The VLDB Journal
DOI 10.1007/s00778-010-0202-x

REGULAR PAPER

QFilter: rewriting insecure XML queries to secure ones using non-deterministic finite automata

Bo Luo · Dongwon Lee · Wang-Chien Lee · Peng Liu

Received: 4 October 2009 / Revised: 5 August 2010 / Accepted: 18 August 2010

Abstract In this paper, we ask whether XML access control can be supported when underlying (XML or relational) storage system does not provide adequate security features and propose three alternative solutions —*primitive*, *pre-processing*, and *post-processing*. Toward that scenario, in particular, we advocate a scalable and effective pre-processing approach, called QFilter. QFilter is based on non-deterministic finite automata (NFA) and rewrites user's queries such that parts violating access control rules are pre-pruned. Through analysis and experimental validation, we show that (1) QFilter guarantees that only permissible portion of data is returned to the authorized users, (2) such access controls can be efficiently enforced without relying on security features of underlying storage system, and (3) such independency makes QFilter capable of many emerging applications, such as in-network access control and access control outsourcing.

Keywords XML · Security · Access control · NFA

This paper is extend from earlier conference version [35].

B. Luo (✉)
The University of Kansas, Lawrence, KS, USA
e-mail: bluo@ku.edu

D. Lee · W.-C. Lee · P. Liu
The Pennsylvania State University, University Park, PA, USA
e-mail: dongwon@psu.edu

W.-C. Lee
e-mail: wlee@cse.psu.edu

P. Liu
e-mail: pliu@ist.psu.edu

1 Introduction

The eXtensible Markup Language (XML) [8] has emerged as the de facto standard for storing and exchanging information on the Internet. As the distribution and sharing of information over the 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, while ensuring confidentiality and efficiency at the same time. However, document (file) level access control is not suitable for today's XML applications, where data access is typically performed at element and attribute levels. To remedy these shortcomings, various proposals in support of fine-grained XML access control have appeared. Most of them can be categorized as either *view-based* or *DBMS-based*. View-based approaches (e.g., [3,4,12,16,24]) identify accessible XML nodes for each user (role) to create a *view* and evaluate user queries on the view. Such approaches provide fast access to the authorized data (especially when views are materialized) but need to deal with view maintenance issues. DBMS-based approaches (e.g., [9,15,54]) tag each node with an authorization list and check accessibility for each candidate answer node during query evaluation. They are less complicate to maintain but require support from database engines. However, to our best knowledge, there are no off-the-shelf XML databases that provide fine-grained security features yet. Furthermore, when RDBMS is used to manage XML data, there is a compatibility issue between RDBMS and XML access control models [31]: (1) the data models are inherently incompatible, and not all data conversion algorithms fully preserve structural properties of XML data; (2) XML nodes are hierarchically nested, while cells in the relational model are impartible; hence, relational access control does not consider propagation issues; and (3) XML model requires fine-grained access control, while traditional RDBMS access control is only enforced at column level. Although record/cell level access control could be enforced with views or Virtual Private Databases (Oracle VPD), it can get very complicated when there are multiple polices that involve the same base table.

The goal of this study is *"to devise pragmatic solutions for implementing fine-grained XML access control that use* **no view-based access controls** *and require* **no security support** *from underlying databases."* We analyze and examine three different classes of solutions for XML access control, namely, *primitive*, *pre-processing*, and *post-processing*. In particular, we advocate a practical and scalable pre-processing solution, called QFilter, as an external component to the database engine. QFilter checks XML queries against access control rules and rewrites them such that parts violating access control policies are pre-pruned. Since QFilter does not use views, it entirely avoids the issues of storage and maintenance costs. Furthermore, since QFilter does not rely on security-related features of underlying databases (e.g., GRANT/REVOKE in RDBMS), it can work with any off-the-shelf XML databases. This property makes QFilter a very practical and flexible solution, especially in distributed environments.

Our contributions are as follows: (1) *We examine three alternative solutions to support XML access control*; (2) *We propose a novel technique,* QFilter*, that utilizes an NFA to rewrite insecure XML queries to secure ones*; (3) QFilter *enforces access control completely independent from underlying XML engine, hence provides remarkable flexibility*; and (4) *Through extensive experiments, we validate that* QFilter *is a practical and effective solution for XML access control enforcement.*

2 Background

2.1 Related work

Two research areas are the most relevant to this paper—XML access control *models* and *enforcement mechanisms*. The focus of this paper is on the latter.

1. Models. Most XML access control models inherit the framework of either role-based access control [47], in which users are assigned with roles and thus can exercise certain access rights characterized by the roles, or credential-based access control, where each user features a set of attributes and access rights are denoted based on the values of attributes. The difference between two models is mainly the way they identify *subjects*, i.e., users. However, this is not closely related to our topic, since we focus on access control enforcement, which mainly considers *objects*, i.e., XML documents and nodes.

Recently, several authorization-based XML access control models are proposed. In [16], an authorization sheet is associated with each XML document/DTD to express authorizations. Later, the authors extend this model by enriching authorization types and providing a complete description of the specification and enforcement mechanism [12]. This model is further extended in [51] to handle XQuery [5]. On the other hand, in [4], an access control environment for XML documents and techniques to deal with authorization priorities and conflict resolution is proposed. In terms of XML data objects, a general framework has been proposed in [22] to normalize data object specification in XML access control using XPath. Moreover, languages for access control policy are developed in XACL [28] and XACML [25]. More recently, ACCOn [7] considers inconsistency and security flaws introduced by XML write-access policies. Finally, the use of authorization priorities with propagation and overriding is related to similar techniques studied in OODB [19,46]. The above XML access control models can specify the authorizations of a subject against an XML data object without ambiguity. While an XML access control model can be enforced in various ways, the model cannot tell which enforcement mechanisms are better ones.

2. Enforcement. Run-time access control enforcement mechanisms implement security check inside database engines and enforce access control along with query evaluation. They first tag each XML node with a label [9,15,52,54] or an authorization list [27,53]. During query processing, XML engines traverse the subtrees of candidate answers and eliminate inaccessible nodes from the final answer. The traversal seriously slows down query processing. Recently, dynamic predicate [32] has been introduced to integrate security check into the query plan through dynamically constructed conditions. To further optimize query processing with access control, security-conscious indices are constructed for access control rules to speed up node-level security check [52]. However, this requires building an effective index for every rule, which is not practical. Otherwise, for queries that hit un-indexed rules, the engine still needs to take excessive efforts for node-level security checking. More importantly, since all the approaches patch on kernels of XML engines, none of them is adopted or could be easily adopted by commercial or open-source XML database vendors.

View-based approaches create and maintain a separate *view* for each role. Earlier approaches, such as [3,12,16,24,50], check the authorization at each node and compute a user view (the accessible portion of the XML document) to the requestor. Queries are then safely evaluated against such pre-built materialized views. With materialized views, query processing is usually very efficient since it bypasses on-the-fly security check, and queries are evaluated on a smaller XML document. However, it is challenging to maintain a large number of (frequently updating) views (one for each role): storage and view synchronization are major concerns. Incremental view adaptation [1] proposes to reduce the cost of view maintenance. Security views [18,29,30] propose to avoid view materialization by enforcing schema (DTD)-level access control through "virtual" security views. They publish a "safe schema", which only represents user accessible portion of the XML document (the view) for users to write queries. They translate user queries (against the security view) to equivalent queries against the original XML document. On the other hand, our proposal aims at avoiding using views entirely.

Pre-processing approaches check user queries and enforce access control before query evaluation, e.g., static analysis approach [41,42], function-based approach [45], access condition table approach [43] policy matching tree [44], secure query rewrite (SQR) approach [40]. Meanwhile, client-based access control [6] resembles post-processing approach. It enforces access control for streaming XML data at the client side using a filtering approach.

The Static Analysis approach [41,42] is the first attempt of non-view-based XML access control. It first converts an input query q to an NFA M_q and access control rules r to another NFA M_r. At static analysis, it (1) accepts q if $M_q \subseteq M_r$ (i.e., q asks for data that are "entirely" authorized) or (2) rejects q if $M_q \cap M_r = \emptyset$ (i.e., what q asks for is "entirely" prohibited). However, when q and r partially overlap, access is statically indeterminate, and run-time security check needs to be enforced. As shown in Sect. 5, since the majority of q and r belong to the partial-overlapping cases, the performance of [41] suffers. Inspired by [41], our QFilter approach (first appeared in [35]) discovered that although the access decision may not be made statically (without touching the data), a *safe query* could still be constructed. For instance, assume that access control rules allow user to read phone numbers of managers, but not employees. When a user asks for phone number of `John`, the access decision could not be made without retrieving the role of `John` from the data. However, we could rewrite the query into a safe one: the phone number of a `manager` named `John`. In the case that `John` is an employee (not a manager), such a query yields NULL answer. Therefore, the partial-overlapping case can also be handled without relying on security features from underlying databases.

Inspired by security view and query rewriting, a more recent work [14] annotates XML schema with access rights and converts it into a finite state automaton to rewrite queries. It requires the presence of original XML schema, which is not always available, especially in distributed environments, and in the case access control is provided by a third party (e.g., [33]). It eliminates wildcards and thus may cause unnecessary rewrite. In rare cases, such rewrite may change the order of nodes in the answer, which is considered wrong (XML nodes are defined as ordered). The proposal has not been implemented and tested in [14].

Finally, another emerging branch of XML access control focuses on sensitive information contained in XML tree structures [11,20,39]. Meanwhile, some recent works [7,13,21,23,38] propose to study XML access control for update operations. However, they are outside the scope of our research: access control of XML nodes against read operation.

2.2 Preliminaries

Since an XML document can be represented as a hierarchy of nested nodes, fine-grained access controls at node level are desired. Authorization in our study is specified via 4-tuple access control rules (ACR) = {**subject**, **object**, **action**, **sign**}, where (1) *subject* is to whom an authorization is granted (i.e., role); (2) *object* is set of XML nodes specified by an XPath expression; (3) *action* consists of read, write, and update; (4) *sign* $\in \{+, -\}$ refers to access "granted" or "denied", respectively. A node without explicit authorization is considered to be "access denied." When a node has multiple relevant rules, and conflict occurs between "+" and "−" rules, "−" rules take precedence.

Compared with the 5-tuple access control policy [12] used in many related works, we do not have the "type" field. The original 5-tuple ACR is represented as: $ACR = \{$**subject**, **object**, **action**, **sign**, **type**$\}$. Particularly, $type \in \{LC, RC\}$ refers to either local check (LC) or recursive check (RC). In local check, authorization is applied to only textual data of the context nodes, or sometimes attributes as well—"`self::text() | self::attribute()`". In RC, authorization is applied to context nodes and propagated to all the descendants—"`descendant-or-self::node()`". [41] converts ACR with RC type to a combination of three LC rules. In our model, "RC" type is enforced by default, i.e., access control specified on a node affects the whole subtree rooting at that node. This setting complies with the XML semantics, where a node is defined to include all the descendants that are nested between the starting and ending tags of the node. In other words, nodes are by default "RC" in XML standard [8]: querying for a node will yield the whole subtree (without the presence of further constraints such as access control). If a rule only applies to the text child of the context node, "`/text()`" is appended to the end of the XPath expression (object). In this way, we exactly follow XPath specification to identify XML nodes.

