QFilter: Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting

Bo Luo, Dongwon Lee, Wang-Chien Lee, Peng Liu
The Pennsylvania State University, University Park, PA 16802, USA
bluo@ist.psu.edu, dongwon@psu.edu, wlee@cse.psu.edu, pliu@ist.psu.edu

ABSTRACT
At present, most of the state-of-the-art solutions for XML access controls are either (1) document-level access control techniques that are too limited to support fine-grained security enforcement; (2) view-based approaches that are often expensive to create and maintain; or (3) impractical proposals that require substantial security-related support from underlying XML databases. In this paper, we take a different approach that assumes no security support from underlying XML databases and examine three alternative fine-grained XML access control solutions, namely *primitive*, *pre-processing* and *post-processing* approaches. In particular, we advocate a pre-processing method called *QFilter* that uses Non-deterministic Finite Automata (NFA) to rewrite user's query such that any parts violating access control rules are pruned. We show the construction and execution of a QFilter and demonstrate its superiority to other competing methods.

Categories and Subject Descriptors
H.2.7 [**Database Management**]: Database Administration – *security, integrity, and protection*

Keywords
XML security, Data security and privacy, Query rewriting

1. INTRODUCTION
The eXtensible Markup Language (XML) [2] has emerged as the de facto standard for storing and exchanging information in the Internet Age. As the distribution and sharing of information over the World Wide Web becomes increasingly important, the needs for efficient yet secure access of XML data naturally arise. It is necessary to tailor information in XML documents for various user and application requirements, preserving confidentiality and efficiency at the same time. Thus, it is critical to specify and enforce access control over XML data to ensure that only authorized users have an access to the portion of the data they are allowed to. An intuitive approach, employed by many current web systems (e.g. Apache), is to allow specification and control of data access at the document (or file) level. However, this simple solution is not sufficient for today's XML applications, where data access needs to be performed at a finer granularity (such as data content at the element and attribute level).

To remedy these shortcomings, various proposals in support of fine-grained XML access controls have recently appeared. However, most of them are either *view-based* [1,4,5] or require significant security-related support from the underlying XML database [3, 13]. An inherited issue of using views for data access control is that the specification and maintenance of views are labour-intensive and time/resource-consuming. It is not scalable for administrators to (manually) create views on fine-grained data for a large number of users [21]. On the other hand, requiring security-related support from underlying databases, one may have difficulty to implement XML access controls using today's off-the-shelf XML products (to our best knowledge, none of the recent developments in [1,4,5,3,13] are adopted to XML database products). Additionally, as RDBMSs have been frequently used to manage XML data in the real world, they may not be able to handle fine-grained XML access control policies [10]. The goal of this study is to provide pragmatic solutions for implementing fine-grained XML access controls that not only are ***view-independent*** but also require ***no-security support*** from underlying databases.

In this paper, we examine three different approaches, namely, *primitive*, *pre-processing* and *post-processing,* to achieve our goal. Especially, we advocate a practical and scalable solution, called *Query Filter (QFilter)*. As an XML access control pre-processor external to the database engine, the QFilter checks XPath queries against access control policies. Instead of simply filtering out queries that do not satisfy access control policies and deferring the rest of queries to XML query engines for further checking and processing (as [13] does), QFilter takes extra steps to rewrite queries in combination of related access control policies before passing the revised queries to underlying XML query engine for processing. As we will show later, QFilter not only achieves security for (almost) free, but also enjoys a faster query evaluation time through query re-writing.

Our **contributions** are three-fold: (1) we identify the need and potential of non-view based XML access controls, and examine three different approaches to implement XML access control enforcement mechanisms. (2) We present the design and implementation of QFilter using Non-deterministic Finite Automata (NFA); (3) we conduct an extensive performance evaluation on the QFilter and other approaches. Experimental result shows that QFilter is very efficient in terms of query execution time and is scalable to the number of access control rules specified in the system.

2. RELATED WORK
Models. Several XML access control models are recently proposed. In [16], authorizations are specified on portions of HTML documents, but no semantic context similar to that provided by XML can be supported. In [5], a specific authorization sheet is associated with each XML document/DTD expressing the

CIKM'04, November 8–13, 2004, Washington, DC, USA.

authorizations on the document. In [4], the model proposed in [5] is extended by enriching the authorization types supported by the model, providing a complete description of the specification and enforcement mechanism. Among comparable proposals, in [1], an access control environment for XML documents and some techniques to deal with authorization priorities and conflict resolution issues are proposed. Moreover, general purpose access control policy languages are developed in such efforts as XACL by IBM [9] and XACML by OASIS [8]. Finally, the use of authorization priorities with propagation and overriding, which is an important aspect of XML access control, may recall approaches in the context of object-oriented databases, like [7] and [15]. Although our proposal is based on existing XML authorization models such as [4], ours is not tightly-coupled to one model, and thus can be easily applied to other models.

Enforcement Mechanisms. Most of the existing XML access control methods are either *view-based* or relying on the XML engine to enforce access control at the node-level of XML trees. The idea of view-based enforcement [4, 5] is to create and maintain a separate view for each user who is authorized to access a specific portion of an XML document. The view contains exactly the set of data nodes that the user is authorized to access. After views are constructed, during run time, users can simply run their queries against the views without worrying about security enforcements. Although views can be prepared offline, in general, view-based enforcement schemes suffer from high maintenance and storage costs, especially for a large number of roles [21]: (1) a virtual or physical view is created and stored for each role; (2) whenever a user prompts *update* operation on the data, all views that contain the corresponding data needs to be synchronized. To tackle this problem, [20] proposes a method using compressed XML views to support access controls. However, view-independent enforcement mechanisms are often more desirable. [3] Addresses the issue of secure XML query evaluation by avoiding unnecessary security checks. Our work is complementary to [3].

To our best knowledge, [13] is the only work comparable to our QFilter approach. [13] performs a *static analysis* that simply classifies a XML query to be either "entirely" authorized or "entirely" prohibited before submitting it to an XML engine. For the "partially" authorized XML queries, [13] relies on the XML engine to filter out the data nodes that users do not have authorizations to access. Our QFilter removes this problem by carefully filtering out those conflicting portions from the input query (by re-writing) so that any off-the-shelf XML databases can be used. In addition, the QFilter has a much better performance than [13] (see Section 5). Finally, compared with various researches on the equivalence/containment/re-writing of XML queries [11, 12], our approach is NFA-based and security-driven.

[22], independently developed, bears some similarity to our QFilter approach: [22] uses NFA to process streaming XML data for access control. Each NFA captures Q ∩ ACR, while QFilter's NFA only captures ACR. Furthermore, in [22] the input of an NFA is XML streaming data and the output is also XML data, while in our approach both the input and output of a NFA is an XML query.

