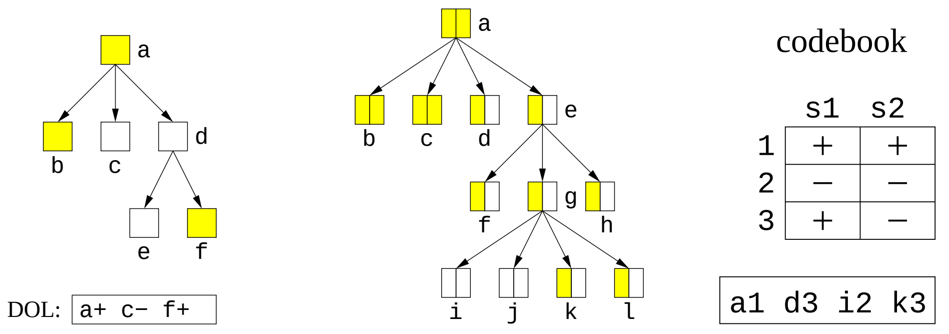
There is a plethora of literature on specifying fine-grained access controls on XML data using high level languages. Instead of manually specifying access control for each XML node,the system administrator defines a set of rules and derives access controls for each node in the XML document through rule-based propagation and inferences. However, since evaluating the rules at run-time is costly, it is desirable to materialize the net effect of these access control rules into incrementally maintainable accessibility maps, in which each XML node is labeled as either accessible or non-accessible for each subject under each action mode[1,2]. The efficient storage and use of such maps is the problem that we address in this paper.

Some recent work [3] involves schema-based access control specifications, which do not require instance-level accessibility maps. However, schema-based approaches are not always applicable, since the schema may be non-existent, or inappropriate for the specification of the desired access controls. Additional discussion of related work can be found in Section 7.

In this paper we present a simple scheme called Document Ordered Labeling (DOL) for the compact representation of fine-grained access control information. Like Compressed Accessibility Maps (CAMs) [4], a recently proposed scheme for representing XML access control data, DOL exploits the structural locality of the access control data to achieve compression. Unlike CAM, DOL is also able to exploit correlations among the access rights of different users to achieve a substantial amount of additional compression in multi-user environments. DOL is a disk-oriented, multi-user scheme, while a CAM is intended to store a single user's access control data in memory.

The DOL access control representation is highly compatible with next-of-kin (NoK) pattern matching, which is an efficient technique for processing XML twig queries [5]. NoK query processing uses a compact representation of document structure to evaluate some kinds of structural query constraints (e.g. parent/child relationships) very efficiently. In this paper, we show how to implement secure twig query processing by integrating DOL-based access control with NoK query processing. We have used both real and synthetic access control data to evaluate the DOL technique. In terms of space efficiency, our results show that a single-user DOL is somewhat less compact than a single-user CAM. However, in a multi-user environment the DOL representation is much more compact than a set of per-user CAMs. In terms of query processing time, we found that multi-user secure twig query evaluation with DOL and NoK is at most 20% more expensive than unsecured evaluation with NoK alone.

2 DOL: Access Control Labeling



(a) one user and his DOL (b) two users (c) DOL of two users

Fig. 1. XML data with fine-grained access control and its DOL

We model an XML document as a tree in which the nodes correspond to the document's elements and the edges represent parent/child relationships among the elements. Sibling nodes in the tree are ordered. Our fine-grained access control model consists of a set of subjects (denoted by $\mathcal{S}$), a set $\mathcal{M}$ of action modes (such as read and write), and a set $\mathcal{D}$ of nodes in the XML tree whose access is to be controlled. Here we use *subjects* to denote both *users* and *user groups*. The subject hierarchy, which describes group membership, is assumed to be maintained separately.

We model the net effect of a set of access control policies over a database instance as an accessibility function

$$\text{accessible}(\mathcal{S} \times \mathcal{M} \times \mathcal{D}) \to \{\text{true}, \text{false}\}$$

The accessibility function specifies whether a given subject can access a given data item in a given action mode. The accessibility function is often represented as an *access control matrix* [6]. For XML data, the accessibility function for a given action mode can also be represented by associating each XML node a list of subjects able to access it for that action mode. We will refer to an XML tree without access control labels as a *data tree*, and to a tree with access control labels as a *secured tree*.

We will initially assume that there is only one single action mode (i.e., $|\mathcal{M}| = 1$). We will relax this assumption in Section 5. We first present the DOL scheme for a single subject, and then show how to generalize it to multiple subjects. Figure 1(a) shows a secured tree for a single subject, and the corresponding

DOL representation of the subject's access rights. Shaded nodes are accessible to the subject, unshaded nodes are non-accessible. We define a *transition node* to be a secured tree node whose accessibility is different from its document-order predecessor (i.e., its immediately preceding node in document-order). As a special case, the root node of a secured tree is also considered to be a transition node. The DOL corresponding to a given secured tree is simply a list, in document order, of the tree's transition nodes, together with their accessibilities. In the DOL shown in Figure 1(a), accessible and non-accessible transition nodes are labeled with "+" and "−", respectively.

Document order is, of course, one of many possible ordering one could choose. We have chosen document order for several reasons. First, since XML parsers and other tools process XML data in document order, a document order encoding of access rights can be constructed on-the-fly using a single pass through a labeled XML document. Second, NoK query processing uses a document order encoding of document structure, and we want DOL to be compatible with NoK. Finally, and most importantly, it allows structural locality of access controls to be exploited to reduce their size. The terms "vertical" and "horizontal" locality have been used to describe locality among parent/child nodes and among sibling nodes, respectively [4]. Structural locality is encouraged by access control specifications that propagate access rights along the hierarchical structure of the XML data, and it has been observed in real access control data [4]. Nodes that are adjacent in document order often have parent/child or sibling relationships in the document. Although this is not always the case, we expect that much of the structural locality that exists in a document's access controls will translate to locality in document order. Such locality will reduce the number of transition nodes, and hence the size of the DOL. In Section 6 we measure the impact of this locality on the size of the DOL using several access control datasets.

2.1 DOL for Multiple Subjects

Aside from the structural locality of access controls for a single subject, we conjecture that different subjects in an access control system may exhibit correlated access constraints. (There may also exist correlations among action modes, we will discuss this in Section 5.) For example, users in the same department may have similar access controls. We wish to further compress the access control labeling by taking advantage of such correlations.

Figure 1(b) shows a tree labeled with access rights for two subjects. In the figure, each node is divided into two parts, with the left part representing the access rights of one subject and the right part representing the access rights of the other. As was the case in Figure 1(a), shading represents accessibility. For example, node `e` is accessible to the first subject but not to the second.

We can encode these access rights in much the same way as we did for a single user, by recording a list of transition nodes. With each transition node we record its access control list. Thus, when several consecutive nodes have the same access control list, we only record it once. Furthermore, we expect the access control lists for the transition nodes will reoccur frequently throughout the secured tree. We can exploit this using dictionary compression: each distinct access control list that appears in the secured tree is recorded once in a codebook (dictionary). With each transition node in the DOL we record a reference to the appropriate access control list in the codebook, rather than the access control list itself.

Figure 1(c) shows the multi-user DOL that corresponds to the secured tree in Figure 1(b). Each transition node in the list has a numeric index. This is the *access control code* (index into the codebook) for that transition node. The codebook itself contains three entries, because only three of the four possible distinct access control lists actually appear in the secured tree. Each codebook entry is an access control list, which we present as a bit vector with one bit for each access control subject.