Like other XML access control approaches, we use XPath [2] instead of XQuery [5] to specify queries and AC rules, since XQuery uses XPath to access data. Table 1 summarizes the notations used throughout the paper. Particularly, XML nodes covered by positive rules and not covered by any negative rule are considered *safe data*. XML query that

Table 1 Notations

Term	Meaning
Q	User's input query in XPath
Q'	Re-written query from Q
D	XML document
$Q(D)$	Answers of evaluating Q against D
SQ/SD	Safe XML query/ safe document
UA/UD	Un-safe XML answer/un-safe document
R	A 4-tuple access control rule
R^+/R^-	R that has sign $+/-$, respectively
ACR	$\{R_i\}$, set of access control rules
ACR^+/ACR^-	All $\{R^+\}$ / $\{R^-\}$ of the ACR

only requests safe data is called *safe query*; and the answer is *safe answer*.

Example 1 We use the XMark [48] schema (Fig. 1a) and access control rules of Table 2 for running examples. The schema demonstrates an online auction scenario. Rules R1 to R8 say that role1 is permitted to access "categories" information, some of "item" and "person" information. Initially, we only consider positive rules without predicates in the XPath of their *object* field. Then, R5′ is referred when we demonstrate how predicates are processed. R9 and R10 are added to discuss negative rules.

Currently, XPath handles operands of set operators as a sequence of node IDs. However, in the context of XML access control, the formal XML set operators are not sufficient. For instance, consider an XML tree with nodes <a> and their descendants <b>. Suppose there is a positive rule R_1: (admin, //a, read, +) and a negative rule R_2: (admin, //b, read, -). Conceptually, admin is granted to read "//a EXCEPT //b" (all nodes <a> and their descendants except <b> nodes and their descendants). However, since node IDs for <a> and <b> cannot be identical, the result of "//a EXCEPT //b" is always "//a". When <b> is a descendent of //a, the answer violates the specified access control rules since R_2 blocks <b>. In other words, the standard semantics of XML set operators do not ensure correct XML access control. To remedy these shortcomings, in [36], we defined *deep set* operators with extended semantics: DEEP-EXCEPT, DEEP-UNION, and DEEP-INTERSECT, denoted as $\overset{D}{-}$, $\overset{D}{\cup}$, and $\overset{D}{\cap}$, respectively. Semantics of deep set operators are illustrated in Fig. 1b. For instance, the semantics for "P1 DEEP-EXCEPT P2" are as follows: (1) when P2 nodes are descendants of P1, subtrees corresponding P2 are pruned from P1 and the remainders are returned; (2) when some P1 nodes are descendants of P2, nodes covered by P2 are eliminated from the answer; and (3) otherwise, it degenerates to the regular except operation, i.e., "P1 EXCEPT P2". In our experimentations, deep set operators are implemented as user-defined functions of XQuery, which does not require any extra support from underlying

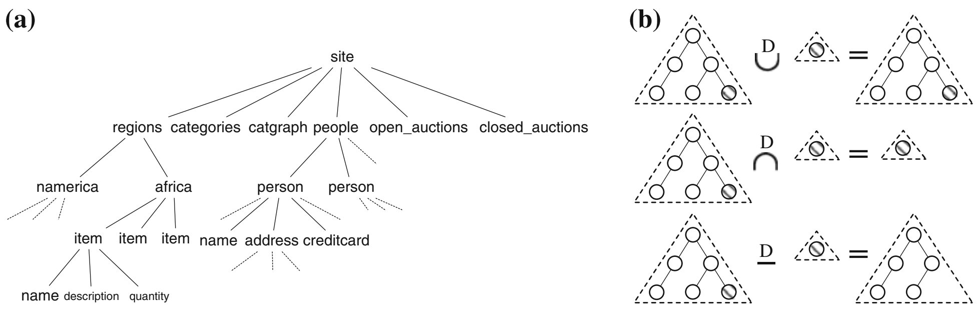


Fig. 1 **a** The XMark DTD; **b** Deep set operators

Table 2 Example rules

```
R1: (role1, /site/categories, read, +)
R2: (role1, /site/regions/*/item/location, read, +)
R3: (role1, /site/regions/*/item/quantity, read, +)
R4: (role1, /site/regions/*/item/name, read, +)
R5: (role1, /site/regions/*/item/description, read, +)
R5′: (role1, /site/regions/*/item[quantity>0]/location, read, +)
R6: (role1, /site/people/person/name, read, +)
R7: (role1, /site/people/person/address, read, +)
R8: (role1, /site/people/person/emailaddress, read, +)
R9: (role1, /site/regions/asia/item/location, read, -)
R10: (role1, /site/regions/africa/item/location, read, -)
```

XML engine. Detailed descriptions and implementations of deep set operators can be found in [36].

3 XML access control enforcement mechanisms

Our goal is to devise practical XML access control mechanisms without using security features from underlying DBMS. Given ACR and Q, a desirable XML access control mechanism would answer Q with only "safe data". In particular, we consider three approaches: (1) **Primitive**: ACR is merged to the query Q to yield a new query $Q' = Q \overset{D}{\cap} ACR$, and the deep set operators are processed by DBMS; (2) **Pre-processing**: Parts of the Q that conflicts with ACR are pre-pruned to yield Q', which is processed by DBMS as usual to return safe answers; (3) **Post-processing**: Q is processed by DBMS to produce unsafe answers, which goes through post-filtering process to prune out the parts that violate ACR, and return only safe parts.

Figure 2 illustrates the current practice of XML query processing (i.e., without access control) and the three approaches described above, using Galax [49] as the XML database. We introduce an abstract mechanism AFilter for post-processing of answers. Later we will show that, with some modification, YFilter [17] could be used as an implementation of AFilter . We will first introduce the three approaches briefly and go into details of the pre-processing approach.

3.1 Primitive approach

The idea of the primitive approach is to view both query and security policies written in ACR as *constraints to satisfy*. Therefore, security is enforced by "merging" two constraints to form tighter constraints. For instance, in Example 1, consider user "John" of *role1*, who surveys the items' location information with a query `Q://item/location`. The meta-semantics of Q and a positive rule $R+$ is that users are allowed to access the regions scoped by "`Q DEEP-INTERSECT R+`". Conversely, that of Q and a negative rule $R-$ is "`Q DEEP-EXCEPT R-`". Collectively, John is allowed to read:

```
 (Q DEEP-INTERSECT (R1 DEEP-UNION R2...
R8))
DEEP-EXCEPT (R9 DEEP-UNION R10)
```

Note that only `R2`, `R9` and `R10` are related to John's query. However, the primitive approach does not analyze the *object* field of rules to distinguish related rules. On the other hand, the primitive approach is built on deep set operators; hence, it does not require any security support from the underlying XML engine (deep set operators are implemented as user-defined functions). The semantics and algorithm of the primitive approach is simple and clear and thus can be easily implemented. However, the primitive approach may generate complex safe queries that are expensive to evaluate, especially when there are a large number of access control rules.

3.2 Pre-processing approach

One may improve the primitive algorithm by further optimizing the safe query. That is, instead of simply generating a complicated Q' with multiple deep set operators interweaved, one may "pre-process" it by exploiting the specifics of XML model and access controls. If what users ask for are entirely prevented by ACR, we can reject the query outright. Similarly, if users ask for data that are entirely granted, no further security check is needed. Lastly, if users ask for partly accessible data, it is beneficial to rewrite Q such that fragments asking for illegal data are pruned. With deep set operators, pre-processing approach is described as:

$$SA = SQ(D)$$
$$SQ = Q \overset{D}{\cap} [(R_1^+ \overset{D}{\cup} R_2^+ \overset{D}{\cup} \cdots \overset{D}{\cup} R_m^+) \overset{D}{-} (R_1^- \overset{D}{\cup} \cdots \overset{D}{\cup} R_n^-)]$$

3.3 Post-processing approach

The post-processing approach 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 unsafe data till the last step, this approach is simple to implement. Moreover, when ACR and data are stored separately in a distributed environment (e.g., database-as-a-service model), this approach can be useful. Post-processing approach is described as follows.

$$\begin{aligned} SA &= ACR(UA) \\ &= ACR(Q(D)) \\ &= [(R_1^+ \overset{D}{\cup} \cdots \overset{D}{\cup} R_m^+) \overset{D}{-} (R_1^- \overset{D}{\cup} \cdots \overset{D}{\cup} R_n^-)](Q(D)) \end{aligned}$$

As shown in Fig. 2d, the AFilter is used to process $ACR(UA)$, i.e., to extract authorized XML nodes from the intermediate unsafe answer. Hence, it is somewhat similar to an XML query processor. In practice, AFilter could be implemented in different ways. In our experiments, we adopt YFilter [17], an query processor for streaming XML data, as an implementation of AFilter.

However, despite the simple look on the surface, its implementation needs to overcome a technical issue. Let us again look at John's query `Q: //item/location`. `R9` and `R10` disallow John to access location information of Asia or Africa items. When Q is first evaluated on an XML document, it projects out only the `<location>` nodes without any ancestors. Therefore, in post-filtering, when `R9` and `R10` are to be enforced against

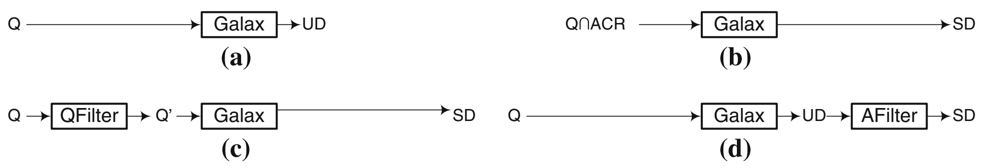


Fig. 2 Ways to support XML access control without using security features of DBMS: **a** no presence of access control mechanisms; **b** primitive approach; **c** pre-processing approach; and **d** post-processing approach

such intermediate answers rooting at `<location>` nodes, they cannot check whether the `<location>` satisfies `/site/regions/africa/item/location` or not. However, if the underlying XML database can produce `<location>` as well as all its "ancestor" tags (e.g., using a recursive function of XQuery), then the post-processing approach by AFilter can be applied without any further security support from databases. Finally, to produce correct XML answers, the extra ancestor tags need to be removed after AFilter is applied.