3. XML ACCESS CONTROL MECHANISMS

In this paper, we adopt an XML access control model similar to [4] and incorporate role-based access control (RBAC) [17] to make ours more pragmatic. In our model, users are assigned with *roles* and thus can exercise certain access rights characterized by their roles. An XML document can be represented as a hierarchy of nested nodes (i.e., elements and attributes) so that fine-grained access controls at node level are established. XPath is used for specification of queries as well as identification of nodes. Our node-level authorization is specified via 5-tuple *access control rules* (ACR): $ACR = \{subject, object, action, sign, type\}$, where (1) *subject* is to whom an authorization is granted (i.e., role); (2) *object* is part of an XML data specified by an XPath expression; (3) *action* consists of read, write, and update; (4) $sign \in \{+, -\}$ refers to either access "granted" or "denied", respectively; and (5) $type \in \{LC, RC\}$ refers to either *local check* (i.e., authorization is applied to nodes in context only) or *recursive check* (i.e., authorization is applied to current nodes and propagated to all their descendants), respectively. In general, all nodes whose authorizations are not specified, either explicitly (via LC rules) or implicitly (via RC rules), are considered to be "access denied". It is possible for a node to have more than one relevant rules. If conflict occurs between + and - rules, - rule takes precedence.[1]

An XPath expression of ACR returns a set of nodes as answers, where the nodes are said *projection nodes*. For instance, the projection node of //dept//budget[@type='public'] is "budget" since the expression returns <budget> as answers, not <dept> nor @type. Furthermore, we differentiate two different notions of answer models: (1) "*Answer-as-nodes*": answers are only projection nodes themselves; and (2) "*Answer-as-subtrees*": answers are projection nodes and their descendents. The former can be viewed as an intermediate answer model of the latter. That is, when a query "/a/b" is evaluated, typical XML query processors quickly find the node IDs matching "/a/b" pattern, then return actual <b> nodes and their subtrees as the final answer to users. Since the answer-as-subtrees model may contain unauthorized data in their subtrees, it is more complex to deal with than the answer-as-nodes model (details are in section 3.1). Therefore, in this paper, we primarily focus on the answer-as-nodes model. If it is always assured that the returned answers do not contain any data that violate access control policies, then they are called a *safe* answers, and otherwise *un-safe* answers. Likewise, if a query is assured to retrieve only safe answers, it is called a *safe* query, and otherwise an *un-safe* query. Table below summarizes the notations that we use through the remainder of the paper.

Term	Meaning
Q	User's input query in XPath expression
Q'	Re-written query from Q
D	XML data
SD / UD	Safe / un-safe XML data
R	A 5-tuple access control rule
R^+ / R^-	R that has sign + / −, respectively
ACR	$[R_1, \ldots, R_n]$, a list of access control rules
ACR^+ / ACR^-	All R^+ / R^- of ACR

Our Goal: to devise practical and scalable XML access control mechanisms without using any security features of underlying

[1] Note that our conflict resolution is different from the one based on the nearest ancestor (e.g., [3]). For instance, if - rule limits the access of "/a/b" nodes (and descendents), but + rule grants the access of "/a/b/c" nodes, then in our model, users cannot access any nodes under "/a/b" since – rule takes precedence. However, in the nearest ancestor based resolution scheme, the conflict at the /a/b/c level is resolved through its nearest ancestor, "/a/b/c" itself, whose access is granted by + rule, and thus the access is granted. We plan to investigate the issue of handling the nearest ancestor conflict resolution scheme in future.

DBMS. *Given a list of access control rules, ACR, and a user query, Q, such an XML access control mechanism answers Q by returning only safe data that do not violate ACR.*

We consider the following three different approaches**:**

1. **Primitive**: In this approach, *ACR* is somehow "merged" to query *Q* to yield a new query *Q' = Q ∩ ACR*, to be submitted to a DBMS. Then, only safe answers that satisfy both the constraints in *Q* as well as in *ACR* are returned.
2. **Post-processing**: *Q* is processed by a DBMS as a regular query to produce (unsafe) answers. Then, this intermediate answers go through post-filtering process to prune out those data that violate the *ACR*.
3. **Pre-processing**: Some parts of the *Q* that have conflicts with ACR are pre-pruned to yield *Q'*. Then *Q'* is processed by a DBMS as usual to return only safe answers.

Figure 1 illustrates the current practice of XML query processing (i.e., without access control) and the three approaches described above. In the figure, Galax [19] serves as the underlying database. QFilter and AFilter are used for pre-processing of queries and pos-processing of answers (details to be presented later). In this paper, we advocate the QFilter approach. Thus, in the following, we first introduce the primitive and post-processing approaches and go into detail of the pre-processing approach in the next section.

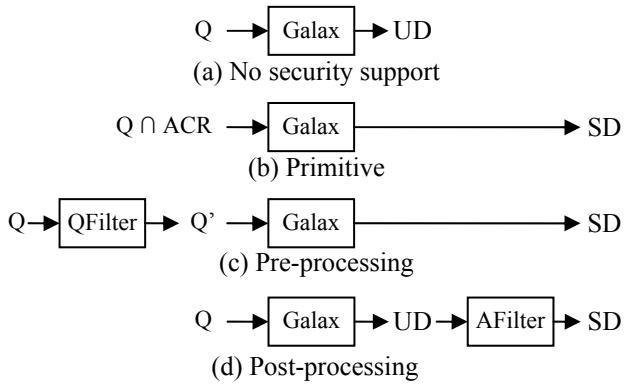


Figure 1: Ways to support XML access control without using security features of DBMS

3.1 Primitive Approach

The idea of the primitive approach is to view both user's query and security policies written in ACR as two constraints to satisfy. Therefore, security enforcement is assured by somehow "merging" two constraints to form tighter constraints. For instance, a manager "John" is to access 'HR' dept's budget with the following query, Q:/dept[name='HR']//budget, and ACR has the following five rules about the 'manager' role (i.e., three LC and two RC rules):

R_1:(manager, /dept//salary, read, +, LC)
R_2:(manager, /dept[year=2004]//budget, read, +, LC)
R_3:(manager, /dept[year=2003]//budget, read, −, LC)
R_4:(manager, /dept/south, read, +, RC)
R_5:(manager, /dept/north, read, −, RC)
R_6:(manager, /dept[name='HR']//budget/secret, read, −, RC)

The meta-semantics of Q and a rule R with + sign is that users are allowed to access the regions scoped by "Q INTERSECT R". Conversely, that of Q and a rule R with – sign is "Q EXCEPT R." Note first that R_1, R_4, R_5 have incompatible projection nodes from Q. Q looks for <budget> as answers while, for instance, R_1 returns <salary> as answers. Therefore, any traditional set operators between two answers would not make sense since it is analogous to comparing apples and oranges. Therefore, when a rule has the LC type and has incompatible projection nodes in its XPath expression from Q, the rule can be safely ignored. However, when the rule has the RC type, the rule cannot be ignored. Consider R_4 who has the same + sign as Q, but whose expression /dept/south has incompatible projection nodes from Q (i.e., south vs. budget). If there happens to be a descendent with a path /dept/south/1/2/budget, then "John" should get a grant to access this budget since R_4 indicates /dept/south and all its descendents should be readable by managers. Therefore, when one merges Q and R_4, one has to use "Q INTERSECT R_4//budget" to ensure domain compatibility of intersection operator. Next, consider rules whose XPath expressions have the compatible projection nodes with Q. Both R_2 and R_3 have + and – signs, respectively. Therefore, "John" can read budget scoped by R_2, but not the budget scoped by R_3: Q' = Q INTERSECT R_2 EXCEPT R_3. Then, the final safe query Q' can be: Q' = Q INTERSECT (R_2 UNION R_4//budget) EXCEPT (R_3 UNION R_5//budget). The formal algorithm, **Primitive**, is given below:

Algorithm: **Primitive**
Input: *Q*, *ACR*; Output: *Q'*

N := projection nodes of *Q*;
S := all rules in *ACR* having the same "role" as *Q*;
For all *s* in *S*
  If *s* has incompatible projection nodes from *Q*
    If *s* has LC type, then remove *s* from *S*;
    If *s* has RC type, then append "//N" to *s*;
P:= rules in *S* with + sign, P_1, ..., P_i;
M := rules in *S* with – sign, M_1, ..., M_j;
Q' = Q INTERSECT (P_1 UNION ... UNION P_i) EXCEPT (M_1 UNION ... UNION M_j);

Note that the final safe query Q' contains no special operator to ensure security. Any XML databases supporting XPath and set operators would suffice.