The overall storage cost of DOL includes the distinct access control lists (the codebook), the transition nodes and their associated codes. The number of distinct access control lists and transition nodes depends on the correlations among subjects' access controls. If the access controls are not closely correlated, the transition nodes will be dense and the number of codebook entries will be large. Suppose we have $|S|$ single subject DOLs, each having T transition nodes (in reality, each DOL would have a different number of transition nodes, but we simplify this here). In the worst case, when subjects access controls are independent, the number of distinct access control codes in the combined multi-subject DOL would grow exponentially with the number of subjects until it reaches an upper bound of $min(|\mathcal{D}|, 2^{|\mathcal{S}|})$. Meanwhile, the number of non-transition nodes would be:

$$|\mathcal{D}| \times (1 - \frac{T}{|\mathcal{D}|})^{|\mathcal{S}|}$$

That is, as S goes up, the number of non-transition nodes would shrink exponentially until each XML node becomes a transition node. In practice, however, we expect the situation to be much better than this worst case. The real access control systems that we have studied do not exhibit worst case behavior. We will see in Section 6 that there *do* exist strong correlations of access controls among subjects in these systems that make the overall size of DOL grow slowly as the number of subjects increases.

3 Physical Representation of the DOL

In this section we describe our physical representation of the DOL, which is intended to be incorporated into an existing query processor framework called NoK [5] to optimize secure query evaluation. For this reason, we begin with a brief overview of NoK query processing.

3.1 NoK Query Modeling and Physical Storage

A NoK query processor accepts twig queries described by pattern trees and evaluates them against an XML document by pattern matching. Each successful pattern match generates a set of bindings between pattern tree nodes and data tree nodes. The query result consists of all possible bindings. For example, the pattern tree in Figure 2(a) will generate one match from the data tree in Figure 2(c).

The NoK query processor first partitions the pattern tree into *NoK* subtrees, each having only parent-child or following-sibling relationships (the so-called "next-of-kin" relationships) among its nodes. Then the processor finds matches for these NoK subtrees from the data tree. Finally it matches the ancestor-descendant relationships using structural joins. For example, the pattern tree in Figure 2 would be split into two NoK subtrees, each matches to a fragment in the data tree. The two fragments are then connected by the ancestor-descendant relationship between nodes a and h.

The NoK query processor uses a physical representation of the data tree that allows it to match NoK subqueries very efficiently. The structure of the data

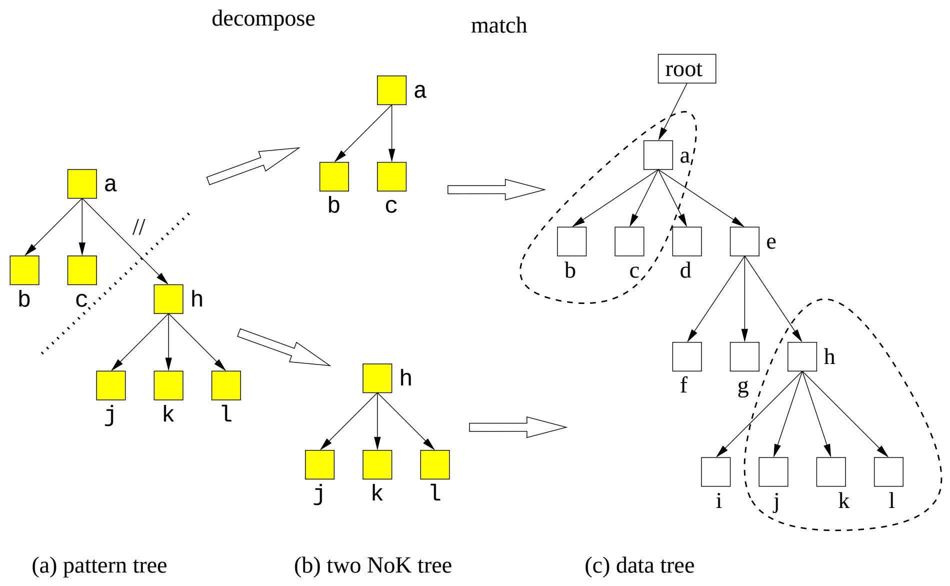
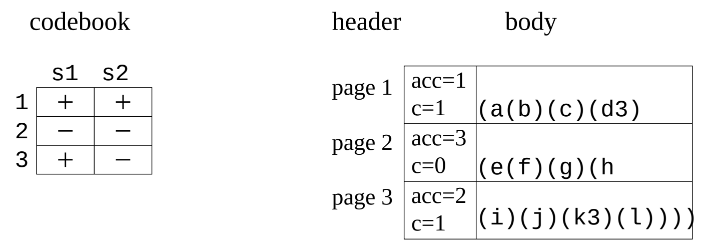


Fig. 2. A Pattern Tree with two NoK trees matched to data tree

tree is stored separately from the node values in a compact representation. The structure is encoded by listing the nodes in document order, with embedded markup to indicate where subtrees begin and end. For example, the structure of the data tree of Figure 2 would be encoded using the following string: $(a(b)(c)(d)(e(f)(g)(h(i)(j)(k)(l))))$, where the nesting of parentheses captures the nesting of subtrees. This document-order string is decomposed into blocks for storage on disk. Each block has a header with meta data for that page (e.g., the number of nesting parentheses for the first node in the page).

It can be seen that nodes connected by "next-of-kin" relationships tend to be clustered in this physical representation and thus such nodes are more likely to be located in the same physical block. By keeping the block meta data in memory and exploiting this clustered representation of document structure, a NoK query processor can match a NoK pattern using a small number of I/O operations[5].

3.2 Integrating Access Control Data



codebook

	s1	s2
1	+	+
2	−	−
3	+	−

	header	body
page 1	acc=1 c=1	(a(b)(c)(d3)
page 2	acc=3 c=0	(e(f)(g)(h
page 3	acc=2 c=1	(i)(j)(k3)(l))))

Fig. 3. DOL at physical level

To integrate access control with NoK query processing, we physically cluster the access control data with the NoK structural data. Specifically, our scheme for physical representation of the access control data consists of the following three components (Figure 3):

- The DOL codebook is maintained in memory for fast accessibility lookup. If the codebook grows beyond the capacity of memory, each accessibility lookup may result in an extra physical page read for loading the codebook entry. However, our results in Section 6 show that, in practice, the codebook will be quite small.
- The DOL transition nodes are embedded into the NoK structural data. Figure 3 shows the embedding for the secured tree of Figures 1(b) (the page header does not show NoK meta-data for simplicity). For the purposes of the illustration, we have assumed that these data are spread across three disk blocks. In the physical encoding, we treat the first node in each block as if it were a transition node, regardless of whether it is actually a transition node. The access control code for this initial transition node is stored in the block header, which is described next. These initial transition nodes ensure

that we can determine the access rights of any node using only the codes in that node's block.

- For each disk block, there is a small access control header which contains two items. The first is the access control code for the first data node in the block. The second is a "change" bit which is set if there is at least one transition node (other than the initial node) in the block, and cleared otherwise. The aggregate size of these headers is small. For example, for 1TB of XMark data, the NoK query processor will first convert it to a succinct physical layout as in [5], which only occupies about 10^7 blocks, assuming a 4KB block size. Measurements from our data sets (Section 6) suggest that 2 bytes is more than sufficient for an access control code. If we conservatively assume 4 bytes per access control block header, the total size of the headers is only 40 MB. Therefore, we can keep all of the block headers in memory (or part of the headers that include enough information for the current pages to be read), and the NoK query processor can further optimize I/O operations, as we shall describe shortly.

3.3 Access Lookup

To check the accessibility of a node d for subject s, the query processor locates the transition node that precedes node d (if d is not itself a transition node). Since the first node in every block is a transition node, the transition node will be found in d's block. That transition node's access control code is then used to identify an entry in the in-memory access control codebook. The s-th bit in that codebook entry indicates the accessibility of the node for subject s. As we will see in Section 4, the NoK query processor checks nodes' accessibility while it matches NoK query patterns. Provided that d's disk block has been loaded for query evaluation by the NoK evaluator, the access control check for d requires no additional I/O.