4 QFilter: an implementation of pre-processing approach

In this Section, we present our NFA-based implementation of the pre-processing approach, named QFilter. QFilter reads a query Q and access control rules ACR as input and returns a modified safe query Q' as output:

$$Q' = \text{QFilter}(Q, ACR) \tag{1}$$

This can be re-written by separating positive and negative rules as follows:

$$Q' = \text{QFilter}(Q, ACR^+) \overset{D}{-} \text{QFilter}(Q, ACR^-) \tag{2}$$

That is, the **Pre-Processing** approach can be implemented by two invocations of QFilter function and a `DEEP-EXCEPT` operator. Two stages of QFilter, construction and execution, are elaborated in this section. Let us take a bird's eye view of QFilter before going into details. A QFilter is constructed from ACR. At runtime, it "filters" out illegal fragments from incoming queries to produce only "safe" queries. In the filtering stage, QFilter has three types of operations: (1) **Accept**: If answers of Q are contained by that of ACR^+ (i.e., Q asks for nodes granted by ACR^+) and disjoint from that of ACR^- (i.e., Q does not ask for nodes 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 fully contained by that of ACR^- (i.e., all answers to Q are blocked by ACR^-), then QFilter rejects the query outright: $Q' = \emptyset$; and (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'.

Example 2 In Example 1, a user submits three queries:

```
Q1:/site/categories//*
Q2:/site/regions/asia//location
Q3:/site/people/person/*
```

When ACR of Table 2 is enforced on these queries:

(1) `Q1` is accepted by `R1`;
(2) `Q2` is accepted by `R2` but rejected by `R9` and is finally rejected since negative rules override positive rules;
(3) `Q3` is rewritten by `R6`, `R7`, and `R8` into:
`/site/people/person/name` $\overset{D}{\cup}$
`/site/people/person/address` $\overset{D}{\cup}$
`/site/people/person/emailaddress`.

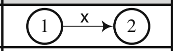
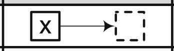
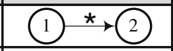
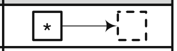
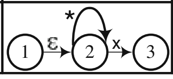
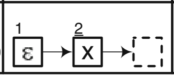
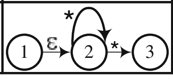
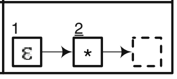
4.1 QFilter construction

In a nutshell, QFilter builds an non-deterministic finite automata (NFA) from *Object* fields (in the form of XPath expressions) of ACR and processes an input query Q according to one of the three operations. More specifically, we are constructing a special type of NFA—a Mealy Machine [37]. In a Mealy Machine, an output is generated at each automata state, and the output is determined by the current state and the input.

In QFilter construction, we first take XPath expressions from all the positive rules for a particular role (ACR^+) to construct a "positive QFilter". To tokenize XPath expressions, we view them as compositions of four basic building blocks: `/x`, `/*`, `//x`, and `//*`. The NFA element for each building block is shown in Fig. 3. XPath expressions with predicates are further described in Sect. 4.3. For a complete XPath, NFA fragments are constructed for path elements and then linked in sequence. For a set of rules that form the ACR, NFA fragment sequence for each rule is constructed and all of them are combined such that identical states are merged. The construction process is similar to that of regular NFA. For instance, Fig. 4a shows the state transition map for ACR^+ in Example 1, and Fig. 4b shows the corresponding NFA.

In our implementation, the QFilter NFA holds a state transition table at each state, mapping acceptable *tokens* (element names of XPath steps) to transition *states*. Moreover, the predicates are also captured in QFilter. The data structure

Fig. 3 NFA element for each XPath building block

Element	State transition	NFA contruct	Element	State transition	NFA contruct
/x			/*		
//x			//*		

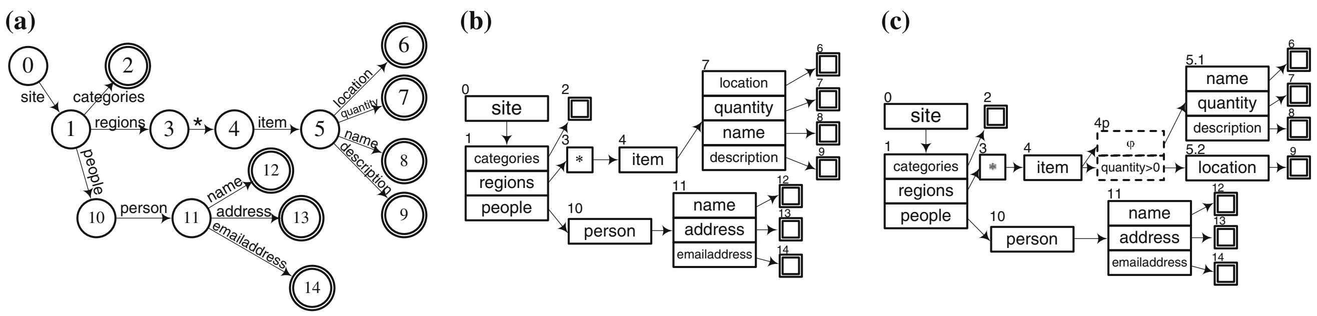


Fig. 4 **a** State transition map of the QFilter; **b** NFA of the QFilter; **c** NFA of the QFilter with predicate processing states

for QFilter (illustrated in Fig. 5) consists of: (1) A state transition table ($stateTransitionTable$) maps acceptable tokens to a $predicateTable$, which maps predicates to corresponding child $QFilterState$; (2) A ε-transition child state (may be null), to handle "//" steps; (3) A binary flag to indicate accept state (e.g., state 6 of Fig. 4); and (4) A binary flag to indicate a "//" state, which recursively accepts tokens, i.e., with a * transition to itself (e.g., state 2 in Fig. 3 row 2).

Taking ACR as input, we construct QFilter from the root state and hold this state for all future access (e.g., add a rule or filter a query). We first create an empty root state and then add each rule to the root state one by one. Algorithm 1 shows QFilter construction (at state level) in details. The general idea for adding each rule is to follow the existing NFA states as much as possible, until no existing state is reusable, and then new states are created. At each state, the $addRule()$ function takes in the current XPath step (token). In case we reach the end of the XPath, the current state is marked as accept state and the rule is added (lines 1–3). In the case of // steps, we first add ϵ-states and then process as regular steps (lines 4–10). If the token is not in the state transition table, we create a new entry for it (lines 14–17, we process * separately). Similarly, if the predicate (may be NULL) is not in the predicate table, we also add a new entry and a pointer to the next state (lines 19–23). And then we move to the next token.

Algorithm 1: QFilterState.addRule

```
   Input: XPath expression of Access Control Rule: $R$
1  if $R.EOS()$ then
2      mark as accept state: $acceptState = True$;
3      return
4  if $(R.currentStep\ is\ “//”)\ \&\ (NOT\ R.doubleSlashProcessed)$
   then
5      if $\varepsilon - transitionChild$ does not exist then
6          Create $\varepsilon - transitionChildState$;
7          mark $\varepsilon - transitionChildState$ as $DSState$;
8      $R.doubleSlashProcessed \longleftarrow true$;
9      $\varepsilon - transitionChildState.addRule(R)$;
10     return
11 $Token \longleftarrow R.elementName$;
12 $Predicate \longleftarrow R.predicate$;
13 $R.nextStep()$;
14 if DSState & $Token = “*''$ then
15     addRule(rule); return;
16 if NOT $stateTransitionTable.hasKey(Token)$ then
17     $stateTransitionTable.put(Token, emptyPredicateTable)$;
18 $predicateTable \longleftarrow stateTransitionTable.get(Token)$;
19 if NOT $predicateTable.hasKey(Predicate)$ then
20     create new filterState $newState$;
21     $predicateTable.put(Predicate, newState)$ ;
22     $stateTransitionTable.put(Token, predicateTable)$;
23     $newState.addRule(R)$;
24 else
25     $(PredicateTable.get(predicate)).addRule(R)$;
```

Example 3 Let us demonstrate the construction of QFilter for ACR^+ in Example 1. Construction starts from /site/categories. State 0 in Fig. 4 is first created for the XPath step /site. Then, state 1 is created for the step /categories, and state 2 is created and marked as "*Accept*". At this time, the QFilter containing 3 states are shown in Fig. 5b. Next, we add another rule R2: /site/regions/*/item/location. For the first step "/site", since identical key is detected at state 0, it is reused. Then at state 1, element name "regions" is not in the state transition table. State 3 is created, and a new entry is inserted into the state transition table at state 1. Subsequently, states 4, 5, and 6 are created in the same way. Finally, after we add all eight rules in ACR^+ to this QFilter, state 1 is constructed as shown in Fig. 5c.

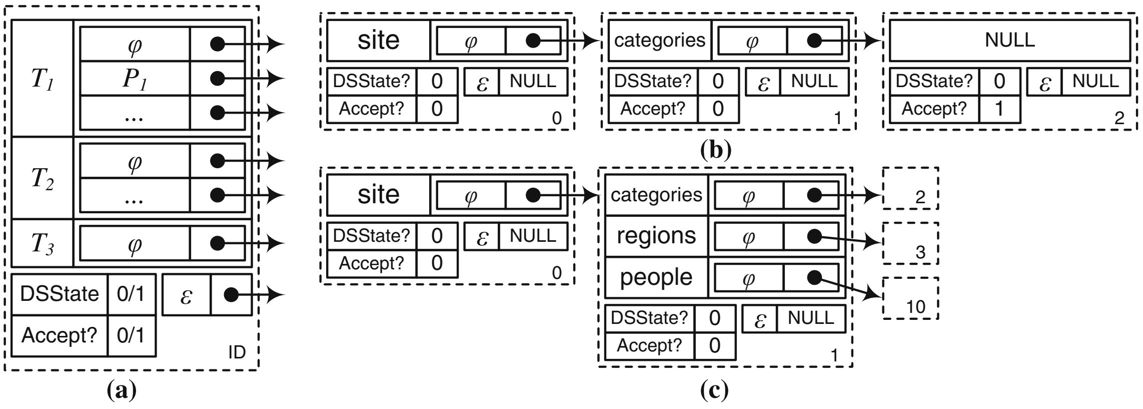


Fig. 5 **a** Data structure of a QFilter state; **b** QFilter constructed for rule `/site/categories`; **c** QFilter constructed for more rules

4.2 QFilter execution

Given a query Q as input, QFilter prunes unsafe fragments of Q to generate a safe 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 outright; and (3) Otherwise, modify Q such that Q' returns a precise "deep-intersection" of Q and ACR. The filtering process becomes complicated when either Q or ACR has non-deterministic operators such as "// " and "*".

At microlevel, we first split Q into XPath steps, tokenize them, and pass to the root state of QFilter to start NFA (Mealy Machine) execution. As a Mealy Machine, each state generates an output, which is determined by the input token and the state transition. A query is accepted when it reaches an accept state, and the intermediate outputs are concatenated to generate a safe query as NFA output. Queries with wildcards may go through several rules (being rewritten by each rule), and the result of QFilter execution becomes an array of safe queries. Each element in the array reflects a rewritten branch of Q. Finally, QFilter weaves the array of XPath queries through using $\overset{D}{\cup}$. The details of QFilter execution, shown in Algorithm 2, are as follows:

- At each state, the (tokenized) element name from Q is matched against the keys in the state transition table. When a "match" is found, we keep the intersection of the element name and the key as the output of this state. For instance, when "*" matches "`regions`", their intersection "`regions`" becomes the output. Predicates from ACR and Q are both kept in the output (details in Sect. 4.3).
- When a "match" is processed, QFilter locates the corresponding "next state" from the state transition table and continues with the next XPath step from Q as input.
- Q is accepted at the accept state. We then link the output of each state sequentially to obtain a final "filtered" output query.
- At each step, multiple matches may exist (e.g., a "*" in Q matches all the keys in the state transition table). Then, QFilter execution splits into branches, and the final output of each branch (if not null) is put into the result array. On the other hand, when multiple predicates exist, QFilter execution is also split into branches. Finally, multiple outputs (in the array) are connected by $\overset{D}{\cup}$.