Lemma 1: *Time complexity of the **Primitive** algorithm is O(n), where n is the size of ACR (i.e., the number of rules)* . ■

Note that the above Primitive algorithm is correct with respect to the "answer-as-nodes" model, but not for the "answer-as-subtrees" one. To demonstrate the reason, consider the following rule:

R_6:(manager, /dept[name='HR']//budget/secret, read, −, RC)

The characteristic of R_6 is that its scope is "contained" by that of Q. Therefore, the correct semantics of Q and R_6 together is to let Q access all <budget> elements and their descendents, except <secret> elements and their descendants. However, this notion cannot be precisely captured by using the set-difference operator like in "Q EXCEPT R", nor by the Primitive algorithm like in "Q EXCEPT R//budget". To correctly handle this case, we need a new notion of "bite" operator like in "Q bitten-by R", pruning subtrees of denied parts of R from accepted parts of Q. That is, in the answer-as-subtrees model, when a node N is evaluated to be accessible, that does not automatically imply that the subtree of N be accessible. In this case, those violating descendents in the subtree of N must be "bitten" by some post-filtering. However, the bite operator is not currently supported in any known XML

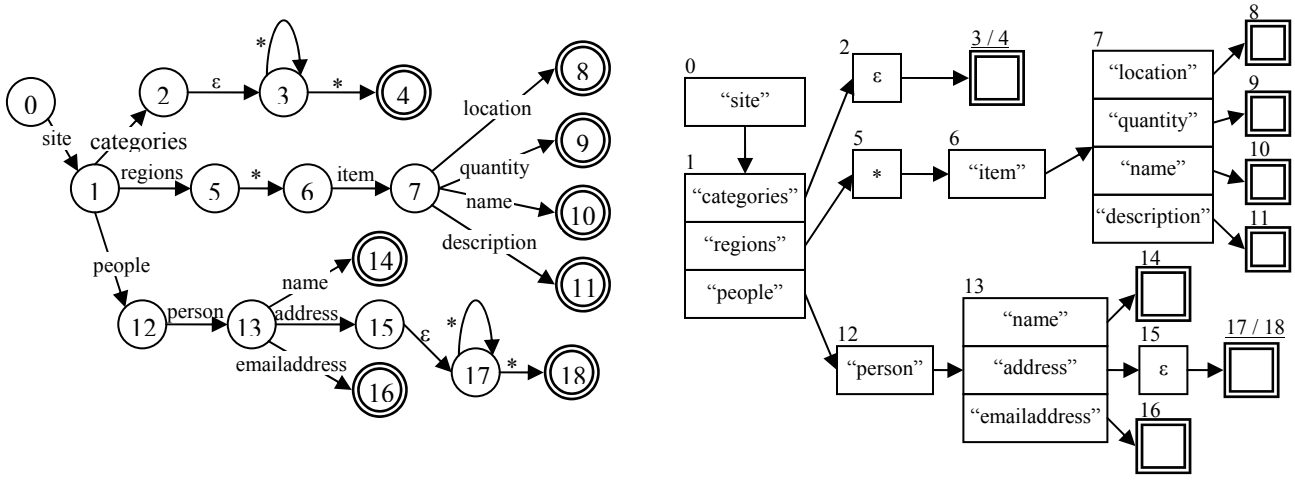


Figure 2. State transition map and NFA of the QFilter

algebras or engines. This problem occurs in both primitive and pre-processing approaches when answer-as-subtree model is used. We leave the study of the answer-as-subtree model and its support via the bite operator as future work, and do not consider the issue anymore; all the following algorithms and experimentations are presented and conducted only for answer-as-nodes model.

3.2 Post-Processing Approach

The post-processing strategy extends regular query processing by going through a "post-filtering" stage, named as *AFilter,* to filter out un-safe answers. Despite their potential inefficiency for unnecessarily carrying un-safe data till the last step, this approach is simple to implement. Moreover, when ACR and data are stored separately in some distributed environment (e.g., database-as-a-service model), this approach may be useful. However, despite the simple look on the surface, its implementation needs to overcome the following technical issue. Consider Q:/dept//budget and R_1:(User, /dept/south/budget, read/write, −, LC). When Q is first evaluated against an XML document D, Q projects out only the tag <budget> without its ancestor tags. Therefore, in the post-filtering stage, when R_1 is to be evaluated against these intermediate answers having only <budget> tags, it cannot check whether the <budget> satisfies /dept/south or not. However, if underlying XML database can produce <budget> as well as all its ancestor tags (e.g., using a recursive function of XQuery), then the post-processing approach can be applied without any further security support from databases. In our experiments, we used YFilter [6] as an implementation of the AFilter concept, and used an external script to recover ancestor tags. However, in the experimental comparison, this extra time to recover ancestor tags is not included.

4. PRE-PROCESSING APPROACH

The two approaches introduced above are relatively simple to implement, and thus can be considered as practical solutions. However, the overall performance may suffer from their naïveness. To remedy this problem, here, we detail the *pre-processing* approach, which shares the similar idea as primitive approach, but, instead of producing a potentially complicated (and thus expensive) query Q', it handles the set operations in a more efficient way (e.g., early-pruning). Consider the query Q:/dept[year<2004]//budget and a rule R_1:(manager, //south/budget, read/update, +, LC). Primitive approach would have generated Q' = /dept[year<2004]//budget INTERSECT //south/budget. However, by filtering out a portion of the Q judiciously, a re-written query Q":/dept[year<2004]/south/budget would satisfy both Q and R_1 while it is likely to be processed much faster than Q'. Similarly, for R_2:(manager, /dept//*, read, −, RC), all the contents that Q is asking for are forbidden by R_2, thus pre-processing approach could simply return null to user outright. However, in the primitive approach, a query "/dept[year<2004]//budget EXCEPT /dept//*//budget" would have been submitted to the underlying database, potentially wasting substantial amount of time.

The challenge is, therefore, to devise a *Query Filter* (named as *QFilter* here) that is capable of quickly filtering out conflicting or redundant parts from the original query Q to yield a new query Q' while ensuring: (1) Q and Q' is equivalent (i.e., producing the same answers); and (2) Q' is evaluated faster than Q.

4.1 QFilter at a Glance

The QFilter reads as input query Q, access control rules ACR, and (optional) schema S, then returns a modified query Q' as output:

$$Q' := \textbf{\textit{QFilter}}(Q, ACR, S)$$

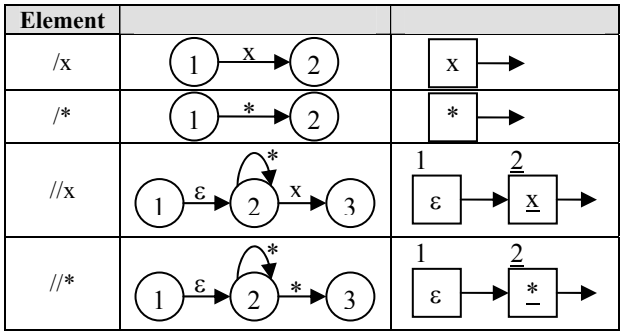
QFilter has three types of operations: (1) **Accept**: If answers of Q are contained by that of ACR^+ (i.e., Q asks for answers granted by ACR^+) and disjoint from that of ACR^- (i.e., Q does not ask for answers blocked by ACR^-), then QFilter accepts the query as it is: Q' = Q; (2) **Deny**: If answers of Q are disjoint from that of ACR^+ (i.e., no answers to Q are granted by ACR^+) or contained by that of ACR^- (i.e., all answers to Q are blocked by ACR^-), then QFilter rejects the query outright: Q' = {}; (3) **Rewrite**: if only partial answer is granted by ACR^+ or partial answer is blocked by ACR^-, QFilter rewrites *Q* into the ACR-obeying output query *Q'*. For instance, consider the following access control rules (individually) and queries:

R_1: (role, /people/person/name, read, −, LC)
R_2: (role, /people//address//*, read/update, +, RC)
R_3: (role, /regions/namerica/item/name, read, +, LC)
Q_1: /people/person/address/street
Q_2:/people/person/creditcard
Q_3:/regions//*

Then, Q_1 is accepted by both R_1 and R_2, denied by R_3. Similarly, Q_2 is accepted by R_1, denied by both R_2 and R_3; and Q_3 is accepted by R_1, denied by R_2, and rewritten to /regions/namerica/item/name by R_3. In sections 4.2 and 4.3, we show how QFilter is constructed and executed for the rules with "+" sign and "LC" types, and later in sections 4.4, 4.5 and 4.6, we extend this basic QFilter to cover more complex cases.

4.2 QFilter Construction

We consider XPath expressions of *ACR* as compositions of "four" basic building blocks: /x, /*, //x, and //*. Complex XPath expressions with predicates (e.g., /x[y='c']) can also be handled and are further described in Section 4.4. The NFA fragment construction for each building block is illustrated below:



For a complete XPath expression, NFA fragments are constructed upon path elements and then linked in sequence. For a set of rules that form the ACR, NFA for each rule is constructed and all the NFAs are combined in the way that identical states are merged. The processing is similar to regular NFA construction. We now give an example to illustrate the process. Consider the following eight XPath expressions that are the object parts of access control rules (now we ignore their type or action parts for simplicity):

R_1: /site/categories//*
R_2: /site/regions/*/item/location
R_3: /site/regions/*/item/quantity
R_4: /site/regions/*/item/name
R_5: /site /regions/*/item/description
R_6: /site /people/person/name
R_7: /site /people/person/address//*
R_8: /site /people/person/emailaddress

We construct the QFilter starting from R_1. For element /site, we create state 0 and a transition on token "site" to state 1. Then a transition on token "categories" is created on element /categories. For element //*, transition from state 2 to 3 and then 4 is created as shown in Figure 2 (left). Transition from state 3 to 4 requires at least 1 token after the ε transition. We use the "next-token-driven ε transition" in the NFA execution, thus state 3 and 4 could be merged in the NFA and set as acceptable state. The remaining access control rules are processed accordingly. Finally, the state transition map and the NFA corresponding to the above eight access control rules are shown in Figure 2.

BuildNFA, the algorithm to construct an NFA from ACR, as illustrated above, is straightforward and omitted. It is not difficult to see that the time complexity of this algorithm is *O(n)*, where *n* is the size of ACR (i.e., the number of rules). Both of Q and R consist of the four basic elements as described above. Next we provide detailed discussion of the NFA execution in those four cases: (1) only /x in both Q and NFA; (2) only /x in Q while /*, //x, and //* exist in NFA; (3) /* exists in Q; and (4) //x and //* exist in Q.

4.3 QFilter Execution

Given a query *Q* as input to the QFilter constructed as above, the output is a filtered query *Q'*. The filtering principle consists of: (1) if ACR allows all data that *Q* requests, keep *Q* as it is; (2) if what *Q* asks for is entirely prohibited by ACR, then reject *Q*; and (3) otherwise, modify *Q* such that *Q'* returns a precise "intersection" of *Q* and *ACR* (or precise "difference" for − sign). The filtering process becomes complicated when either Q or ACR has non-deterministic operators such as "//" and "*", which can match multiple branches in the NFA.

1. Deterministic transitions: There is only one deterministic transition, "/x", among four basic elements. In this case, the QFilter works exactly like regular NFA; an incoming query is either accepted or denied by the automaton, and the output of filtering is either the incoming query itself (if accepted) or empty string (if denied). For instance, when a query /site/people/person/name is executed, it passes through state 0→1→12→13 of the state transition diagram in Figure 2 and is finally accepted at state 14. Similarly, a query /site/people/person/creditcard passes through state 0→1→12 and rejected.

2. Non-deterministic transitions: This occurs when there is only direct child expressions (/x) in Q but more than one possible outgoing transitions (i.e., * and ε transitions) exist besides deterministic ones. We follow all possible transition paths through the NFA. Particularly, the //x and //* states are recursively processed (e.g., the underlined states 3/4 and 17/18 shown in Figure 2, right). If any of the paths ends at an accept state (i.e., the query is acceptable by at least one of ACR), the original query is passed through the NFA. For example, a query /site/regions/namerica/item/name passes through state 1→5 to state 6 since wildcard "*" accepts token "namerica".

3. Query rewriting at wildcard *: A query with wildcard "*" normally matches more than one state transitions. Taking Q:/site/* as an example, it moves from state 1 (/site) to state 2 (/site/categories), state 5 (/site/regions), and state 12 (/site/people). Here wildcard "*" means it can transit from the current state to any of its directly subsequent states. At any state, if the next input token in the query is "/*", we break the query into several branches in accordance with all the direct children of the current state. In each branch, we rewrite the "*" operator in Q with the corresponding path transition token, e.g., the /site/* is broken into three branches at state 1, and for instance, the branch transiting to state 2 is rewritten into /site/categories. "*" operator is kept only if a corresponding "*" transition exists, thus we mark this branch as the original query. We go on executing each branch of the query. If a branch of the original query exists and ends at an accepted state, the output of QFilter is the Q itself. Otherwise, the output is the union of all the accepted branches of the Q.

4. Query rewriting at "//" state: Both "//x" and "//*" in Q mean the state transition from the current state to all its subsequent states. In this case, the query is broken into branches that continue at each of the subsequent states of the current state. Such a query needs to be rewritten. Generally speaking, we rewrite first slash of the //x or //* token with the path from the current state to the destination state (where the branch continues to be executed). Each branch of the QFilter execution restarts from the second slash. A mapping table can be created to make these rewriting faster. As an example,

given a query Q:/site/people//name, /site/people starts the state transition like 0→1→12. Then, when it encounters the "//", it breaks Q into the following branches 6 branches:

1. /site/people/name restarting at state 12
2. /site/people/person/name restarting at state 13
3. /site/people/person/name/name restarting at state 14
4. /site/people/person/address/name restarting at state 15
5. /site/people/person/emailaddress/name restarting at state 16
6. /site/people/person/address//name restarting at state (17/18)

The next input token of the query at each branch is /name. Obviously only the branches 2 and 6 are accepted. Thus the final output is: Q' = /site/people/person/name UNION /site/people/person/address//name. In order to speed up the traversal of all the sub-states of the current state, we build a look-up table for each state. It is an index to all the sub-states, together with the replacing string. As an example, the look-up table of state 12 is as follows:

Start	Destination	Rewrite Query
12	12	
	13	/person
	14	/person/name
	15	/person/address
	16	/person/emailaddress
	17/18	/person/address/

4.4 Handling Predicates

In Section 4.3, the QFilter execution algorithm is based on the simple case where "predicates" (e.g., "[b=10]" in //a/[b=10]/c) are not used in Q or ACR. Here, we extend this to handle predicates. First, when input query has predicates in it, they are simply kept, and whenever a path token of the query is accepted or rewritten, the predicate (if any) is appended to it; otherwise, if a path token is rejected, the predicate is also rejected. Second, when predicates exist in ACR, the handling step needs a bit more elaboration: (1) predicate locating and (2) predicate merging.