In some cases, the query processor can make use of the in-memory DOL page header to avoid unnecessary page reading: if the starting transition node in the header indicates non-accessible to the user, and the "change" bit in the header is not set (meaning there is no other transition nodes in this page), all nodes in that page are non-accessible to the user. Thus the query processor can entirely avoid loading that page.

3.4 DOL Updates

We consider two types of access control update operations: the updates to the accessibility of nodes (e.g., adding read permission for a given subject to a node), and updates to the subjects.

We first consider how to change the accessibility of a single node, x. Suppose we are to make x "accessible" for a certain subject. We first locate the nearest preceding transition node (or the node itself if it is a transition node). We will denote this node by $\hat{x}$. If $\hat{x}$'s access control code indicates "accessible" for the subject, we stop. Otherwise, we mark x as a new transition node and set its access control list to be the same as that of $\hat{x}$, except that x is accessible to the specified subject. We need to find this access control list in the codebook, and assign its access control code to node x. If the required access control list is not already in the codebook, then we need to add it. Finally, if the node immediately after x is not already a transition node, we mark it as a transition node with an access control code equal to that of $\hat{x}$. All these operations occur in memory after loading the page containing x. Note that $\hat{x}$ is guaranteed to be on the same page as x, because the first node in every page is always a transition node. For the same reason, if the node after x is on a different page than x then it must also be a transition node, which will not require modification. Thus the I/O cost for updating x's access control list is a page read followed by a page write flushing the updates to disk.

To change the accessibility of all of the nodes in a subtree rooted at node x, we can use the above procedure to change the accessibility of each node in the subtree, in document order. The I/O cost of such a bulk update will be much smaller than one read and one write per subtree node. This is because the physical representation clusters these nodes. If each page can hold B nodes, the total I/O cost for updating the accessibility of a subtree with N nodes will be N/B pages reads (and writes). It is worth noting that all updates to DOL have the *update locality* property, i.e., an update to a subtree only affects the nodes within the pair of transition nodes that surround the subtree. This property guarantees that updates are confined within a contiguous region of the affected data.

In addition to the updates to the XML data, we need to consider updates to $\mathcal{S}$, the set of access control subjects. With DOL, it is relatively simple to add a new subject who has no (initial) access rights, or whose access rights initially match those of some existing subject. This can be accomplished by simply adding an additional column to each entry in the in-memory codebook. No changes to the embedded transition nodes and the references are required. Deletion of a subject can also by accomplished within the codebook. This may leave unnecessary codes embedded in the structural data, since the deletion of a subject may decrease the number of transition nodes. However, any such redundancy can be corrected lazily.

4 Secure Query Evaluation

Our semantics for secure query evaluation are identical to those used by Cho et al. [7]. Recall that the (unsecured) evaluation of a twig query Q returns all of the possible sets of bindings of query pattern nodes to data nodes. Secure evaluation of Q for subject s eliminates from this result any sets of bindings that include data nodes that are inaccessible to s. For example, the pattern tree shown in Figure 2 will return a single set of bindings if nodes a, b, c, h, j, k and l in the data tree are all accessible to the subject s. It will return no bindings if any of those nodes are inaccessible to s. Note that the accessibility of nodes d, e, f, g and i has no impact on the secure evaluation of the particular query shown in Figure 2.

4.1 DOL for Secure NoK Pattern Matching

In Section 3.1 we described that a NoK query processor works by first decomposing a pattern tree into NoK subtrees, and then attempting to match each NoK subtree to the data (by using B+ trees on the subtree root's value or tag names to start the matching). One node in the NoK pattern tree is set as the *returning node*, which means the nodes in the data tree that match this node should be returned as result of this pattern matching.

Algorithm 1. ε-NoK Pattern Matching

```
NoK-Pattern-Matching($proot$, $sroot$, $R$)
     Pre-condition: sroot is accessible
 1: if $proot$ is the returning node
 2:     then List-Append($R$, $sroot$);
 3: $S \leftarrow$ all children of $proot$;
 4: $u \leftarrow$ First-Child($sroot$);
 5: repeat
 6:         if ACCESS($u$) = TRUE
 7:            then for each $s \in S$ that matches $u$ with
                          both tag name and value constraints
 8:                    do
 9:                        $b \leftarrow$ NoK-Pattern-Matching($s, u, R$);
10:                        if $b$ = TRUE
11:                           then $S \leftarrow S \setminus \{s\}$;
12:         $u \leftarrow$ Following-Sibling($u$);
13:     until $u$ = NIL or $S = \emptyset$
14: if $S \neq \emptyset$
15:     then $R \leftarrow \emptyset$;
16:             return FALSE;
17: return TRUE;
```

The secure NoK pattern matching algorithm is shown in Algorithm 1. The input parameter *proot* is the current node from the NoK pattern tree, and *sroot* is the current document node that is being matched to *proot*. The third parameter R is set to $\emptyset$ initially and will contain a list of data tree nodes (in document order) that match the returning node. To match NoK subtrees, the query processor uses a recursive navigational approach, starting with an initial match from the data for the root of the NoK pattern tree. It then proceeds by recursively matching children of *proot* to children of *sroot*. This top-down recursive pattern matching requires $O(|P| \times |D|)$ time (instead of exponential time) to find *all* matches, where P is the size of the pattern tree and the D is the size of the document [5]. The subroutines FIRST-CHILD and FOLLOWING-SIBLING use the block-oriented physical encoding of the document structure to return the first child of the *sroot* in document-order, or the next sibling of the current node, respectively. The subroutine ACCESS (line 6) checks the accessibility of the child of the current document subtree root to be matched. Since a node's accessibility is checked immediately after it is loaded (by FIRST-CHILD or FOLLOWING-SIBLING), and since its access control code will be found on the same page as the node itself, no additional I/O will be required for node accessibility checks. Note that the pre-condition of Algorithm 1 is the *sroot* nodes be accessible. This means before we use Algorithm 1 to recursively match NoK pattern trees, the root of the NoK data tree should be checked to make sure it is accessible. According to the query evaluation semantics given early in Section 4, we can also skip the recursion on the child if the child is not accessible.

After NoK subtree matches are located, they can be structurally joined based on ancestor-descendant relationships. We have the following theorem:

Theorem 1 *Algorithm ε-NoK, together with any non-secured structural join algorithm, securely evaluates XML twig queries.*

Proof The NoK pattern matching algorithm returns all and only those matched pattern trees[5]. Further, NoK pattern matching together with any (non-secured) structural join algorithm will evaluate any XML twig query [5]. Algorithm ε-NoK is based on the NoK pattern matching algorithm and adds an accessibility check for each potential matching node from the data tree. This ensures that each matching NoK pattern trees reported by ε-NoK consists only of accessible nodes. Unlike the unsecure NoK pattern matching algorithm, ε-NoK terminates its recursion as soon as it encounters a non-accessible node within a potential pattern match in the data tree. Since all data nodes in a pattern match must be accessible, this early-out mechanism will not cause ε-NoK to fail to report any accessible matches. Furthermore, it will *not* cause ε-NoK to miss any accessible, matching NoK tree that is beneath the non-accessible node, since the (secure) NoK matching algorithm will be invoked for each document node that has the same node-tag as the root of the NoK pattern [5]. If there is an accessible matching NoK tree underneath the non-accessible node, it will be checked and returned by another ε-NoK invocation. Therefore ε-NoK retrieves all and only the accessible NoK patterns according to the secure query semantics. Since ε-NoK pattern matching returns only accessible matches, and since the accessibility of nodes outside of these matches is not relevant to the secure query evaluation semantics, the structural-join algorithm does not require any further accessibility checks. Therefore, the ε-NoK algorithm and any non-secured structural join algorithm will securely evaluate XML twig queries. $\square$

4.2 Alternative Access Control Semantics

There is another (more restrictive) semantics for secure query evaluation, which is defined by Gabillon and Bruno [8]. This semantics specifies that a subtree rooted at a non-accessible node can not provide answers, even if it contains accessible nodes. For example, the pattern tree in Figure 2 will not find any matches from the data tree if node `e` is not accessible, even if all of the remaining nodes are accessible.