Example 4 Let us use the QFilter in Example 2 (Fig. 4b) to check query Q: `/site/people/person/name`. At state 0, since the first token "`site`" matches the key "site", "`/site`" becomes the output, and execution continues to state 1. In this way, Q goes through states 0, 1, 10, and 11 and is finally accepted at state 12 as "`/site/people/person/name`".

Example 5 We process Q: `/*/*/person/name` with the same QFilter. The first `/*` is accepted by state 0, and the output is "`/site`". The next `/*` is accepted by state 1, and the execution continues into 3 branches: (1) to state 2, with output "`/site/categories`"; (2) to state 3, as "`/site/regions`"; and (3) to state 10, as "`/site/people`". Branch 1 is accepted at state 2 as "`/site/categories/person/name`". Note that this query is not valid in XMark schema and will be rejected at query processing (static analysis phase). In Sect. 4.7, we will further discuss how QFilter could be combined with XML schema. Branch 2 goes through state 3 and is rejected at state 4, since "`name`" does not match anything in the state transition table. Finally, branch 3 goes through states 10 and 11 and is accepted at state 12, as "`/site/people/person/name`".

4.3 Handling predicates

Predicates such as "`[b=10]`" in "`//a/[b=10]/c`" frequently occur in Q or ACR. When Q has predicates in it, they are kept intact initially. Whenever an XPath step is accepted

Algorithm 2: QFilterState.filter

```
Input: XPath query $Q$ (with pointer to current step); its filtered
       part: prefix
Output: String array of filtered query branches: $arrayQ'$
if $Q.EOS()$ then
    if is accept state then
        return $prefix$;
    else
        return NULL;
if $Q$ is double slash then
    $arrayQ' \longleftarrow QFilterState.DSFilter()$;
    return $arrayQ'$;
if $\varepsilon - transitionChildState!{=}NULL$ then
    $Q' \longleftarrow \varepsilon - transitionChildState.filter(Q)$;
    if $Q'$ != NULL then
        $arrayQ'.insert(Q')$;
$Token \longleftarrow Q.elementName$;
$Predicate \longleftarrow Q.predicate$;
$Q.nextStep()$ ;
if current state is a DSState then
    $prefix' \longleftarrow prefix$+"/"+Token+Predicate;
    $Q' \longleftarrow filter(Q, prefix')$;
    if $Q'$ != NULL then
        $arrayQ'.insert(Q')$;
foreach match between Token and (key[i] in
stateTransitionTable) do
    if key[i]="*" then
        $Token' \longleftarrow Token$;
    else
        $Token' \longleftarrow key[i]$;
    $predicateTable \longleftarrow stateTransitionTable.get(key[i])$;
    foreach predicate[i] in predicateTable do
        $prefix' \longleftarrow$
        $prefix + "/" + Token' + Predicate + predicate[i]$;
        $nextState \longleftarrow predicateTable.get(predicate[i])$;
        $Q' \longleftarrow nextState.filter(Q, prefix')$;
        if $Q'$ != NULL then
            $arrayQ'.insert(Q')$;
```

or re-written, then the predicate (if any) is attached to it. Otherwise, if a path is rejected, the predicate is also rejected. For predicates in ACR, from Fig. 5a, we can see that each element name is mapped to a table, holding all the predicates affixed with it. During QFilter execution, when the input matches a token in the state transition table, we further process the predicates: (1) ACR predicate in the predicate table is attached to the output of the current state; (2) Multiple entries in the predicate table are not exclusive. That is, QFilter execution is split into multiple branches, and each takes an entry in the predicate table; and (3) For each branch, QFilter execution continues at the "next state" corresponding to the entry in the predicate table.

Example 6 Let us replace R5 of Example 1 by R5' in Table 2. Then, the QFilter of Fig. 4b is re-constructed to Fig. 4c. Each non-leaf non-ε state carries an empty predicate processing state "φ" but omitted for simplicity.

Example 7 Let us use the QFilter of Example 6 and Fig. 4c to process Q: "`/site/regions/namerica /item/location`". Q first goes through states 0, 1, 3, and 4, with an intermediate output as "`/site/regions/namerica/item`" (Note that, at state 3, "*" in the transition table matches with the current element name "namerica", and their intersection, "namerica", is kept in the output). Then, QFilter execution splits into two branches at predicate state 4p. Branch 1 (intermediate output: "`/site/regions/namerica/item`") goes to state 5.1, while branch 2 goes to 5.2, with intermediate output: "`/site/regions/namerica/item[quantity>0]`" (predicate from rule `R5'` is attached). Branch 1 is rejected at state 5.1, while branch 2 is finally accepted at state 9. Finally, the QFilter output is `/site/regions/namerica/item[description]/name/text()`

4.4 Handling queries with //

The descendant-and-self axis in XPath ("`//x`") asks for element "`x`" with any path(s) preceding it. The `//x` step (respectively, `//*` step) in Q matches the key `x` (respectively, any key) in the current NFA state, or any of its descendant state. Therefore, either "`//x`" or "`//*`" in Q triggers the state transition from the current state to all of its subsequent states and then matches "`x`" or "`*`" with keys in their state transition table. In this case, Q is split into branches that continue at each of the subsequent states of the current state (where the "`//`" input is detected). Such a query needs to be rewritten. In general, we rewrite "// " with the path from the current state to the destination state, where the branch continues to be executed. Then, each branch of the QFilter execution restarts by matching the input element name ("`x`" or "`*`") with keys in the state transition table.

Example 8 Let us use the aforementioned QFilter to process the query "`/site/people //name`". The first two steps "`/site/people`" trigger the state transition from 0 → 1 → 12. Then, when it encounters the "//", Q breaks into the following six branches, each having the input element "name": (1) `/site/people` restarts at state 12; (2) `/site/people/person` restarts at state 13; (3) `/site/people/person/name` restarts at state 14; (4) `/site/people/person/address` restarts at state 15; (5) `/site/people/person/emailaddress` restarts at state 16; and (6) `/site/people/person/address/` restarts at state (17/18). Obviously, only the states 13 (branch 2) and 17/18 (branch 6) can accept the input token `name`. Thus, the final output is "`/site/people/person/name UNION /site/people/person/address //name`".

Table 3 “// ” transition look-up table

Start	Destination	Re-written query
	10	
	11	`/person`
10	12	`/person/name`
	13	`/person/address`
	14	`/person/emailaddress`

To speed up the traversal, we can build a look-up table for each state. It is an index to all the sub-states, together with a string to rewrite “//” into a safe path. As an example, the look-up table of state 12 is shown in Table 3. On the other hand, a “`//`” state in query triggers multiple matches in QFilter, yielding extra overhead. However, note that the computation is not as excessive as it looks like. For a “`//x`” query, most of the redundant branches are rejected at the next state, since `x` does not match anything in the state transition table. On the other hand, “`//*`” states are not commonly used in real-world queries. Moreover, they should only appear at the end of XPath queries. Otherwise, a “`//*/x`” state can be easily written into `//x`.

4.5 Handling negative rules

Unlike engine-level or view-based methods, pre-processing XML access control approaches need special handling for negative rules. However, as we have shown, traditional “except” operations defined in XPath is not sufficient, since it is not capable of creating new nodes. In our approach, shown in Eq. 2, for ACR set with negative rules, a separate QFilter$^-$ is built. Hereafter, we call the QFilter for positive rules the *positive QFilter* and the QFilter for negative rules the *negative QFilter*. The construction and execution of the negative QFilter is the same as those of the positive QFilter (minor differences in operations at accept states will be discussed in next subsection). The output from both QFilters are connected by the deep-except operator, as we have described in Sect. 2.1. As we have implemented deep set operators through XQuery user-defined-functions (UDF), the new query could be answered by any state-of-art XML engine that supports XQuery. In our implementation, for better performance, instead of passing the original query through the negative QFilter , we use the output from positive QFilter as the input to the negative QFilter. Therefore, we have:

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

In this way, input to negative QFilter has been pruned by positive QFilter, so that they contain fewer wildcards. Therefore, sub-queries are more likely to be dropped at negative QFilter, or the result safe queries require less computation in query evaluation. For instance, if we have rules `{user, /a//*, read, +}`, `{user, //c, read, -}`, and query `/*/c`, our implementation drops the query without evaluation, instead of returning `/a/c` $\overset{D}{-}$ `/*/c` (this query will yield NULL eventually).

Now we analyze the properties of negative rules,and show that not all negative rules require deep set semantics. In practice, a good portion of negative rules could be handled by regular except semantics. As we described in Sect. 2.1, deep-except operator is used to remove inaccessible nodes from the answer. Recall that, in our XML access control model, all nodes are inaccessible by default. When a user is prohibited to access a node, there is no need to write a negative rule (R^-) to revoke its accessibility, unless the node has been granted access by positive rules (ACR^+). In this way, negative rules are only used to specify exceptions to global permissions, i.e., “revoke” access proposed by ACR^+. Deep-except operator is used to enforce such *revoke* operation. However, it depends upon the type of the negative rules whether we need to use *deep except* or regular *except*. In particular, we distinguish two types of negative rules:

Definition 1 (*NE vs. DE negative rules*) A negative rule in ACR restricts users from accessing a set of nodes $\{r_1^-, \ldots r_n^-\}$. If **none** of the nodes is a descendant of the context node of a positive rule, i.e.:

$$r_i^- \notin \langle R^+//*\rangle, \quad \forall r_i^- \in \{r_1^-, \ldots r_n^-\}; \forall \langle R^+\rangle \in \langle ACR^+\rangle$$

then it is called a **Node Elimination** (**NE**) negative rule. Else, if one of the nodes is a descendant of the context node of a positive rule, i.e.:

$$r_i^- \in \langle R^+//*\rangle, \quad \exists r_i^- \in \{r_1^-, \ldots r_n^-\}; \exists \langle R^+\rangle \in \langle ACR^+\rangle$$

it is called a **Descendant Elimination** (**DE**) negative rule.

Intuitively, a “*Node elimination*” (NE) negative rule removes context node from $\langle ACR^+\rangle$, while a “*descendant elimination*” (DE) negative rule removes descendants from context node of $\langle ACR^+\rangle$.