Predicate Locating is to locate the "*corresponding"* predicates of *ACR* and Q. In QFilter execution, path elements from queries are matched with elements from rules (rewritten is regarded as a special matching). Predicates of matching path elements are called "*corresponding*" predicates. For instance, if we apply security check ignoring the predicates, R: "/site/*/*/item[name]/location" accepts Q: "/site/*/*/item[@quantity>5]/location". In this case, the path element "/item" in Q matches "/item" in R, thus their predicates "[name]" and "[@quantity>5]" are called *corresponding* predicates. A path element without predicate is treated as empty predicate; an actual predicate may correspond to an empty predicate.

QFilter processes query strings via extended NFA, which essentially conducts token matching. We further match their corresponding predicates using the *predicate processing states*. Different from a regular QFilter state, which conducts matching operation, a predicate processing state has the following properties: (1) token from the predicate of input query string is not going to "match" the token of the predicate processing state; the processing in the state is not *accept/rewrite/deny;* rather, it is *"Predicate Merging";* (2) different predicate processing states at the same location are not exclusive; route of input query is broken into different branches upon different predicate processing states.

For instance, from the example of section 4.2, suppose we replace R_4 by the new R_4: "/site/regions/*/item[description]/name". Then the QFilter shown in figure 2 is re-constructed as shown in figure 3 (each state with children carries an empty predicate processing state ("φ"), but omitted for simplicity.) When an input query "/site/regions/*/item[@quantity>0]/name" is processed, it goes through states 0→1→5→6. After token "item" is accepted at state 6, the route breaks into two branches: (1) the predicate [@quantity>0] of the query is corresponded with φ predicate, a *predicate merging* operation is conducted, and QFilter execution continues at state 7.1 with token "name"; (2) "[@quantity>0]" is corresponded with "[description]" and they are merged, QFilter execution continues at state 7.2 with token "name".

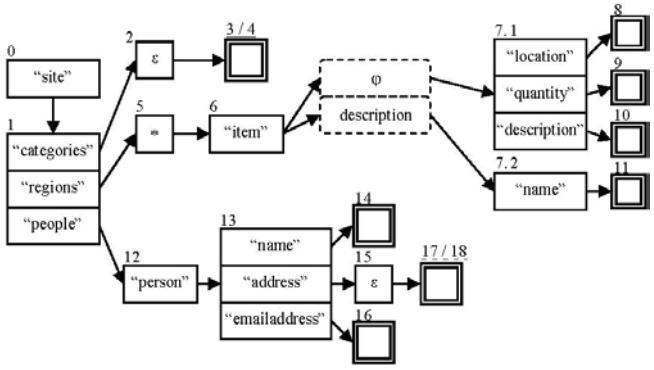


Figure 3. QFilter with predicate processing states

Predicate Merging is to merge the corresponding predicates located in the previous step. In QFilter, two strings are simply connected. For query "/site/regions/*/item[@quantity>0]/name" in the previous example, its route breaks into two branches at state 6: (1) "[@quantity>0]" merges with φ yielding "[@quantity>0]" itself; (2) "[@quantity>0]" merges with "[description]" yielding predicate "[@quantity>0][description]". Branch 1 is denied at state 7.1 while branch 2 is accepted at state 11. Thus, the final output is: "/site/regions/*/item[@quantity>0][description]/name".

Some other optimizations are possible on the merged predicates (which are better off handled by query optimizer.) E.g., (1) [@quantity>0][@quantity>5] can be fused as [@quantity>5]; and (2) query /*/*/person[@id="1"][@id="2"] should be rejected.

4.5 Handling Rules with RC Type or – Sign

So far, we have discussed the QFilter when ACR has only rules with + sign and LC type. In this section, let us extend QFilter to include rules with – sign and RC type. First, as first proposed in [13], all rules with RC type are equivalent to three rules with LC type. For instance, /x with RC type is semantically equivalent to three expressions: /x, /x//*, /x//@* with LC type. Therefore, by re-writing all rules with RC type into equivalent ones with LC type, QFilter can be constructed as it is. Next, consider the case of rules with – sign. When ACR contains rules with + sign (i.e., ACR^+) and ones with – sign (i.e., ACR^-), the overall semantics of output query Q' is (as already shown in the **Primitive** algorithm):

$$Q' = Q \text{ INTERSECT } (ACR^+ \text{ EXCEPT } ACR^-)$$
$$= (Q \text{ INTERSECT } ACR^+) \text{ EXCEPT } (Q \text{ INTERSECT } ACR^-)$$

Now, note that "Q INTERSECT ACR" can be computed by QFilter(Q, ACR) with the optional schema S removed for simplicity. Therefore, by having two separate QFilters for + and – sign rules, one can get the following final formula for Q':

$$Q' = QFilter(Q, ACR^+) \text{ EXCEPT } QFilter(Q, ACR^-)$$

where the algorithm for QFilter is shown below:

Algorithm: **QFilter**
Input: Q, ACR; Output: Q'

NFA := BuildNFA(ACR);
Switch (Q, NFA):
Accpet: Q' := Q;
Deny: Q' := {};
Rewrite: Q' := rewrite Q according to Sec 5.3 and 5.5;

4.6 Security Analysis

The security of our QFilter approach can be proved using the following three theorems.

Theorem 1: *The QFilter execution algorithm always generates the correct answer when Q and object parts of ACR are limited to XPath expressions without predicates; i.e.*

$$Q' = \boldsymbol{M}(Q) = Q \cap ACR \quad (1)$$

***M**(Q) refers to the evaluation of a query Q against the extended NFA, **M**, created by QFilter algorithm.* ■

Sketch of Proof:
To prove equation 1, we should have:

$$Q' \subseteq (Q \cap ACR) \quad (2)$$
$$\text{and} \quad (Q \cap ACR) \subseteq Q' \quad (3)$$

(2) is amount to:

$$Q' \subseteq Q \quad (4)$$
$$Q' \subseteq ACR \quad (5)$$

Since the only case where **M** behaves different from regular automata is for the wildcard matching and query rewriting operations, we focus on that.

Equation (4) can be proved as follows: For the accepted queries, the output of QFilter is the original query itself: M(Q) = Q. For the rejected queries, the output of M is an empty string or error message, where we also have M(Q) = φ ⊆ Q. For the rewritten queries, since the rewriting algorithm only changes wildcards into more specified path strings, it is obvious that M(Q) ⊆ Q.

Equation (5) can be proved by constructing an *equivalent path expression* (EPE), which is a subset of the access control automata that routes to the query. The construction of EPE is as follows:

- EPE starts with an empty string.
- EPE is extended along with the processing of query expression, i.e. EPE is extended accordingly at each NFA operation.

By constructing the EPE for all possible cases and show them to be contained in one of the ACR rules, one can show EPE ⊆ *ACR,* and in turn since M(Q) ⊆ EPE, **M**(Q) ⊆ EPE ⊆ *ACR*. The details of all cases are omitted due to space constraint.