Secure NoK pattern matching, as implemented in Algorithm 1, is not sufficient for enforcing this more restrictive semantics. In addition to ensuring that each NoK query pattern matches only accessible nodes, we must also ensure that when these matches are structurally joined to produce the final answer, the structural joins do not depend on inaccessible nodes. Continuing with the example from Figure 2, this means that we must ensure that the matches for the two NoK subqueries (rooted at nodes `a` and `e`), do not structurally join through inaccessible nodes, e.g., node `e`.

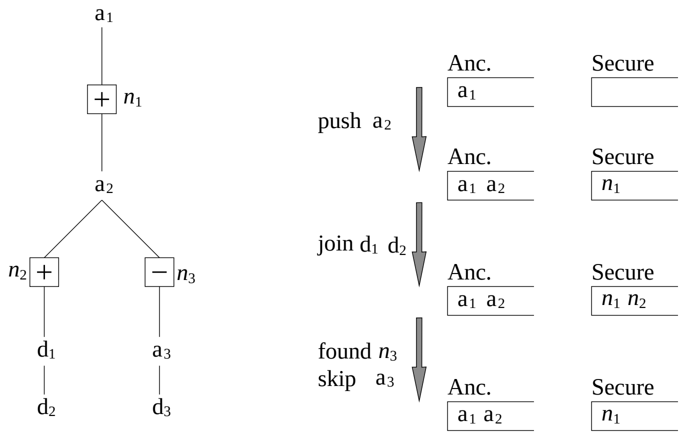


Fig. 4. Anc stack and Secure stack during structural join

We have developed a secure structural join algorithm which, together with the secure NoK pattern matching algorithm from Figure 2, can be used to implement the more restrictive semantics of Gabillon and Bruno if this is desired. The secure structural join algorithm must check the access control lists of all nodes on the path between the NoK subtrees that are being joined. In the NoK physical representation, these nodes are not necessarily clustered on the same physical pages as the NoK subtrees. As a result, secure query evaluation under this more restrictive semantics may be much more expensive than secure evaluation under our original semantics.

We based our secure structural join algorithm on the most efficient structural join paradigm, the *Stack_Tree_Desc* (STD) join algorithm [9]. The algorithm uses a stack (the *Anc* Stack) for storing ancestor nodes, with each node in the stack being a descendant of the node below it. Once a descendant node is found to be a descendant of the top of stack, it is joined with the nodes in the Anc stack all at once. Figure 4 illustrates the structural join of ancestor nodes $\{a_1, a_2, a_3\}$ with descendant nodes $\{d_1, d_2, d_3\}$. Since a_2 is a descendant of a_1, the Anc stack has a_1 below a_2. When d_1 and d_2 are found to be descendants of a_2 (the stack top), they are joined with a_1 as well, avoiding extra Anc-Desc relationship checking with a_1.

For secure structural join, we need to check the nodes on the path between the Anc-Des pairs. There are two accessible nodes n_1, n_2 and one non-accessible node n_3 in the figure. Both d_1 and d_2 have n_1 on the path from a_1, therefore we could check the accessibility of n_1 once for both Anc-Des pairs. On the other hand, there is a non-accessible node (n_3) on the path from a_2 to d_3. That means we could also abort the join between a_1 and d_3, since a_1 is ancestor of a_2 and can not reach d_3 via a path of all accessible nodes. We could also ignore a_3 since it has a non-accessible ancestor (n_3) and will not join with any descendants.

The above optimizations can be accomplished by using an additional stack (the Secure stack) containing the nodes that is already checked for accessibility. The complete secure structural join algorithm (ε-STD, Algorithm 2) consists of a *Stack_Tree_Desc* variant plus a security checking sub-routine (**Check**).

Before we push an ancestor node into the Anc stack, we check the nodes on the path from stack top to the current ancestor node (via sub-routine **Check**). We check the nodes in sequence, and if a node is accessible, we push it onto the Secure stack (Line 5 in **Check**), otherwise we skip the current ancestor node and all its descendants (Line 6 in **Check**). We do the same when a descendant node is joined with the Anc stack. Before we push nodes into the Secure stack, we pop out the top nodes that are not ancestors of the node to be pushed in (Line 2 in **Check**). Figure 4 illustrates the changing content of the Secure stack. The last change occurs when a_3 is to be pushed into the Anc stack and n_3 is to be checked for accessibility. This requires us to pop n_2 from the Secure stack since it is not ancestor of n_3. Then we find that n_3 is non-accessible and skip a_3 and d_3.

We have the following proposition for ε-STD algorithm's I/O cost:

Theorem 2 *If there are N nodes between the ancestors and descendants for structural join, and these N nodes are located on M pages, where $M \leq N$, the*

Algorithm 2. ε-STD Structural Join

```
ε-STD(AList, DList)
      Pre: AList, DList are sorted on document order
 1: Let $a$, $d$ be the first node in AList and DList
 2: while $a$ and $d$ are not null
 3:       do if $top$(Anc) is not ancestor of $a$ or $d$
 4:             then $pop$(Anc);
 5:             else  if $a$ is ancestor of $d$
 6:                      then if CHECK($a$, AList) = TRUE
 7:                              then $push$(Anc, $a$)
 8:                           $a$ = Alist.$next$;
 9:             else  if $a$ is before $d$ in document order
10:                      then $a$ = Alist.$next$
11:             else  if CHECK($d$, DList) = TRUE
12:                      then for each $a1$ in Anc
13:                                do output ($a1$, $d$)
14:                           $d$ = DList.$next$
CHECK(n, List)
1: while $top$(Secure) is not an ancestor of $n$
2:       do $pop$(Secure)
3: for each $p$ on the path from $top$(Secure) to $n$
4:       do if ($p$ is accessible)
5:             then $push$(Secure, $p$)
6:             else  remove $n$'s descendants from List
7:                   return FALSE
8: return TRUE
```

ε-STD algorithm loads each of the M pages at most once (in document order) for all the AD joins.

Proof The algorithm only loads the nodes in one pass, thus the same page will not be loaded twice. □

We also have the following theorem for the correctness of ε-STD algorithm for securely computing structural joins:

Theorem 3 *Algorithm ε-NoK and Algorithm ε-STD together compute the alternative semantics defined in [8].*

Proof The *Stack_Tree_Desc* join algorithm computes all and the only matching Ancestor-Descendants [9]. The ε-STD algorithm adds a **Check** sub-routine before it stores each ancestor/descendant for matching. The **Check** routine relies on the **Secure** stack, which, at all times, contains only accessible nodes. The **Check** sub-routine returns *true* if and only if all of the ancestors of the input node n are accessible. Therefore, both the ancestors (in **AList**) and the descendants (in **DList**) must have only accessible ancestors in order to be matched by the ε-STD algorithm. Hence, the ε-STD algorithm reports all and only those matching Ancestor-Descendants from the *Stack_Tree_Desc* algorithm that have no intervening inaccessible nodes, as required by the alternative secure query evaluation semantics. Further, from the proof of Theorem 1 we know ε-NoK retrieves all and only the accessible NoK patterns. If one node in an accessible NoK pattern tree has accessible ancestors only, every other node in the that NoK tree will also have accessible ancestors only. Therefore, the ε-STD algorithm reports Ancestor-Descendant NoK tree pairs that contains accessible nodes only, and every node in the output has accessible ancestors only. Hence the theorem holds. □

5 Multiple Action Modes

Access control correlation may also exist between different action modes. The correlation may come from the action mode hierarchy [10] where a *write* access right implies a *read* access right from the same subject to the same object. However, we envision that even when there is no such action hierarchy between different action modes, there can be still correlation between different action modes. In fact, our codebook based approach could be similarly applied on multi-action modes regardless of the existence of an action mode hierarchy.