Example 9 Let us revisit our example ACR in Table 2. Positive rule `R2` grants users access to `<location>` nodes. Correspondingly, negative rule `R9` revokes access to some of these `<location>` nodes. Since `R9` revokes access to context nodes of `R2`, it is an *NE negative rule*. On the other hand, assume we add another negative rule: `Rx: (role1,/site/people/person/address/zipcode,read,-)`. In this way, `R7` grants access to `<address>` nodes, while `Rx` revokes access to one of its children `<zipcode>`. Hence, `Rx` removes descendants from context node of `R7` and therefore is an *DE negative rule*.

In [34], we show that deep except is only needed when answers are partially blocked by DE negative rules, in which descendants of the original context nodes are removed. Therefore, deep-except concept needs to be employed to construct new XML nodes.

4.6 Accept state operations and QFilter output

The notions of *answer-by-node* and *answer-by-subtree* have brought extra burdens in access control design and implementation. It also caused confusion and misunderstanding. In this work, we follow XML and XPath specifications [2] to implement the *answer-by-subtree* model: a node has everything between the starting and closing tags. In the tree concept, an XML node includes the subtree with all its descendant nodes. Hence, we inherently employ *recursive check*: granting access to a node is effective to the entire subtree. In this way, our approach could be seamlessly adopted by any XML database systems and access control models that comply with the XPath standard.

Operations at accept states are more complicated than regular states, partly due to the *answer-by-subtree* model. As shown in Sect. 2.2, access control rules that only applies to text child of the context node are called `text()`-only rules, in contrast to regular rules. In QFilter, we have two types of accept states: regular accept states and "`text()`-only" accept states. In implementation, they are distinguished by a flag.

1. Positive QFilter
   - (a) Query ends at an accept state: the query is accepted. All the outputs from its path in the QFilter are concatenated to give the final output. A special case is when the accept state is a "`text()`-only" accept state, then "`/text()`" is appended to the end of the output XPath expression.
   - (b) Query exceeds an accept state: in this case, the query reaches an accept state, but there are more XPath steps yet to be processed by the NFA. For instance, when ACR allows `/a` and query is `/*/*/c`, the query reaches an accept state after its first XPath step is accepted by QFilter. When it is a regular accept state, the query is accepted since it asks for a subtree of an accessible node. Outputs from previous QFilter states are concatenated, and the unprocessed XPath steps from the query are appended to construct the final output. In the example, output from previous QFilter states is `/a` and residues from the query is `/*/c`, thus making the full output: `/a/*/c`. On the other hand, when the accept state is a "`text()`-only" accept state, the query is rejected unless its remaining part is also "`text()`".
   - (c) Query ends at middle states: we encounter a designing choice for those queries that stop at non-accept states. In traditional automata theory, such strings are rejected. However, in our application, such scenario happens when a query asks for a node, while the user is only allowed to access a portion of the subtree. For instance, user asks for `/a`, while he only has access to `/a/b/c`. In such cases, it is a design choice whether to reject the query or to return accessible portion of the answer (e.g., /a/b/c).
2. Negative QFilter

   As shown in Sect. 4.5, with the presence of negative rules, accepted sub-queries from the positive QFilter are further processed by the negative QFilter. Here, we use the word "accept" from NFA perspective. Note that a query accepted by negative QFilter indicates that it is to be denied access (i.e., "rejected" from access control perspective)
   - (a) Query ends at an accept state: the query is accepted, and the output is generated in the same way as positive QFilter.
   - (b) Query exceeds an accept state: when it is a regular accept state, the query is accepted. The output is generated in the same way as positive QFilter. When the accept state is a "`text()`-only" accept state, the query is rejected unless the remaining part is also "`text()`".
   - (c) Query ends at middle states: in this case, access to some descendants of the answer nodes is prohibited by negative rules. We automatically follow the QFilter to all the descendent accept states and generate the complete outputs. Note that this is a clear case of *descendant elimination*; thus, deep-except operator will be employed to remove the inaccessible nodes.

4.7 QFilter with XML schema

Until now, we have made no assumption on the availability of XML schema. We have not used XML schema since it may not be available with the access control enforcement mechanism, especially when access control is enforced outside of the data server, e.g., [33]. In this way, only ACR and Q are required for QFilter to enforce access control, making it independent and universally applicable. However, the disadvantage is that processing a wildcard query may result in many sub-queries that are not valid in the schema. For instance, running query "`//name`" through our example QFilter will yield a sub-query "`/site/people/`

person/address/name" which is not valid in XMark (<address> nodes do not have any <name> children). This does not mean that QFilter is wrong—such invalid sub-queries will be efficiently eliminated by the XML engine in the static analysis phase, so that they do not hurt the security or efficiency of the system. However, large amount of unnecessary sub-queries are annoying, especially when users intend to read the safe queries. Moreover, when QFilter is implemented outside the XML data server [33], such queries also bring communication overhead. Fortunately, we can easily replicate the XPath validation process in the XML engine to eliminate such queries. In our approach, we build an automata to capture the XML schema and make all the states accept states (each state represents an XPath expression that is valid under the schema). We validate all the sub-queries generated by the QFilter through the schema automata. A sub-query is valid if and only if it stops at one or more accept states. Therefore, with the schema automata, invalid subqueries, such as "/site/people/person/address/name", are dropped.

On the other hand, some follow-up works of the original QFilter paper [35] solves the problem by combining the schema with ACR and unfolds all wildcard queries to combinations of valid simple queries. This approach may not be the best since it may change the order of the nodes in the answer when it is not necessarily (note that XML is defined as ordered). This approach may create many piddling sub-queries for a wildcard query, when it is not necessary. For instance, with the XML schema, //person/* will be re-written into //person/name UNION //person/age UNION //person address UNION ....

4.8 Storage and computation

For a single QFilter, the (in memory) size is proportional to the number of NFA states, which is proportional to the number of rules in the ACR and number of XPath steps in each rule. [10] has shown that real- world DTDs are small. Meanwhile, when recursive XPath and predicates are not considered, the total number of possible rules (valid XPath expressions) for a schema is limited. On the other hand, since the rules are manually designed, it is expected that the size of ACR is relatively small. More importantly, there exists path sharing as we have described in QFilter construction, which reduces the size of the QFilter. In practice, an QFilter constructed from 50 (XMark) rules consumes approximately 40 KBs in our implementation.

For a system with multiple roles, we create one QFilter for each role. QFilters are identified with their root states and stored in an array. In extreme cases, when QFilters are employed in a web-scale application, there may be a huge number of roles and hence introduce a storage issue. To overcome this problem, we have introduced Multi-Role QFilter [33], which merges QFilters on the same schema (to intensively utilize path sharing) and attaches an access list (implemented as a bitmap) to each state. In this way, we save large amount of memory with the price of minimum run-time overhead.

Computational cost of QFilter includes time for both QFilter construction and execution. For QFilter construction, the complexity is $O(N)$, where N is the number of steps of XPath expression in ACR. For QFilter execution, (1) When there is no wildcard in Q, filtering Q costs $O(M)$, where M is the number of steps of Q. The worst case occurs when Q is accepted or rewritten; (2) When the wildcard "*" exists in Q, filtering costs $O(|NFA|)$. The worst case occurs when Q is "/*/*$\cdots$/*", since it requires the traversing of the entire NFA; (3) For Q with "// " step, the cost becomes $O(M * n_1 * n_2 * \cdots * n_k)$, where k is the number of wildcards "// " in Q and n_i is the size of the child QFilter at the state which first meets the i^{th} "// " path. Note that this is an acceptable cost since the worst case query of "//*//*$\cdots$//*" is rather rare in real-world XML queries. Overall, QFilter is computationally practical since the worst case query never occur. In the next section, we validate this claim in the experimentation.

4.9 Comparison with static analysis

As we have briefly discussed in Sect. 2.1, Static Analysis [41,42] is the most similar approach with QFilter. Now we compare both methods from a theoretical perspective.

An XML access control rule set (ACR) defines a set of *safe* XPath expressions. If we treat each ACR rule as a grammatical rule, the ACR set then defines a regular language $\mathcal{L}_{\mathcal{ACR}}$, in which each XPath step is a token, and each safe XPath is a valid word. In both approaches, ACR automata are used to capture $\mathcal{L}_{\mathcal{ACR}}$.

If we only consider simple path (without wildcards or predicates) and atomic nodes, such an XPath query is a word w_Q, which could be validated in L_{ACR}: Q is safe when $w_Q \in \mathcal{L}_{\mathcal{ACR}}$. However, the query may contain wildcards (e.g., /*/a), and the context node may have descendants (e.g., /b may have /b/c, /b/d, etc). Such a query represents multiple words on the same alphabet as $\mathcal{L}_{\mathcal{ACR}}$ and therefore also defines a language: $\mathcal{L}_Q$. In Sect. 3, we represent the safe query with deep set operators. Similarly, in Static Analysis, the safe query is represented as a new language: $\mathcal{L}'_Q = \mathcal{L}_Q \cap \mathcal{L}_{\mathcal{ACR}}$. However, to be evaluated against any XML engine, a valid XPath/XQuery expression needs to be derived from $\mathcal{L}'_Q$: (1) when $\mathcal{L}_Q \subseteq \mathcal{L}_{\mathcal{ACR}}$, $\mathcal{L}'_Q = \mathcal{L}_Q$. Since $\mathcal{L}_Q$ is deducted from Q, we can use Q as the safe query. (2) when $\mathcal{L}_Q \cap \mathcal{L}_{\mathcal{ACR}} = \mathtt{NULL}$, $\mathcal{L}'_Q = \mathtt{NULL}$ and access is denied. (3) finally, when $\mathcal{L}_Q \cap \mathcal{L}_{\mathcal{ACR}}! = \mathtt{NULL}$, we have a more general case that requires converting $\mathcal{L}_{Q'}$ back to XPath. A viable approach is to construct an NFA for $\mathcal{L}_{Q'}$, convert the NFA to a regular expression, and further translate it to comply with XPath. However, the conversion is expensive [26] and does not guarantee an optimal, human-readable XPath expression as output. Unfortunately, static analysis does not derive such a safe query and therefore requires run-time check from the underlying XML engine (to our best knowledge, no commercial XML DBMS currently delivers fine-grained access control function).

Compared with Static Analysis, we take a different path. Naturally, automata are used to check whether words (in this case, XPath queries) are valid under the grammar. We follow this route to use QFilter to check if the input is valid in $\mathcal{L}_{\mathcal{ACR}}$. In the case that the input word contains wildcards but $\mathcal{L}_{\mathcal{ACR}}$ accepts only specified pathes, we make use of the output function of the Mealy machine to rewrite it into a valid word in $\mathcal{L}_{\mathcal{ACR}}$. In summary, Static Analysis approach works with two languages, while QFilter uses one language to validate (and rewrite) input words (queries). Therefore, QFilter is capable of providing a relatively more human-friendly safe query in all cases and is theoretically more efficient. Note that, in some re-write cases of QFilter, access control decisions are statically indeterminable. Nevertheless, we are able to construct a safe query so that no security enforcement mechanism is required from the underlying XML engine. For instance, comparing a positive rule `//item[quantity>3]` and a user query `//item[name='laptop']`, although access control decisions cannot be made without accessing the data, we are able to construct Q'=`//item[quantity>3 and name='laptop']` as a safe query.