Equation (3) can be expressed as:

$$(Q \cap ACR) - \mathbf{M}(Q) = \boldsymbol{\varphi}$$

Which is equivalent to:

$$[ACR - \mathbf{M}(Q)] \cap [Q - \mathbf{M}(Q)] = \boldsymbol{\varphi}$$
$$\text{Or} \quad [ACR - \mathbf{M}(Q)] \cap Q = \boldsymbol{\varphi} \quad (6)$$

Make us of the EPE defined above, (6) is decomposed into:

$$[(ACR - \text{EPE}) \cup (\text{EPE} - \mathbf{M}(Q))] \cap Q = \boldsymbol{\varphi}$$

This is equivalent to:

$$[(ACR - \text{EPE}) \cap Q] \cup [(\text{EPE} - \mathbf{M}(Q)) \cap Q] = \boldsymbol{\varphi}$$

From the construction of EPE, we can see the condition that an EPE path is more general than the output query is when the input element from query Q is a more specified path while it routes through a wildcard path of NFA. Thus:

$$[(\text{EPE} - \mathbf{M}(Q)) \cap Q] = \boldsymbol{\varphi}$$

Therefore, we only need to show:

$$[(ACR - \text{EPE}) \cap Q] = \boldsymbol{\varphi}$$

Note that EPE is exactly the state transition route of Q through the NFA. At the beginning of NFA execution, EPE is empty that it has the probability to go through any state of NFA. Hence the maximum size of the EPE is bounded by that of *ACR* and the set (*ACR* - EPE) is empty. Whenever one does a state transition, a token from the query Q enables this state transition thus constrains the EPE. In this way, some states of ACR is removed from the potential EPE into set (*ACR* - *EPE*), because it has conflict with the query Q. Hence we have:

$$[(ACR - \text{EPE}) \cap Q] = \boldsymbol{\varphi} \quad \text{(Q.E.D)}$$

Theorem 2: *The QFilter execution algorithm always generates the correct answer, when (1) queries are (arbitrary) XPath expressions, and (2) object parts of ACR are limited to XPath expressions without predicates.* ■

Sketch of Proof: For query Q with predicates, we remove predicates to construct Q^*, which is XPath expression without predicates, and $Q \cap Q^* = Q$. Let $Q^{*\prime}$ be the output of QFilter with Q^* as input. From theorem 1 we have:

$$Q^{*\prime} = \boldsymbol{M}(Q^*) = Q^* \cap ACR$$

The direct predicate processing method in QFilter execution is to insert predicates in Q to its corresponding path in $Q^{*\prime}$. Thus we have:

$$Q' = Q^{*\prime} \cap Q = Q \cap Q^* \cap ACR = Q \cap ACR \quad \text{(Q.E.D.)}$$

Theorem 3: *The post-processing predicate verification algorithm of Section 4.4 always generates the correct answer, when object parts of ACR are XPath expressions that do not have both "//" path and predicate in one rule.* ■

Sketch of Proof: For rules without predicates, theorem 2 already proved the correctness of QFilter approach. For rules with predicates, referring to our algorithm, suppose Q' is accepted/rewritten by a rule R which has predicate. Since R do not have "//" path, Q' cannot have "//" path either, thus Q' and R have the same number of path elements. By inserting the predicate in R into its counterpart in Q', we generate Q'' which is restricted by both Q' and R. (Q.E.D)

4.7 Computational Complexity

Computational complexity of QFilter includes the computation of QFilter construction and execution. For QFilter construction, the complexity is $O(n)$, where n is the number of path steps in XPath expressions of ACR. For QFilter execution, we only provide upper bounds on the computational complexity: (1) When there is no wildcard in a query, the computation of filtering that query is $O(m)$, where m is the number of path steps in a query. The upper limit is reached for the cases of accepted or rewritten queries; (2) When wildcard "*" exists in the query, the complexity is denoted as $O(n)$, where n is the size of NFA. The worst case occurs only for input "/*/*…/*", which requires the traversing of entire NFA; (3) For queries with "//" paths, the computation is $O(m*n_1*n_2*...*n_k)$, where m is the number of path steps in a query, k is the number of wildcard "//" in a query and n_i is the size of the child QFilter at the state which first meets the i^{th} "//" path. This is still quite acceptable since: (1) probability of "//" paths in queries are relatively low, and

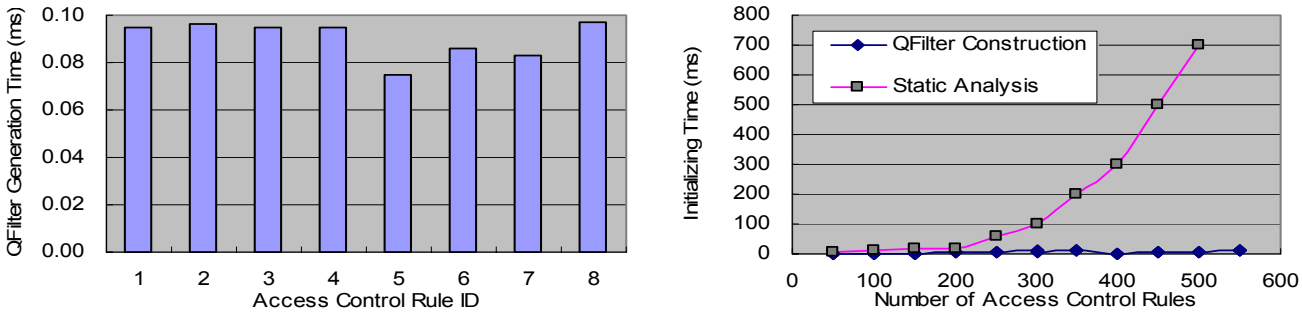


Figure 4. QFilter construction performance (QFilter vs. Static Analysis [21]).

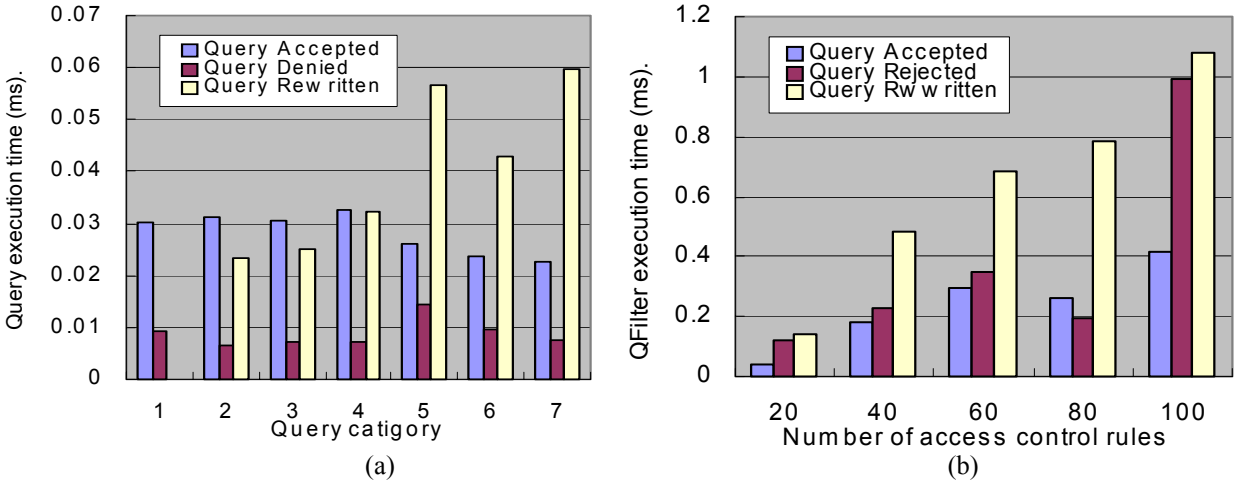


Figure 5. Average query execution time of three types of QFilter.