Suppose a system has multiple action modes ($\mathcal{M} > 1$). We have two choices in building codebook entries. First, we can view the $\mathcal{S} \times \mathcal{D} \times \mathcal{M}$ 3-dimensional access cube as a stack of $\mathcal{D}$ independent slabs; each slab is a $\mathcal{S} \times \mathcal{M}$ 2-dimensional matrix, holding the complete access control information for a specific object. Following this perspective, we can store one transition node for each object that has different access control slab than its predecessor, and each transition node points to one "slab" of size $\mathcal{S} \times \mathcal{M}$ in the codebook. Alternatively, we can view the 3-dimensional cube as $\mathcal{D} \times \mathcal{M}$ 1-dimensional vectors; each vector has $\mathcal{S}$ bits, recording the complete access control information for a specific object and action mode. In this case, the entries of codebook are still vectors of size $\mathcal{S}$; but for each transition object (one that does not agree with its predecessor on the accessibility for any user under any action mode), we need to maintain $\mathcal{M}$ pointers, one for each action mode [1]. Suppose T denotes the number of transition nodes (the number of transition nodes are the same for either approaches), R denotes the size of one pointer from a transition node, V_l denotes the number of distinct access control slabs, and V_c denotes the number

[1] We could also keep only the changing pointers for each transition node, but in that case, DOL lookup becomes expensive since we might need to look ahead for several transition nodes

of distinct access control vectors. The space cost of the vector approach and the slab approach can be calculated by formula 1 and 2 respectively.

$$SIZE_{slab} = \underbrace{T \times R}_{\text{on disk}} + \underbrace{V_l \times (|\mathcal{M}| \times |\mathcal{S}|)}_{\text{in memory}} \tag{1}$$

$$SIZE_{vector} = \underbrace{T \times |\mathcal{M}| \times R}_{\text{on disk}} + \underbrace{V_c \times (|\mathcal{S}|)}_{\text{in memory}} \tag{2}$$

Both approaches have advantages and disadvantages. Since T grows with $\mathcal{D}$ and is usually much greater than $\mathcal{S}$ and $\mathcal{M}$, the overall space cost is likely to be dominated by the cost of the on-disk part. Comparing with the vector approach, the slab approach has less overall space cost, as it has a smaller on-disk part, but it requires more memory, depending on the value of V_l and V_c. We analyze these values in Section 6.3.

6 Performance Evaluation

We evaluated the DOL technique using both synthetic and real access control data. We generated single-user, single-access-mode synthetic access controls on XML data from XMark benchmarks [11] by randomly choosing some nodes from the document as *seeds*, and then labeling these seeds as accessible or non-accessible. We simulate horizontal structural locality by selecting from the seeds' direct siblings and set them with the same accessibility of the seeds, provided that the siblings are not themselves seeds. Then, we simulate vertical structural locality by propagating accessibilities of labeled nodes to their descendants using the Most-Specific-Override policy [1], i.e., a node inherits its accessibility from its closest labeled ancestor. We always choose the document root as seed to ensure that all nodes are labeled. Access control generation is controlled by two parameters. The *propagation ratio* determines percentage of nodes that are seeds while the *accessibility ratio* determines the percentage of seeds that are accessible.

In addition, we used two sets of real multi-user access control data. The first data set describes the access control information from a production instance of OpenText LiveLink [2], which provides web-based collaboration and knowledge management services in a corporate intranet. The LiveLink system has a total of 8639 subjects (among which there are 1568 leaf users), around 370,000 data items in a tree-structure with an average depth of 7.9 and a maximum depth of 19, and ten action modes. The second data set consists of the access control data from a multiuser Unix file system at the University of Waterloo. This

[2] LiveLink is a trademark of OpenText Corporation.

system has 182 users and 65 user groups, more than 1.3 million files/directories, and three action modes (read, write, execute). Although neither of these systems stores real XML data, both provide tree-structured data models and instance-level access controls. In the absence of real access controlled XML data (to the best of our knowledge), we treat these systems as surrogates for real multi-user access controlled XML databases for the purposes of our experiments. In order to map our unordered tree data to ordered XML nodes, we pick one arbitrary ordering between sibling nodes in our test data: files under the same directory are ordered by their names, and sibling LiveLink data items are ordered by their appearance in the LiveLink system catalog. These two ordering choices are orthogonal to access controls, but the structural locality in the access controlled data still holds, as we shall see from the experiments.

6.1 Space Efficiency in Single-User Environments

We first evaluate DOL for a single subject. Since CAM [4] is the state of the art compact labeling for single subject access controls, we compare DOL with CAM. We first use an XMark document of about $17,000$ nodes with synthetic access controls produced by different accessibility and propagation ratios. Our metric is the ratio of the number of CAM nodes to the number of DOL transition nodes (the codebook size is negligible for one subject). Thus, values less than 1.0 favor CAM and those greater than 1.0 favor DOL.

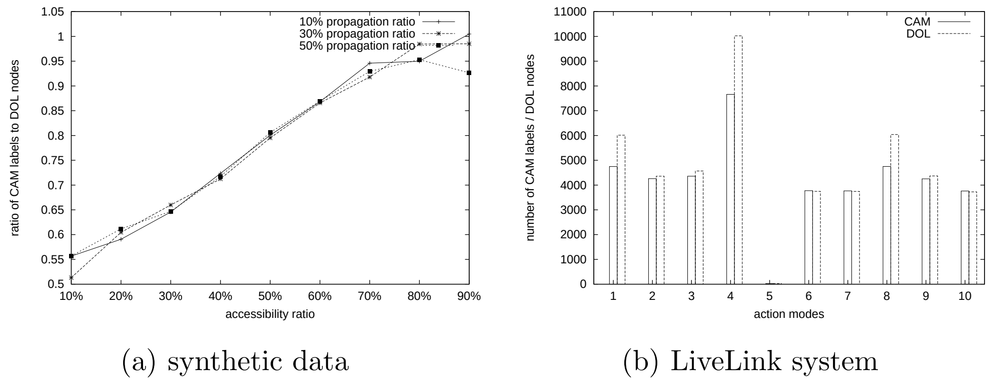
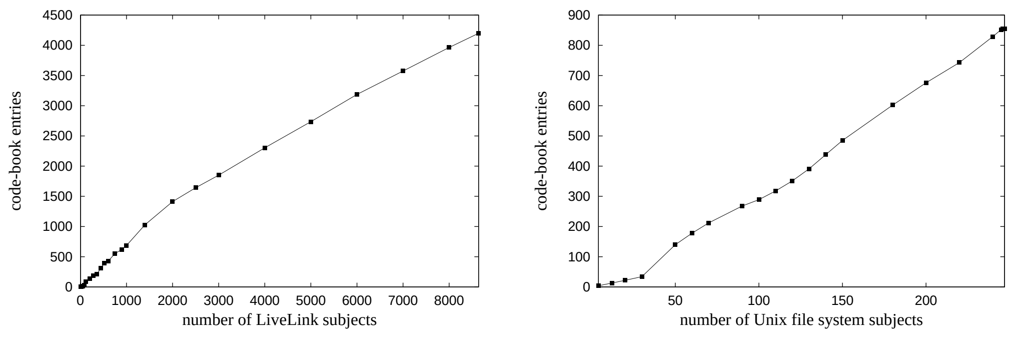


Fig. 5. CAM labels and DOL transition nodes for single subject

Figure 5(a) shows the comparisons as the accessibility ratio varies from 10% to 90%. We tried three propagation ratios with these different accessibility and the results are similar. When the accessibility ratio is low (few nodes are accessible), the number of CAM nodes is around 53% of the number of DOL transition nodes. As accessibility goes up, this difference becomes smaller.