5 Experiments

To use QFilter for XML access control, we first constructs QFilter based on access control rules. Then, input queries are processed by QFilter to generate safe queries; and safe queries are sent to underlying XML engine to retrieve the answer. According to this process, we test QFilter in the following aspects: (1) QFilter construction, (2) QFilter execution, and (3) end-to-end query processing (including security check and evaluation). Since we have discussed the storage cost of QFilter in Sect. 4.8, the other major concerns of performance are computation efficiency. In this section, we focus on the evaluation of the speed of QFilter from the aforementioned three aspects.

5.1 Setup

We used the well-known XMark DTD and document generator [48] to generate synthetic XML documents. As a preprocessing approach, QFilter performance is irrelevant to size of XML documents, we use a 5 MB document for our experiments. We used Galax 0.3.1 [49] for XQuery/XPath query evaluation. 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 [17] from UC Berkeley as an implementation of AFilter (as shown in Fig. 2d). A YFilter is an NFA constructed from XPath queries in a way that is very similar with QFilter. However, the execution mechanisms are entirely different and hence introduce some differences in construction as well. For instance, YFilter captures and processes predicates in a different way than QFilter. As we have introduced, QFilter uses a special, non-reject NFA state to handle both value-based and structure-based wildcards. On the other hand, YFilter has developed an approach, namely *Inline*, to process value-based predicates when the host elements in the XML stream are matched with the XPath expressions in the NFA. Meanwhile, structure-based predicates are decomposed and constituent paths are matched separately; a collection operator is employed to handle the "join".

YFilter was originally introduced as an query processor for streaming XML data. It matches XML messages against users' queries (i.e., XML documents vs. XPath). In our case, YFilter is constructed with ACR and then takes intermediate unsafe XML answer as inputs and produces safe documents. Authors of [17] have helped us to modify YFilter so that it outputs only matched nodes (instead of the entire XML document) and hence serves the functionality of an AFilter.

The types of queries and number of access control rules are important in our experimentation and thus carefully selected and measured. Both user-defined (denoted as UD) and synthetic (denoted as SN) XPath expressions were used. Hence, 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. Note that user-defined queries over synthetic rules does not really make sense. Therefore, we only tested other three combinations. All synthetic XPath expressions were generated by YFilter [17] package. The Customer Advertisement Manager (CAM) role (extended from our running example) is created for user-defined ACR. CAM is in charge of delivering advertisements to customers thus is permitted to access items' and users' basic information except for credit card and user profile. This policy can be captured by the rules shown in Table 4.

In order to show the impact of predicates in ACR, we test both rules with and without predicates. Hereafter, we use "user-defined rules with predicate" to indicate rules shown in Table 4. On the other hand, we use "rules without predicates" indicates the remaining rules after "`[@quantity>0]`" fragment is removed from them. User-defined queries are mainly used to validate the correctness of QFilter. In addition, we also created queries with the synthetically generated XPath expressions as shown in Table 5 to evaluate the scalability. We have created 500 queries for each query set of Table 5.

Table 4 User-defined ACR: CAM case

No.	Rule
1	`(CAM, /site/regions/*/item[@quantity>0]/location, read, +)`
2	`(CAM, /site/regions/*/item[@quantity>0]/quantity, read, +)`
3	`(CAM, /site/regions/*/item[@quantity>0]/name, read, +)`
4	`(CAM, /site/regions/*/item[@quantity>0]/description, read, +)`
5	`(CAM, /site/categories, read, +)`
6	`(CAM, /site/people/person/*, read, +)`
7	`(CAM, /site/people/person/creditcard, read, +)`
8	`(CAM, /site/people/person/profile, read, +)`

Table 5 Synthetically generated 10 user query sets (QS1–QS10) with different probabilities of "*" and "// " at each XPath step, and total number of predicates in the XPath expression

QS	*	//	P	QS	*	//	P	QS	*	//	P	QS	*	//	P
								1	0	0	0	2	0	0	2
3	0	10%	0	4	10%	0	0	5	10%	10%	0	6	10%	10%	2
7	0	20%	0	8	20%	0	0	9	20%	20%	0	10	20%	20%	4

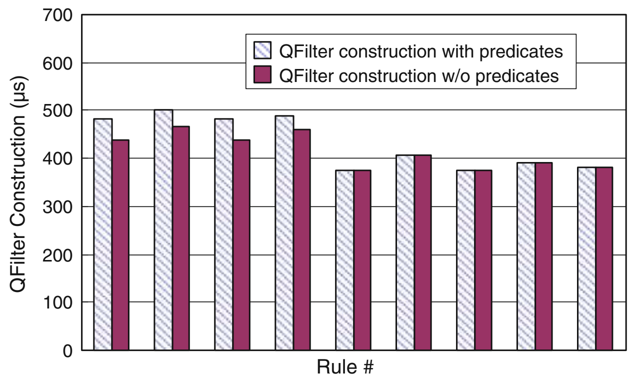


Fig. 6 QFilter construction using one single user-defined rule

5.2 Evaluating QFilter construction

In real-world applications, QFilter is likely to be constructed offline. Once the service starts, we do not need to modify or reconstruct QFilter unless ACR is changed. Thus, QFilter construction speed is of less importance to users. Nevertheless, experiments show that QFilter construction is fast enough, even to be done online.

We first construct QFilter with user-defined rules (eight rules for the role CAM, as shown in Table 4) and record the construction time. We construct QFilter using each of the rules with and without predicates and compare the speed. According to Fig. 6, QFilter construction time for different rules mainly depends on the complexity of the XPath expression, i.e., number of QFilter states to be built. QFilter construction is faster for shorter and simpler rules, since less parsing time is spent and less states are created. We also see that predicates bring more overhead to QFilter construction, since an additional predicate processing state is created.

Note that, in real-world applications, QFilters are not created for each individual rule. Rather, one QFilter is created for all the + rules and another QFilter for all the − rules. For CAM role, one QFilter for all the "+" rules is constructed in 1,155 μs, and one QFilter for all the "−" rules is constructed in 496 μs.

Next, we construct QFilter with synthetic rules and record the construction time. In each experiment, we generate 10, 50, 100, 200, 300, 400, 500, 600, 700, 800, 900, and 1,000 rules (distinct rules), respectively, each with the maximum length of 10 path elements. We use uniform distribution in selecting child elements. Different groups of synthetic rules are defined in Table 6.

Figure 7 shows that QFilter construction is fast and scalable. * or // paths do not slow down the construction. On the contrary, when we set higher * or // probability in rules, QFilter construction becomes faster. There are two reasons for this: (1) since XPath string parsing takes much of the QFilter construction time, the existence of * and // in the path makes the string shorter: as one path step, * is shorter than a string value path name; moreover, the XPath generator we used tends to generate shorter XPath expressions (with less steps) upon existence of //; and (2) in QFilter implementation, * and // paths are processed separately (not in the state transition hash table); thus, we do not search or insert the state transition table, which makes it faster.

For predicates, QFilter construction is faster with small number of predicates, because predicate states are

Table 6 Synthetically generated ACR with different probabilities of "*" and "// " at each XPath step and number of predicates

	RS	*	//	P	RS	*	//	P	RS	*	//	P	RS	*	//	P
(a)	1.1	0	0	0	1.2	10%	0	0	1.3	20%	0	0	1.4	30%	0	0
(b)	2.1	0	0	0	2.2	0	10%	0	2.3	0	20%	0	2.4	0	30%	0
(c)	3.1	0	0	0	3.2	0	0	1	3.3	0	0	2	3.4	0	0	3

(a) impact of * path; (b) impact of // path; and (c) impact of predicates

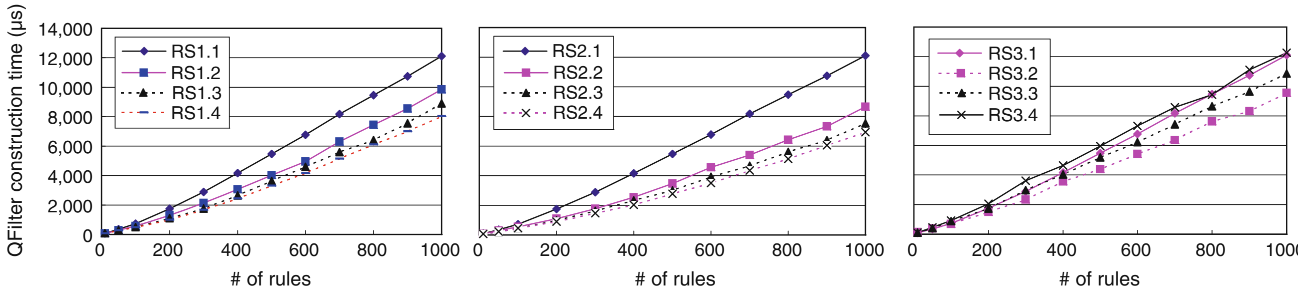


Fig. 7 QFilter construction using synthetic rules. From left to right: impact of (1) * path; (2) // path; (3) predicates

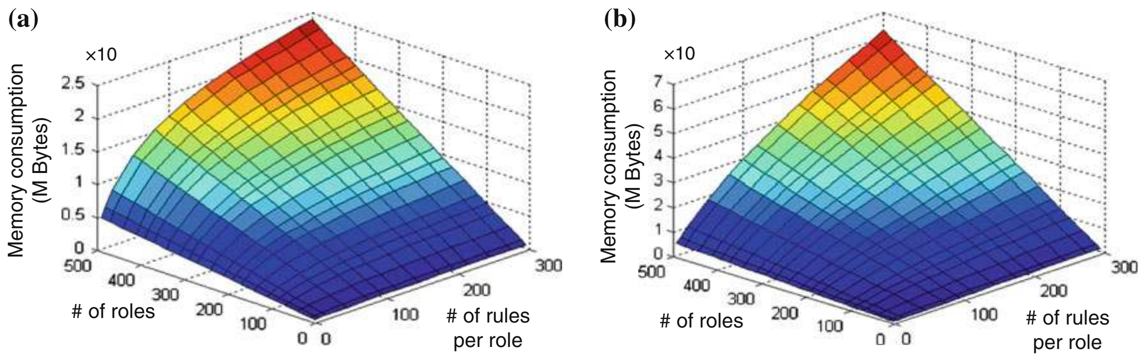


Fig. 8 Memory consumption of multiple QFilters: **a** ACR with no wildcard; **b** 10% probability of wildcard

constructed faster than regular NFA states. For XPath strings of similar length, those with predicates are processed faster. But many predicates (e.g., 3 predicates in RS3.4) may increase the length of the XPath strings and thus slow down QFilter construction.

As we have described in Sect. 4.1, in a system with multiple roles, multiple QFilters are constructed accordingly. In our implementation, they are hold in an array. As shown in Fig. 8, memory consumption of the QFilter array is proportion to the number of roles (i.e., number of QFilters) in the system. Although the actual memory consumption heavily depends on the actual implementation, the linear memory cost is very acceptable in most cases. In extreme cases, where there are a huge number of roles with many rules per role, or when the system memory is very limited, we have introduced Multi-role QFilter in [33] to further reduce memory consumption.