(2) this worst case only occurs for a query like "/……//*//*…//*", which seldom appears in real-world XML queries. Overall, QFilter is very practical since the worst case for filtering rarely occurs. We experimentally verify this claim in the following section.

5. EXPERIMENTAL RESULTS

5.1 Set-up

We used the well-known XMark schema [18] and its XML document generator to generate test XML documents. Since the size of test data was not a major factor to determine the performance of various methods, here we present only the case of test set with 1.5 MB (The types of queries or number of access control rules were more important in our experimentation, and thus carefully selected and measured.) As an underlying XML database, we used Galax 0.3.1 [19] that can evaluate XQuery efficiently. Pre-processing approach, QFilter, was implemented in Java (JDK 1.4.2) and communicated with Galax through its Java-API. For post-processing approach, we used the YFilter [8] from UC Berkeley as an implementation of AFilter.

For XPath expressions in Q and ACR, both user-defined (UD) as well as synthetic (SN) tests were used. That is, we have four test cases by combining two factors in two dimensions; UD-Q/UD-ACR, UD-Q/SN-ACR, SN-Q/UD-ACR, and SN-Q/SN-ACR. All synthetic XPath expressions were generated by YFilter package. We also prepared two kinds of access control policies in the experiments: (1) user defined and (2) synthetic roles. For instance, the following Costumer Advertisement Manager (CAM) is a user-defined role extended from the example in Section 4; CAM is in charge of delivering advertisements to costumers, thus is permitted to access users information except for their credit card, profile, and item' basic information. This policy can be captured by the following rules (all rules with RC type are already converted to equivalent ones with LC type [13]):

1. (CAM, "/site/regions/*/item/location", +, LC)
2. (CAM, "/site/regions/*/item/quantity", +, LC)
3. (CAM, "/site/regions/*/item/name", +, LC)
4. (CAM, "/site/regions/*/item/description", +, LC)
5. (CAM, "/site/categories", +, LC)
6. (CAM, "/site/categories//*", +, LC)
7. (CAM, "/site/people/person", +, LC)
8. (CAM, "/site/people/person//*", +, LC)
9. (CAM, "/site/people/person/creditcard", −, LC)
10. (CAM, "/site/people/person/profile", −, LC)

Then we build two extended NFAs for + and - rules respectively, thus construct a QFilter. Since the performance of QFilter is dominated by handling such building blocks as "//" paths, and since predicates (in either queries or access control rules) have almost no impact on the performance of QFilter, in our experiments we focus on the other building blocks instead of predicates.

5.2 QFilter vs. Static Analysis

Note that the static analysis method of [13], which is the only pre-processing based method like ours, can handle only two cases; i.e., access fully granted ($Q \subseteq R$) or access fully denied ($Q \cap R = \varphi$), where Q is an input query and R is ACR^{+}. However, our QFilter method is able to process all three cases; i.e., $Q \subseteq R$, $Q \cap R = \varphi$, and

partial overlap ($Q \not\subset R \wedge Q \cap R \neq \varphi$). Therefore, QFilter method can run on any XML databases whether or not they have security support, which is not possible for [13].

To validate our claim, we first compared our algorithms against the known pre-processing approach, *Static Analysis* method [13]. Since the end-to-end processing time (i.e., from the moment a query is submitted to the time "safe answers" are returned) of [13] was not available to us, we only compared time to construct and to check security policies between QFilter and static analysis methods.

Filter Construction: In real world applications, QFilter is likely to be constructed offline. Once the service starts, we do not need to modify or reconstruct the QFilter unless access control rules are changed. Thus, the speed of QFilter construction is of less importance to users. Nevertheless, experiments show that QFilter construction is fast enough to be done in on-line. We first construct a separate QFilter for user-defined rules of the CAM role and show the times in Figure 4(a). QFilter construction time for different rules mainly depends on the complexity of the XPath expression, i.e., number of QFilter states to be built. Next, to see the impact of the number of rules on QFilter construction, we randomly generated 550 XPath expressions and measure the construction time by treating those expressions as ACR. Furthermore, to directly compare QFilter against [13], all synthetic rules were carefully generated according to the XML specification DTD (xmlspec-v21.dtd). In each experiment, + and – signs are 50% each. Figure 4(b) shows the results. Although BuildNFA algorithm has O(n) time complexity, it appears to be "flat", taking only 17 mille-seconds for 550 rules. Therefore, the QFilter approach has minimal overhead in updating the access control policy. However, the initialization of the other pre-processing approach in [13] was very sensitive to the number of rules, so that the graph increases sharply. Note the initialization time of static analysis mechanism was estimated from Figure 7 of [13] where it supports predicates using upper-bound and low-bound constrainers.

Filter Execution: After QFilter is created with ACR, we use it to filter the input query Q to yield safe query Q'. Using the CAM role, we first tested how the types of user query Q affect the filtering speed. That is, we prepared *seven* different query categories and for each category, we generated 100 synthetic queries based on the XMark DTD: (1) Queries with deterministic paths only, i.e., no wildcard and "//" paths; (2) Queries with one wildcard "*"; (3) Queries with two "*" wildcards; (4) Queries with more than two "*" wildcards; (5) Queries with one "//" path; (6) Random queries with 0.1 wildcard probability and 0.1 "//" probability at each location; and (7) Random queries with 0.2 wildcard probability and 0.2 "//" probability at each location. Using these random XPath expressions as input queries to QFilter, and we measured the average execution time for each query category and for each output type (accept, deny and rewrite) of QFilter. The results are shown in Figure 5. From this, we can see that QFilter is generally faster in accepting and denying queries, but slower to rewrite queries with wildcards, especially with "//" paths. This is because QFilter needs to traverse more states to process "*" and "//".

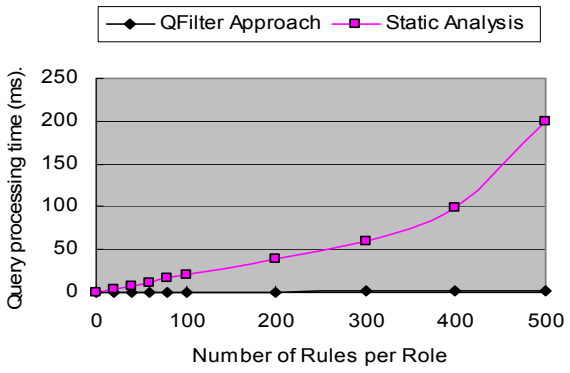


Figure 6: XPath security check time.

Next, we tested how QFilter execution performance degrades as the number of ACR increases. We constructed a QFilter using 20 to 100 synthetic rules based on XMark DTD (SN-ACR) and tested with random queries (SN-Q). Figure 5(b) shows the average QFilter execution time for each rule set for each output type (accept, deny and rewrite). By and large, as the number of ACR increases, the QFilter execution time to filter out conflicting parts from Q increases too. This is understandable since there are more branches to test in QFilter. However, note that the longest time it took to rewrite Q, when QFilter has 100 synthetic rules, was still negligible with only 1 millisecond.

To better show the scalability of QFilter, next, QFilter's execution time was compared to that of [13], which essentially spend substantial time to check the containment of two automata. The result is shown in Figure 6. When 500 synthetic rules were used, QFilter was faster than the static analysis method of [13] up to 200 times, thanks to the QFilter's NFA-based query rewriting of the partial overlap case: ($Q \not\subset R \wedge Q \cap R \neq \varphi$).