We also compared single user CAM and DOL using the LiveLink data. The LiveLink system supports ten different action modes. For each of the ten action



(a) LiveLink (b) Unix file system

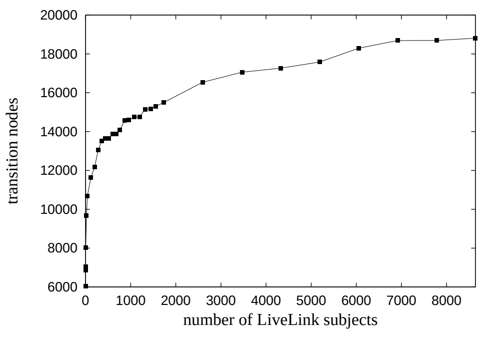
Fig. 6. Codebook entries for multi-subjects

modes we sample a number of users and built CAM and DOL for each single user. The ratio of the number of DOL labels to the number of CAM nodes for an average user is shown in Figure 5(b), with the ten action modes shown on the horizontal axis. In the worst cases, DOL had 20-25% more nodes than CAM. In other cases, the two schemes performed about the same.

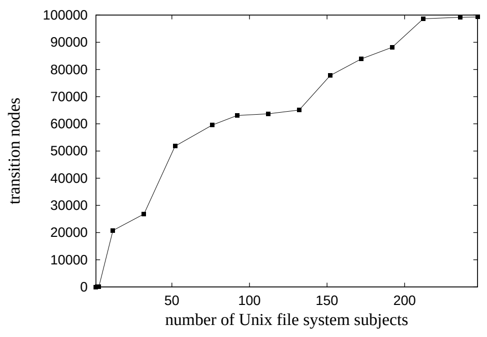
Note that our performance metric implicitly favors CAM because it assumes that CAM nodes and DOL transition nodes are the same size. In practice, however, the DOL nodes are likely to be much smaller. This is because CAM stores the access rights separately from the data. As a result, each CAM node must include a reference to a document node and pointers to the node's children in the CAM, in addition to the access control information itself. In contrast, DOL, which piggybacks access control information into the document encoding, stores only an access control code per transition node. Thus, although CAM may have fewer nodes than DOL, the total space required for CAM may be greater.

6.2 Space Efficiency in Multi-User Environments

We used our two real data sets to evaluate the space efficiency of DOL in multi-user environments. We only present the results under "SEE" action mode of LiveLink and the "READ" action mode of the Unix file system, since other action modes of both data show similar trends. To get a sense of how the codebook size might vary as a function of the number of subjects, we selected a number of subjects randomly and computed DOL codebooks for the selected subjects only. In Figures 6(a) and 6(b) we have plotted the number of codebook entries as a function of the cardinalities of these subsets. (Each set of users is a randomly chosen subset of the next larger set). If subjects' access controls were uncorrelated, we would expect to see exponential growth in the number of codebook entries as the number of subjects increased. However, our results show that the growth is much slower in practice. With



(a) LiveLink



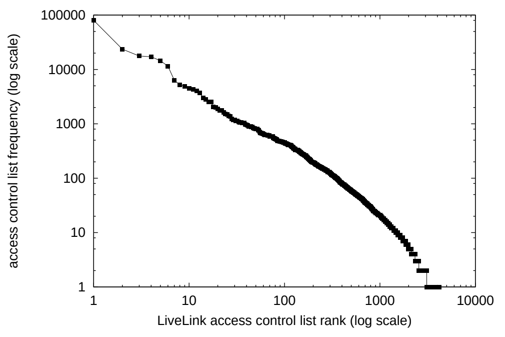
(b) Unix file system

Fig. 7. Transition nodes for multi-subjects

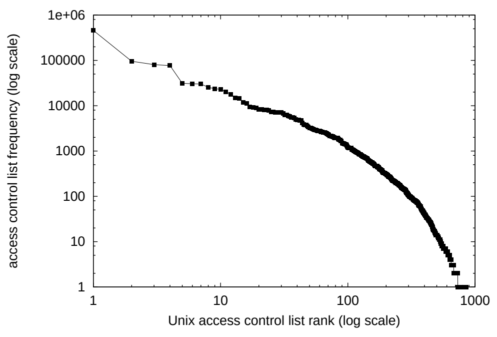
all 8639 subjects, the LiveLink system required around 4000 codebook entries. At approximately 1100 bytes per codebook entry (one bit per subject), the complete LiveLink codebook would occupy only about 4.4MB of memory. The Unix system required about 855 codebook entries for 247 subjects, with an overall size of only 25KB.

The other major storage concern is the number of DOL transition nodes. Figures 7(a), 7(b) show the numbers of transition nodes required for the LiveLink and Unix systems as the number of subjects increases (using the same methodology that was used for Figures 6(a) and 6(b)). Figure 7(a) shows a sub-linear growth of transition nodes. For over 8000 subjects, the number of transition nodes is only about 4 times larger than the number for a single subject. For Unix file system, we see a similar situation in Figure 7(b), in which the number of transition nodes of 247 subjects is only twice as many as the number for 50 subjects. Recall there are about $370,000$ objects in the LiveLink system, and the number of files (directories) in the Unix system is about 1.3 million. Thus, the density of transition nodes is less than 10% for both systems (for all the subjects). These results indicate that the access rights for different subjects are highly correlated in real world.

To compare the overall storage cost between DOL and CAM, we first look at a single subject in LiveLink (under action mode 1): DOL needs about 6000 transition nodes while CAM needs 4500 labels. However, for all 8639 subjects in the same system under the same action mode, DOL needs 18800 transition nodes while CAM needs 8639×4500 labels, a difference of three orders of magnitude. Assuming each DOL transition node requires a 2 byte access control code (for the 4000 codebook entries), and each CAM label takes 2 bits for its accessibility encoding, and (unrealistically) only 1 byte for node pointers, the DOL's total space requirement will be a 4MB codebook plus 40KB of embedded transition nodes, while CAM's will be 46.6MB. The Unix file system's situation is similar. Clearly, correlation among the subjects contributes substantially to compression effectiveness.



(a) LiveLink



(b) Unix file system

Fig. 8. Rank Frequency of Access Control Lists for Unix and LiveLink

A final observation, from both systems, is that the frequency of the access control lists in the codebook is highly biased. Figure 8 shows the rank/frequency plots of the access control list in the Unix and LiveLink Systems. Here we choose "Read" action mode for Unix and action mode 1 for LiveLink since they are representative of the remaining action modes. In the Unix system, about ten access control lists account for more than 80% of all the files' access controls. A single access control list (granting access to everyone) accounts for more than half of the access controls on all the files. The LiveLink system showed similar behavior, with ten access control lists account for about half of the all objects' access controls.

This frequency skew, together with the locality of access rights, helps to explain the space efficiency of the DOL approach. It also suggests that, in small-memory environments, good performance can be achieved without keeping the entire codebook in memory. If only the most frequently used access control lists are kept in memory, it should be possible to answer most access control lookups efficiently. In our experiments with LiveLink data, if we load in memory 1000 most frequently entries out of all the 4198 entries of the codebook, we can answer more than 95% of the accessibility queries without loading any codebook entries from the disks.

6.3 Multiple Action Modes: Vectors Versus Slabs

Recall that the total space cost of the codebook scheme includes two parts: the on-disk part occupied by the access control codes and the in-memory part occupied by the access control codebook. When there are multiple action modes, the DOL can be implemented either as a codebook with access control vectors for all action modes (where each transition node will have multiple pointers to different codebook entries for each action mode) or as an array of access control slabs (where each transition node points to one slab entry for all action modes), as described in Section 5.