5.3 Evaluating QFilter 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 test how the properties of user query Q affect the filtering speed. That is, we prepare ten different query categories (as shown in Table 5), and for each category, we generated 500 synthetic queries based on the XMark DTD. Using these random XPath expressions as input to QFilter, we measure the number of accepted, denied, or rewritten queries in each group. We also separate a category "*minus*" to indicate the queries that are rewritten by negative rules. Then, we measure the average QFilter execution time for each group and for each output type (*accept*, *deny* and *rewrite*). The results are shown in Figs. 9 and 10.

From Fig. 9, we can summarize: (1) for rules without any predicate(left), queries in set 1 (no "*", no "// ") are either

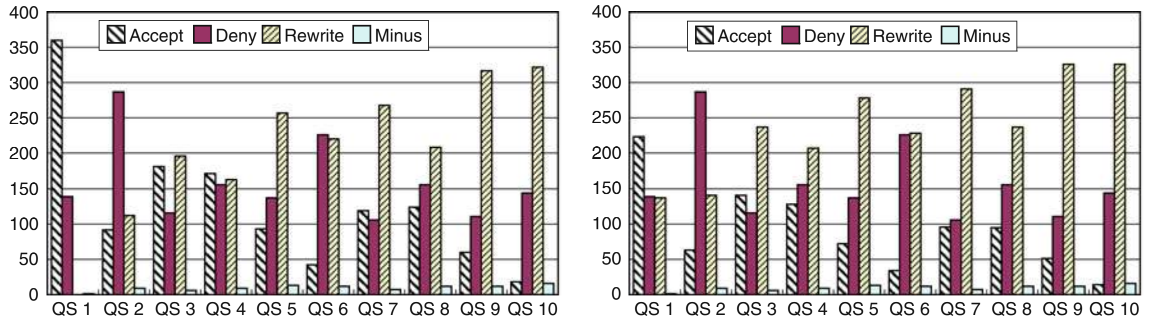


Fig. 9 Summary of QFilter outputs: number of *accept/reject/rewrite/minus rewrite* queries. *Left* rules without predicate; *right* rules with predicate(s)

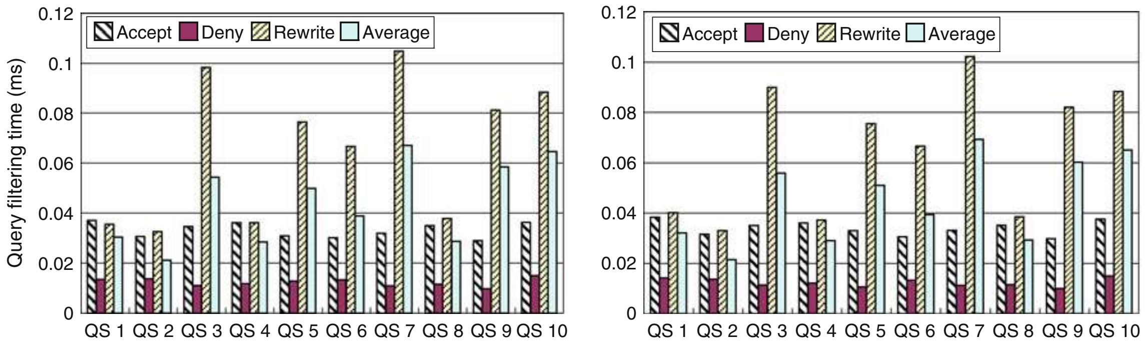


Fig. 10 QFilter execution speed for three types of outputs and their average. *Left* rules without predicate; *right* rules with predicate(s)

rejected or accepted, since there are no wildcards to be rewritten; for rules with predicates, they may be rewritten: predicates can be inserted; (2) queries with higher probability of wildcards "*" and "// " are more likely to be rewritten; (3) fewer queries in set 6 and 10 are rewritten than sets 5 and 9: existence of predicates in queries causes less regular path steps in each query; thus, these queries generally have a lower probability of "*" and "// "; (4) Comparing two figures, we can see that emergence of predicates in rules do not affect denied queries, but some originally accepted queries are rewritten (predicates are inserted).

Here, let us explain more about (2). Queries in sets 3–10 are generated with 10 or 20% probability of having "*" or "// " at each step. However, the probability does not automatically indicate that generated queries should have one or more "*" or "// " steps. When we manually look into the generated queries and the QFilter results, we found that some of these queries do not have any "*" or "// " steps, and most of them are either accepted or denied. From Fig. 10, we can summarize the following: (1) 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 "// "; and (2) Predicates in rules does not bring much overhead to QFilter execution. Average processing time is quite similar, and query rewriting time is even reduced, since some of the originally accepted queries are just rewritten at predicate state, which is faster than "*" amd "// " rewritten.

Next, we test how QFilter execution performance degrades as the number of rules in ACR increases. We constructed a QFilter using 20–500 synthetic rules based on XMark DTD (SN-ACR) and tested with random queries (SN-Q). We create two sets of rules as follows: `RS1` contains rules with no * path, no // path, and no predicates; and `RS2` contains rules with 10% * probability, 10% // probability, and 2 path-based predicate. On the other hand, we pick query sets 1, 2, 9, and 10 and then process them using QFilter with the above rules. Figure 11 shows the average QFilter execution time for each rule set By and large; as the number of rules in 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 500 synthetic rules, was still within only 10 millisecond.

5.4 End-to-end query processing and comparison

Finally, we compare the end-to-end processing time among four approaches of Fig. 2: (1) No security check is made (thus final data is un-safe); (2) Primitive approach; (3) AFilter; and (4) QFilter. End-to-end query processing time denotes the total time needed to process Q: from receipt of query until

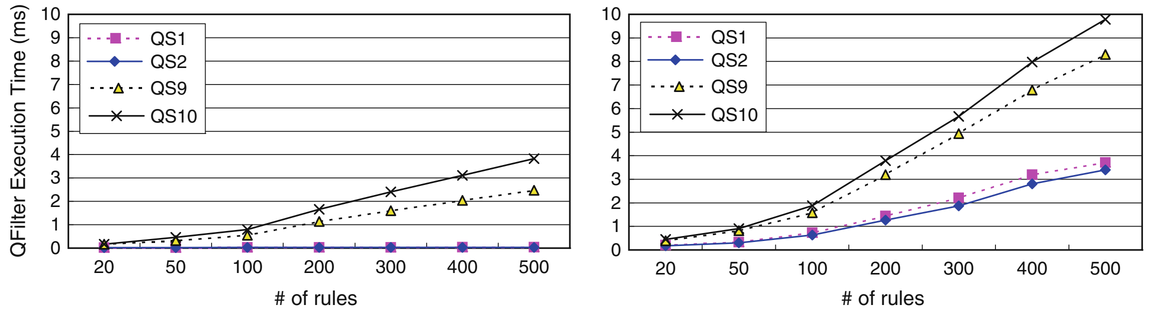


Fig. 11 QFilter execution time for synthetic rules and synthetic queries. *Left* rule set 1, *right* rule set 2

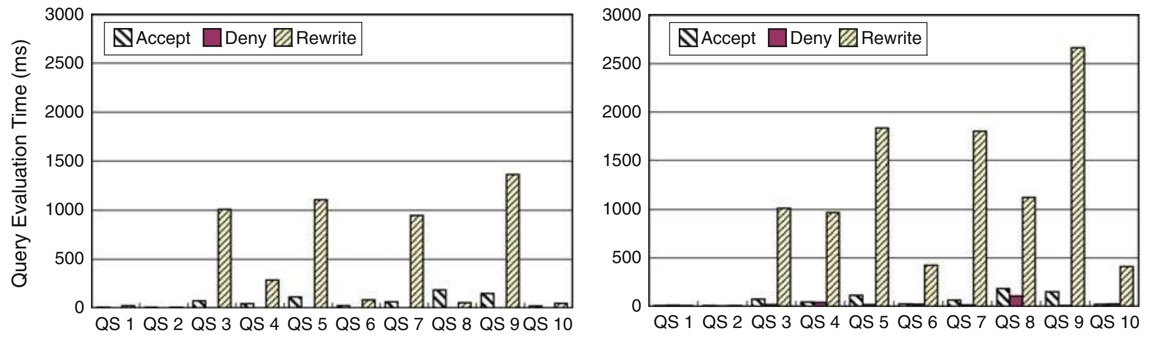


Fig. 12 End-to-end query processing time for QFilter approach: (*left*) query processing with QFilter; (*right*) query processing without QFilter

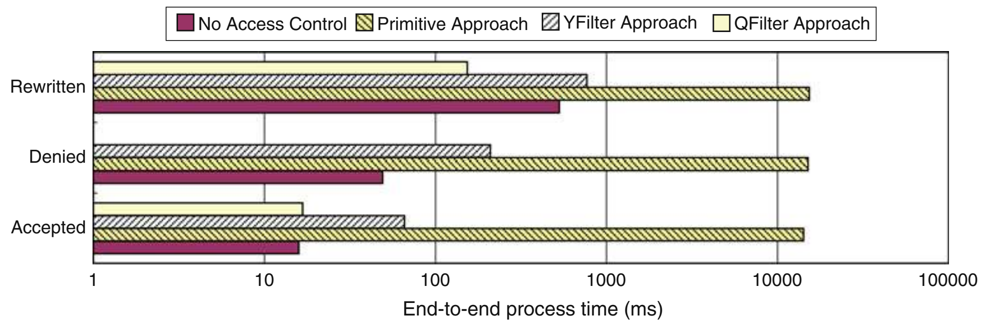


Fig. 13 End-to-end query processing time comparison of all approaches, in logarithmic scale

answer is returned. Note that we do not count the I/O time of the query input and the answer output. Note that for (d), since XML engines return only queried nodes without their ancestor tags, we manually wrote an external script to recover ancestor tags for UD. 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 an "under-estimate".

Figure 12 shows the categorized end-to-end query evaluation performance for QFilter approach and compares it with no-access-control approach (baseline). The original 5 MB XMark document is used. Figure 13 summarizes the comparison of the four approaches. In this experiment, due to the high computation of primitive approach, we used a smaller size (1.5 MB) document. QFilter-based pre-processing approach is a clear winner regardless of the query categories, and thus a promising solution for XML access control. It significantly outperforms the primitive approach. Note that many re-written queries contain wildcards (* or //); hence, they are evaluated less efficiently. An interesting phenomenon is that QFilter even outperforms no security check case on re-written queries. 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 contains less wildcards and yields less amount of XML nodes. Please also note that QFilter approach does not directly exploit

XML indices, which is usually constructed inside the XML engine. However, since the safe queries are submitted to the XML engine in the form of XPath/XQuery, all the engine-level optimization mechanisms, including indexing, are still applicable.

Since the post-processing approach requires a data filtering stage after Q is evaluated, 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.