5.3 Primitive, Pre- and Post-processing

Next, we compared the end-to-end processing time among four approaches of Figure 1: (1) No security check is made (thus final data is un-safe); (2) Primitive approach; (3) AFilter (post-processing); and (4) QFilter (pre-processing). End-to-end query processing time denotes the total time, in logarithmic scale, needed to process Q. In Figure 1, (a) indicates the query processing without any security check, where the output document UD is un-safe, the end-to-end time is mainly evaluation time of Q; (b) indicates the primitive approach, which generates the safe result D, the end-to-end time is mainly the intersect query ($Q \cap ACR$) evaluation time; (c) shows the QFilter approach, where the end-to-end time includes the QFilter construction time and filtered query (Q') evaluation time; and (d) indicates the post-processing approach, where the end-to-end time includes the original query evaluation time and un-safe answer, UD, filtering time. Note that we do not count the I/O time of the query input and the answer output. Note that for (d), due to the problem described in Section 3.2, we manually wrote an external script to recover ancestor tags when UD is generated, but to be fair, that extra time for running script was not counted in. However, it is worthwhile to point out that if one uses the recursive function of XQuery to implement this in XML databases, the cost would have been even higher. Thus, what we report here for post-processing approach is a significant "under-estimate".

Figure 7 summarizes the comparison of the four approaches. QFilter-based pre-processing approach is a clear winner regardless of the query categories, and thus a promising solution for XML access controls; it significantly outperforms the primitive approach and an (un-safe) query processing which does not enforce XML access control. Interesting phenomenon is that QFilter even outperforms no security check case even for the queries re-written. This implies that when Q is filtered to Q' by QFilter, as a side-

effect, Q' was optimized so that Q' is processed more efficiently than Q. That is, when Q' is processed by Galax, since its query constraints have been tightened by additional conditions added by QFilter, it is typically much faster than the original query, while ensuring returning only safe answers.

Since the post-processing approach requires a data filtering stage after Q is evaluated, thus it is surely slower than the original query processing and much slower than QFilter approach. In many cases, QFilter can quickly determine whether the query is fully "Accepted" or "Denied" where the query filtering time is negligible compared to potential save from unnecessary query evaluation time.

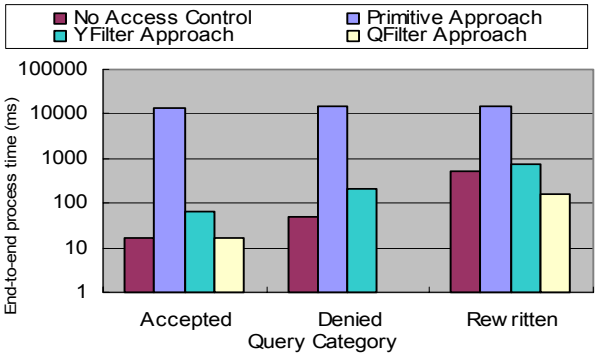


Figure 7. End-to-end query processing time comparison of all approaches.

6. CONCLUSION AND FUTURE WORK

As the distribution and sharing of information over the Web are getting increasingly important, they mandate efficient yet secure access of XML data. It is necessary to tailor information in XML documents for various user and application requirements, preserving confidentiality and efficiency at the same time. Thus, it is critical to specify and enforce access control over XML data to ensure that only authorized users have an access to the portion of the data they are allowed to. In this paper, we consider several practical approaches that support XML access controls without relying on security features of underlying XML databases. A pre-processing based method, called QFilter, has been developed and shown to be particularly efficient. QFilter, based on Non-deterministic Finite Automata (NFA), rewrites user's query to a new one that will not return data violating access control rules. We prove the security of the QFilter via theoretical analysis and demonstrate its performance through extensive experiments. Result shows that QFilter is superior to the other state-of-the-art techniques.

7. ACKNOWLEDGEMENT

Authors would like to thank Yanlei Diao for providing the YFilter software package [6], and Makoto Murata for providing their source data of [13] for comparison. Also, the comments of anonymous reviewers and S. Sudarshan significantly helped to improve the paper.

REFERENCES

[1] E. Bertino and E. Ferrari. Secure and Selective Dissemination of XML Documents. ACM TISSEC, 5(3):290–331, Aug. 2002.

[2] T. Bray, J. Paoli, and C. M. Sperberg-McQueen (Eds). Extensible Markup Language (XML) 1.0 (2nd Ed.). W3C Recommendation, Oct. 2000..

[3] S. Cho, S. Amer-Yahia, L. V.S. Lakshmanan, and D. Srivastava. Optimizing the Secure Evaluation of Twig Queries. In VLDB, Hong Kong, China, Aug. 2002.

[4] E. Damiani, S. De Capitani di Vimercati, S. Paraboschi, and P. Samarati. A Fine-Grained Access Control System for XML Documents. ACM TISSEC, 5(2):169–202, May 2002.

[5] E. Damiani, S. De Capitani Di Vimercati, S. Paraboschi, and P. Samarati. Design and Implementation of an Access Control Processor for XML Documents. Computer Networks, 33(6):59–75, 2000.

[6] Y. Diao and M. J. Franklin. High-Performance XML Filtering: An Overview of YFilter. IEEE Data Eng. Bulletin, Mar. 2003.

[7] E. Fernandez, E. Gudes, and H. Song. A Model of Evaluation and Administration of Security in Object-Oriented Databases. IEEE TKDE, 6(2):275–292, 1994.

[8] S. Godik and T. Moses (Eds). eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS Specification Set, Feb. 2003.

[9] M. Kudo and S. Hada. XML document security based on provisional authorization. In ACM CCS, 2000.

[10] D. Lee, W. C. Lee and P. Liu. Supporting XML Security Models using Relational Databases: A Vision. In *XML Database Symposium (XSym)*, Berlin, Germany, 2003.

[11] G. Miklau and D. Suciu. Containment and equivalence for an XPath fragment. In 21st ACM PODS, 2002

[12] J. Moffett, M. Sloman, and K. Twidle. Specifying Discretionary Access Control Policy for Distributed Systems, Nov. 1990.

[13] M. Murata, A. Tozawa, and M. Kudo. XML Access Control using Static Analysis. In ACM CCS, Washington D.C., 2003.

[14] S. Osborn. Mandatory Access Control and Role-Based Access Control Revisited. In ACM Workshop on Role Based Access Control, pages 31–40, Fairfax, VA, 1997.

[15] F. Rabitti, E. Bertino, W. Kim and D. Woelk. A Model of Authorization for Next-Generation Database Systems. ACM Trans. on Database Systems (TODS), 16(1):89–131, 1991.

[16] P. Samarati, E. Bertino, and S. Jajodia. An Authorization Model for a Distributed Hypertext System. IEEE Trans. on Knowledge and Data Eng. (TKDE), 8(4):555–562, 1996.

[17] R. Sandhu, E. Coyne, H. Feinstein, and C. Youman. Role-Based Access Control Models. IEEE Computer, 29(2), 1996.

[18] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, April 2001.

[19] J. Simeon and M. Fernandez. Galax V 0.3.5, Jan. 2004. http://db.bell-labs.com/galax/.

[20]T. Yu, D. Srivastava, L. V.S. Lakshmanan, and H. V. Jagadish. Compressed Accessibility Map: Efficient Access Control for XML. In VLDB, Hong Kong, China, Aug. 2002.

[21] R. Rizvi, A. Mendelzon, S. Sudarshan, P. Roy. Extending Query Rewriting Techniques for Fine-Grained Access Control. In ACM SIGMOD 2004.

[22] L. Bouganim, F. D. Ngoc,and P. Pucheral. Client-Based Access Control Management for XML documents. In VLDB, Toronto, Canada, 2004.