Action Mode	***read***	***write***	***execute***	***all***	***slab***
CodeBook#	855	883	857	995	1608

Fig. 9. CodeBook Entries for Vector/Slab-based Approaches (Unix System)

We first tested the vector approach on the Unix file system. Figure 9 illustrates the number of distinct access control vectors (i.e., the number of codebook entries) for read, write, and execute modes, together with the number of distinct codebook entries across all three action modes (the ***all*** column). We can see the total number of distinct access control vectors of all three action modes is much fewer than the sum of the number of distinct vectors under the individual action modes. This tells us that the access control vectors of different action modes are similar.

Next, we tested the slab approach by concatenating the access control vectors into slabs. If the access controls under different action modes are not correlated, we would have seen all combinations of concatenated access control vectors in the slabs, making the number of distinct slabs grow exponentially in the number of action modes. However, as we observed from our real data, the number of distinct slabs is only about four times the number of distinct access control vectors (see column ***slab*** in Figure 9). Since the Unix system has 247 subjects, each access control vector would require 247 bits, leading to a total codebook size of approximately 31 KB under the vector-based approach. Since there are three access modes, each slab would require 741 (247×3) bits, giving a total codebook size of 149 KB under the slab-based approach. Therefore, the codebook of the slab approach is only around five times the size of the codebook under vector approach.

Action Mode	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***	***9***	***10***	***all***	***slab***
CodeBook#	4235	3926	5591	4273	4144	4362	4684	22	4325	4198	9863	10976

Fig. 10. CodeBook Entries for Vector/Slab-based Approaches (LiveLink System)

We noticed an even stronger correlation between action modes in the LiveLink data (Figure 10), where the number of distinct access control vectors for all action modes is less than 25% of the total of all distinct access control vectors under each action mode, and the number of distinct slabs is almost the same as number of distinct access control vectors for all action modes. Since the LiveLink system has around 8600 subjects, each access control vector would require 1075 bits, thus the the vector-based codebook has size of 10.6MB (1075×9863). The slab-based codebook has each entry as ten vectors concatenated, thus its size is approximately 118MB ($10976 \times 1075 \times 10$). Therefore, the slab-based codebook is only about ten times the size of the combined vector-based codebook.

Aside from codebook entries, we can be sure that the density of transition

Q1	/site/regions/africa/item[location][name][quantity]
Q2	/site/categories/category[name]/description/text/bold
Q3	/site/categories/category/description/text/bold
Q4	//parlist//parlist
Q5	//listitem//keyword
Q6	//item//emph

Fig. 11. Sample Queries

nodes for the slab approach is the same as the vector approach, since either approach would need a transition node whenever the accessibility changes for one user under one action mode. However, under the vector approach, each transition node needs $\mathcal{M}$ pointers to the entries of the codebook for the $\mathcal{M}$ action modes, whereas the slab approach only requires one. Considering that the number of transition nodes grows with the number of objects in the system (recall from Section 6.2 that transition nodes are around 10% of the total objects for all users for both real data sets), the size of transition nodes would dominate the size of codebook by several orders of magnitude. Thus we could save considerable space by using the slab based approach at the cost of a (moderately) larger codebook in main memory.

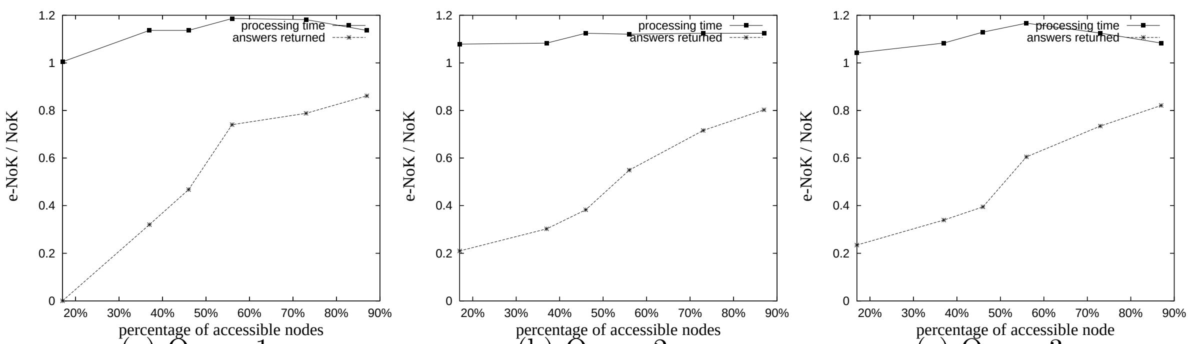
6.4 Query Evaluation

We compared the performance of secure NoK query evaluation with non-secured NoK query evaluator to measure the overhead of fine-grained access control. We implemented both ε-NoK, ε-STD, and the non-secure versions of the NoK and STD algorithms using Java 1.5. All of the experiments were conducted using a PC with a Pentium III 997MHz CPU, 512MB RAM, and 40GB hard disk running Windows XP.

Our test database is a 50MB XMark instance (832911 element nodes) with synthetic access controls. The data are stored on disk with each page at 4K bytes. The benchmark queries are shown in Figure 11. The top three queries represent three classes of NoK pattern trees: those with branches at the end (Q1), in the middle (Q2), or a single path (Q3). The bottom three queries are for ancestor-descendant structural joins and represent those having descendants located close to the ancestors(Q4), far from the ancestors (Q6), or at a medium distance (Q5).

Figures 12(a),12(b) and 12(c) show the performance of the ε-NoK algorithm. The two lines in each figure depict the ratio of processing time and answers returned between the ε-NoK and non-secure NoK algorithms. The processing time of the ε-NoK algorithm is at most 20% greater than the processing time of the non-secure NoK algorithm, and does *not* depend on the accessibility ratio. This is still true when majority of the document is accessible (thus most answers of the original NoK algorithm are returned, and nodes in these answers are all checked). The reason is that accessibility checking does not require extra I/O for the ε-NoK algorithm.

Figures 13(a),13(b) and 13(c) show the performance of the secure ε-STD algorithm. Figure 13(a) shows the ratio of processing time between the ε-STD and non-secure STD algorithms for Queries 4, 5, and 6. As we can see, ε-STD is much more expensive. However, when the number of answers decreases as node accessibility decreases (in Figure 13(b)), the corresponding processing time also goes down. Also, we note that processing time decreases faster for the queries with longer ancestor-to-descendant paths. This shows that ε-STD is more sensitive to node accessibility and query topology, and its performance is better when majority of the document is non-accessible. The reason is that the ε-STD algorithm can optimize by skipping the accessibility checks for some nodes on the ancestor-to-descendant path. This is verified in Figure 13(c), where the y-axis shows the percentage of nodes (between each ancestor-descendant pairs) checked with accessibility by ε-STD Algorithm. As we can see, the number of accessibility checks goes down (at different speed for three queries) as node accessibility goes down.