6 Conclusion

Three dimensions of novel solutions are presented to support XML access controls without using views or security support of underlying databases. In particular, a pre-processing-based method, called QFilter, has been elaborated and shown to be particularly efficient and effective. QFilter, based on non-deterministic finite automata (NFA), rewrites user's insecure queries to secure ones not return any data violating access control rules. We validate QFilter by showing its guarantee not to return any violating data via theoretical analysis and by demonstrating its effectiveness through extensive experiments. As a result, QFilter demonstrates efficient and effective XML access control capabilities: (1) it does not require support from underlying database engine, which makes it feasible for any XML DBMS, native or RDBMS-based; (2) it consumes very small amount of memory, especially comparing with traditional view-based approaches; and (3) its execution time is very short so that it is practical in real-world applications.

Acknowledgments Liu was supported by Air Force Office of Scientific Research (AFOSR) FA9550-07-1-0527 (MURI), Army Research Office (ARO) W911NF-09-1-0525 (MURI), and National Science Foundation (NSF) CNS-0905131.

References

1. Ayyagari, P., Mitra, P., Lee, D., Liu, P., Lee, W.C.: Incremental adaptation of xpath access control views. In: ASIACCS '07: Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security, pp. 105–116 (2007)
2. Berglund, A., Boag, S., Chamberlin, D., Fernández, M.F., Kay, M., Robie, J., Siméon, J.: XML Path Language (XPath) 2.0. W3C Working Draft (2003). http://www.w3.org/TR/xpath20
3. Bertino, E., Castano, S., Ferrari, E.: Securing xml documents with author-x. IEEE Int. Comput. **5**(3), 21–31 (2001)
4. Bertino, E., Ferrari, E.: Secure and selective dissemination of XML documents. ACM Trans. Inf. Syst. Secur. (TISSEC) **5**(3), 290–331 (2002)
5. Boag, S., Chamberlin, D., Fernández, M.F., Florescu, D., Robie, J., Simeon, J.: XQuery 1.0: An XML Query Language. W3C Working Draft (2003). http://www.w3.org/TR/xquery
6. Bouganim, L., Ngoc, F.D., Pucheral, P.: Client-based access control management for XML documents. In: VLDB. Toronto, Canada (2004)
7. Bravo, L., Cheney, J., Fundulaki, I.: Accon: checking consistency of xml write-access control policies. In: Proceedings of the 11th International Conference on Extending Database Technology, pp. 715–719 (2008)
8. Bray, T., Paoli, J., Sperberg-McQueen, C. M., Maler, E., Yergeau, F.: Extensible Markup Language (XML) 1.0, 5th edn. (2008)
9. Cho, S., Amer-Yahia, S., Lakshmanan, L.V., Srivastava, D.: Optimizing the secure evaluation of Twig queries. In: VLDB. Hong Kong, China (2002)
10. Choi, B.: What are real dtds like? In: WebDB (2002)
11. Cuppens, F., Cuppens-Boulahia, N., Sans, T.: Protection of relationships in xml documents with the xml-bb model. In: First International Conference on Information Systems Security (ICISS), pp. 148–163 (2005)
12. Damiani, E., De Capitani di Vimercati, S., Paraboschi, S., Samarati, P.: A fine-grained access control system for XML documents. ACM Trans. Inf. Syst. Secur. (TISSEC) **5**(2), 169–202 (2002)
13. Damiani, E., Fansi, M., Gabillon, A., Marrara, S.: Securely updating xml. In: Knowledge-Based Intelligent Information and Engineering Systems, 11th International Conference (KES), pp. 1098–1106 (2007)
14. Damiani, E., Fansi, M., Gabillon, A., Marrara, S.: A general approach to securely querying xml. Comput. Stand. Interfaces **30**(6), 379–389 (2008)
15. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: Securing xml documents. In: 7th International Conference on Extending Database Technology, pp. 121–135 (2000)
16. Damiani, E., Vimercati, S.D.C.D., Paraboschi, S., Samarati, P.: Design and implementation of an access control processor for XML documents. Comput. Netw. **33**(6), 59–75 (2000)
17. Diao, Y., Franklin, M.J.: High-performance XML filtering: an overview of YFilter. IEEE Data Eng. Bulletin (2003)
18. Fan, W., Chan, C.Y., Garofalakis, M.: Secure xml querying with security views. In: SIGMOD '04: Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data, pp. 587–598. ACM Press, New York, (2004). http://doi.acm.org/10.1145/1007568.1007634
19. Fernandez, E., Gudes, E., Song, H.: A model of evaluation and administration of security in object-oriented databases. IEEE Trans. Knowl. Data Eng. (TKDE) **6**(2), 275–292 (1994)
20. Finance, B., Medjdoub, S., Pucheral, P.: The case for access control on xml relationships. In: 14th ACM International Conference on Information and Knowledge Management, pp. 107–114 (2005)
21. Fundulaki, I., Maneth, S.: Formalizing xml access control for update operations. In: 12th ACM Symposium on Access Control Models and Technologies, pp. 169–174 (2007)
22. Fundulaki, I., Marx, M.: Specifying access control policies for xml documents with xpath. In: Ninth ACM Symposium on Access Control Models and Technologies, pp. 61–69 (2004)
23. Gabillon, A.: An authorization model for xml databases. In: 2004 Workshop on Secure Web Service, pp. 16–28 (2004)
24. Gabillon, A., Bruno, E.: Regulating access to xml documents. In: Das'01: Proceedings of the Fifteenth Annual Working Conference on Database and Application Security, pp. 299–314. Kluwer Academic Publishers, Norwell (2002)
25. Godik, S., Moses, T. (Eds): eXtensible Access Control Markup Language (XACML) Version 1.0. OASIS Specification Set (2003). http://www.oasis-open.org/committees/xacml/repository/

26. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation. Addison-Wesley, Reading (2007)
27. Jiang, M., Fu, A.W.C.: Integration and efficient lookup of compressed xml accessibility maps. IEEE Trans. Knowl. Data Eng. **17**(7), 939–953 (2005)
28. Kudo, M., Hada, S.: XML document security based on provisional authorization. In: ACM Conference on Computer and Communications Security (CCS) (2000)
29. Kuper, G., Massacci, F., Rassadko, N.: Generalized xml security views. In: the Tenth ACM Symposium on Access Control Models and Technologies, pp. 77–84 (2005)
30. Kuper, G., Massacci, F., Rassadko, N.: Generalized xml security views. Int. J. Inf. Secur. **8**(3), 173–203 (2009)
31. Lee, D., Lee, W.C., Liu, P.: Supporting XML security models using relational databases: a vision. In: XML Database Symposium (XSym). Berlin, Germany (2003)
32. Lee, J.G., Whang, K.Y., Han, W.S., Song, I.Y.: The dynamic predicate: integrating access control with query processing in xml databases. VLDB J. **16**(3), 371–387 (2007)
33. Li, F., Luo, B., Liu, P., Lee, D., Mitra, P., Lee, W.C., Chu, C.H.: In-broker access control: towards efficient end-to-end performance of information brokerage systems. In: IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing, pp. 252–259 (2006)
34. Luo, B., Lee, D., Liu, P.: Pragmatic XML access control using off-the-shelf RDBMS. In: 12th European Symposium On Research in Computer Security (ESORICS). Dresden, Germany (2007)
35. Luo, B., Lee, D., Lee, W.C., Liu, P.: QFilter: fine-grained run-time XML access control via NFA-based query rewriting. In: ACM CIKM. Washington (2004)
36. Luo, B., Lee, D., Lee, W.C., Liu, P.: Deep set operators for XQuery. In: Second International Workshop on XQuery Implementation, Experience and Perspectives (XIME-P). Baltimore (2005)
37. Mealy, G.H.: A method for synthesizing sequential circuits. Bell Syst. Tech. J. **34**, 1045–1079 (1955)
38. Mella, G., Ferrari, E., Bertino, E., Koglin, Y.: Controlled and cooperative updates of xml documents in byzantine and failure-prone distributed systems. ACM Trans. Inf. Syst. Secur. **9**(4), 421–460 (2006)
39. Mohan, S., Klinginsmith, J., Sengupta, A., Wu, Y.: Acxess—access control for xml with enhanced security specifications. In: 22nd International Conference on Data Engineering, p. 171 (2006)
40. Mohan, S., Sengupta, A., Wu, Y.: Access control for xml: a dynamic query rewriting approach. In: 14th ACM International Conference on Information and Knowledge Management, pp. 251–252 (2005)
41. Murata, M., Tozawa, A., Kudo, M.: XML access control using static analysis. In: ACM Conference on Computer and Communications Security (CCS). Washington (2003)
42. Murata, M., Tozawa, A., Kudo, M., Hada, S.: Xml access control using static analysis. ACM Trans. Inf. Syst. Secur. **9**(3), 292–324 (2006)
43. Qi, N., Kudo, M.: Access-condition-table-driven access control for xml databases. In: Samarati, P., Ryan, P.Y.A., Gollmann, D., Molva, R. (eds.) ESORICS, Lecture Notes in Computer Science, vol. 3193, pp. 17–32. Springer (2004)
44. Qi, N., Kudo, M.: Xml access control with policy matching tree. In: ESORICS 2005, 10th European Symposium on Research in Computer Security, pp. 3–23 (2005)
45. Qi, N., Kudo, M., Myllymaki, J., Pirahesh, H.: A function-based access control model for xml databases. In: 14th ACM International Conference on Information and Knowledge Management, pp. 115–122 (2005)
46. Rabitti, F., Bertino, E., Kim, W., Woelk, D.: A model of authorization for next-generation database systems. ACM Trans. Database Syst. (TODS) **16**(1), 89–131 (1991)
47. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-Based Access Control Models. IEEE Comput. **29**(2) (1996)
48. Schmidt, A.R., Waas, F., Kersten, M.L., Florescu, D., Manolescu, I., Carey, M.J., Busse, R.: The XML Benchmark Project. Tech. Rep. INS-R0103, CWI (2001)
49. Simeon, J., Fernandez, M.: Galax V 0.3.5 (2004). http://db.bell-labs.com/galax/
50. Stoica, A., Farkas, C.: Secure xml views. In: Gudes, E., Shenoi, S. (eds.) DBSec, IFIP Conference Proceedings, vol. 256, pp. 133–146. Kluwer (2002)
51. De Capitani di Vimercati, S., Marrara, S., Samarati, P.: An access control model for querying xml data. In: Workshop on Secure web services, pp. 36–42 (2005)
52. Xiao, Y., Luo, B., Lee, D.: Security-conscious XML indexing. In: International Conference on Database Systems for Advanced Applications (DASFAA). Bangkok, Thailand (2007)
53. Yu, T., Srivastava, D., Lakshmanan, L.V., Jagadish, H.V.: Compressed accessibility map: efficient access control for XML. In: VLDB. Hong Kong, China (2002)
54. Zhang, H., Zhang, N., Salem, K., Zhuo, D.: Compact access control labeling for efficient secure xml query evaluation. Data Knowl. Eng. **60**(2), 326–344 (2007)