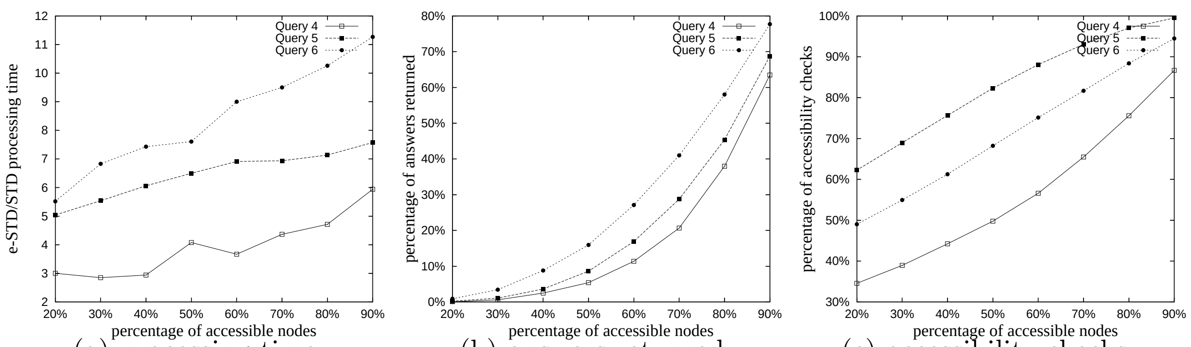


(a) Query 1 (b) Query 2 (c) Query 3

Fig. 12. Performance between ε-NoK and NoK as a function of node accessibility

7 Related Work

Jajodia et al. [1] and Bertino et al. [2] define models that are capable of describing a wide spectrum of access control policies such as positive and negative authorization, propagation policies, conflict resolution, and closed versus open world assumptions. The IBM XACL project [12] proposed an XML access control language for authorizing access (read, write, create, delete) to fine-grained



(a) processing time (b) answers returned (c) accessibility checks

Fig. 13. Performance between ε-STD and STD as a function of node accessibility

XML data. It is capable of defining propagations, conflict resolving and provisional authorization. Bertino et al. [13] proposed a model that is capable of describing various access control policies. Their model also considers "push" queries, which is for massive distribution of data to subscribers. Damiani et al. [14] proposed access control query modeling specific for XML documents to facilitate secure information flow for the Web. A similar framework is implemented in the Author-X project [15].

Gabillon and Bruno [8] define view-based semantics for secure query evaluation. They define a secured view for each user group, and queries are applied against the secure views. However, their approach prunes all subtrees with a non-accessible root, regardless whether there are accessible nodes in that subtree or not. Cho et al. [7] define a more relaxed pattern-matching based semantics. This semantics allows answers to come from a subtree whose root is non-accessible. They use schema information to rewrite queries to optimize query evaluation time.

Stoica et al. [16] use secured views and secure data schema (both generated from original data schema and access control policies) for answering XML queries. Similarly, Fan et al. [3] use secured views generated from DTD schemas and access control rules for secure query evaluation. The access control rules in these two approaches are based on the data schema, which must exist. However, instance-based access control is necessary when there is no schema, or when the desired access controls cannot be described by schema level specifications. Lee et al. [17] propose a secure XML evaluation framework in which access controls are specified on the XML model but enforced in an underlying relational database. However, the XML data instance needs to be first shredded into the relational model. There is also work [18] [19] on efficient dissemination of sensitive XML data using pattern matching. The DOL approach can be similarly used for dissemination of XML data to multiple users. The difference is that DOL supports fine-grained access controls at the instance level.

Yu et al. [4] developed the CAM representation for fine-grained access controls on XML data. A separate CAM structure is required for each user under each action mode, so CAM does not exploit the commonality among multiple users to achieve better compression ratio. Recently, Jiang and Fu proposed Integrated Compressed Accessibility Maps (ICAM) as an improvement over the original CAM approach.[10] Like DOL, this approach exploits correlations between different action modes and compresses the CAMs for different action modes into one integrated structure. However, the approach also requires that the action modes follow a strict *operational hierarchy*, i.e., a subject with write access rights on an object always has read access on that same object. This is not always desireable. e.g., a user may be able to write to a file while not being able to read it, as in the Mandatory Access Control model [20]. The DOL, in contrast, neither requires nor exploits an operational hierarchy.

8 Conclusion

We have presented a compact XML access control labeling scheme called DOL that supports efficient secure query evaluation. DOL exploits both access control structural locality within the XML data and correlations between users' access rights. Physically, access controls are embedded into the representation of the document structure used for NoK query evaluation. Our experiments demonstrate DOL's storage efficiency in multi-user environments as well as DOL's ability to support secure query evaluation efficiently.

Acknowledgements Thanks to Frank Tompa and the anonymous reviewers for their feedback on this work. Thanks also to Gary Promhouse of OpenText for his feedback, and for providing the LiveLink access control data. This work was supported by Communications and Information Technology Ontario (CITO), the Natural Sciences and Engineering Research Council of Canada (NSERC), and by the OpenText Corporation.

References

[1] S. Jajodia, P. Samarati, M. L. Sapino, V. S. Subrahmanian, Flexible Support for Multiple Access Control Policies, ACM Trans. Database Sys. 26 (2) (2001) 214–260.

[2] E. Bertino, B. Catania, E. Ferrari, P. Perlasca, A Logical Framework for Reasoning about Access Control Models, ACM Transactions on Information and System Security 6 (1) (2003) 71–127.

[3] W. Fan, C.-Y. Chan, M. Garofalakis, Secure XML Querying with Security Views, in: Proc. ACM SIGMOD Int. Conf. on Management of Data, 2004, pp. 587–598.

[4] T. Yu, D. Srivastava, L. V. S. Lakshmanan, H. V. Jagadish, A Compressed Accessibility Map for XML, ACM Trans. Database Syst. 29 (2) (2004) 363–402.

[5] N. Zhang, V. Kacholia, M. T. Özsu, A Succinct Physical Storage Scheme for Efficient Evaluation of Path Queries in XML, in: Proc. 20th Int. Conf. on Data Engineering, 2004, pp. 54–65.

[6] B. Lampson, Protection, in: ACM Operating Systems Review, 1974, pp. 8–24.

[7] S. Cho, S. Amer-Yahia, L. V. S. Lakshmanan, D. Srivastava, Optimizing the Secure Evaluation of Twig Queries, in: Proc. 28th Int. Conf. on Very Large Data Bases, 2002, pp. 490–501.

[8] A. Gabillon, E. Bruno, Regulating Access to XML Documents, in: Proc. 15th Int. Conf. on Database and Application Security, Kluwer Academic Publishers, 2002, pp. 299–314.

[9] S. Al-Khalifa, H. V. Jagadish, N. Koudas, J. M. Patel, D. Srivastava, Y. Wu, Structural Joins: A Primitive for Efficient XML Query Pattern Matching, in: Proc. 18th Int. Conf. on Data Engineering, 2002.

[10] M. Jiang, A.W. Fu, Integration and Efficient Lookup of Compressed XML Accessibility Maps, IEEE Trans. Knowledge and Data Eng. 17 (7) (2005) 939–953.

[11] The XML benchmark project, available at http://www.xml-benchmark.org.

[12] M. Kudo, S. Hada, XML Document Security based on Provisional Authorization, in: 7th ACM Conference on Computer and Communication Security, 2000, pp. 87–96.

[13] E. Bertino, E. Ferrari, Secure and Selective Dissemination of XML Documents, ACM Transactions on Information and System Security 5 (3) (2002) 290–331.

[14] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, P. Samarati, A Fine-grained Access Control System for XML Documents, ACM Transactions on Information and System Security 5 (2) (2002) 169–202.

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

[16] A. G. Stoica, C. Farkas, Secure XML Views, in: Proc. 15th Int. Conf. on Database and Application Security, 2002, pp. 133–146.

[17] D. Lee, W.-C. Lee, P. Liu, Supporting XML Security Models Using Relational Databases: A Vision, in: Proc. 1st Int. XML Database Symposium, 2003, pp. 267–281.

[18] M. Altinel, M. J. Franklin, Efficient Filtering of XML Documents for Selective Dissemination of Information, in: Proc. 26th Int. Conf. on Very Large Data Bases, Morgan Kaufmann Publishers Inc., 2000, pp. 53–64.

[19] Y. Diao, R. T. Peter M. Fischer, Michael J. Franklin, YFilter: Efficient and Scalable Filtering of XML Documents, in: Proc. 18th Int. Conf. on Data Engineering, 2002, pp. 341–353.

[20] R. S. Sandhu, P. Samarati, Access Control: Principles and Practice, IEEE Communications Magazine 32 (9) (1994) 40–48.