A Flexible Framework for Architecting XML Access Control Enforcement Mechanisms

Bo Luo, Dongwon Lee, Wang-Chien Lee, and Peng Liu

The Pennsylvania State University
University Park, PA, 16802, USA
{bluo,dlee,wlee,pliu}@ist.psu.edu

Abstract. Due to the growing interests in XML security, various access control schemes have been proposed recently. However, little effort has been put forth to facilitate a uniform analysis and comparison of these schemes under the same framework. This paper presents a first attempt toward a flexible framework that can capture the design principles and operations of existing XML access control mechanisms. Under this framework, we observe that most of existing XML access control mechanisms share the same design principle with slightly different orderings of underlying building blocks (i.e., data, query, and access control rule). Furthermore, according to the framework, we identify four plausible approaches to implement XML access controls, namely built-in, view-based, pre-processing and post-processing. Finally, we compare the actual performance of different approaches.

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 data they are allowed to access. Toward this goal, recently, many research and industrial proposals have appeared (e.g., [1, 3, 4, 8]).

However, there has been little effort to facilitate a uniform analysis and comparison of these proposals. Therefore, in this paper, we made such an attempt to identify necessary building blocks and operations under the framework. Having such a framework brings several benefits: (1) Without a uniform view, comparing different XML access control mechanisms is somewhat similar to comparing apple with orange. By having a flexible framework that can represent many proposals, one can easily compare different approaches from the same and "fair" perspective; (2) Framework can help users see the architectural uniqueness of an approach in an intuitive manner. That is, once one understands the basic

building blocks of the framework, it is intuitive to view other proposals in terms of the building blocks; (3) By combining different building blocks in different orders, one can devise novel approaches (or implementations) of XML access control mechanisms that are not known before.

In summary, the **contributions** of this paper are as follows:

– We present a flexible framework that consists of three building blocks (i.e., data, query, access control rules) and a set of operations (e.g., evaluate, merge, etc). Based on these elements, we present different ways of implementing XML access controls with their pros and cons, namely built-in, pre-processing, post-processing, etc. To our best knowledge, this is the first attempt to model and compare different XML access control mechanisms under the same roof (Section 3).

– We demonstrate that the proposed framework can easily capture majority of the known XML access control mechanisms (e.g., [8, 9, 1, 4]) in a succinct and consistent way (Section 4).

– Finally, after implementing four representative XML access control mechanisms, we present experimental result from a performance study of those mechanisms (Section 5). In general, the pre-processing approach outperforms the others.

The rest of the paper is organized as follows: Section 2 presents related works of this paper. Section 3 presents the main framework that we propose, and Section 4 discusses architectures of some of the existing XML access control mechanisms under the proposed framework. Section 5 provides a performance comparison of four representative XML access control mechanisms. Finally, a conclusion is drawn in Section 6.

2   Related Work

XML access control in general has two aspects: *access control models* and *enforcement mechanisms*. The focus of this paper is on the latter.

Several authorization-based XML access control models are proposed. In [11], 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 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. 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 [10]. Although our proposal is based on existing XML authorization models such as [4], we focus

on how to architect and implement XML access control mechanisms on top of XML engines without security support.

From the enforcement mechanism perspective, existing XML access control methods are either view-based or relying on the XML engine to enforce node-level access control. The idea of view-based enforcement is to create and maintain a *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. The view is generated by using the set of authorizations granted to the user to filter off the nodes that the user should not access. During run time, each user can simply run his queries against his view. In [5] and [4], a detailed view-based enforcement mechanism is proposed. Although views can be prepared offline, view-based enforcement has two serious limitations: (1) not scalable in managing and maintaining views when there are a large number of roles (or users), (2) high storage cost. To tackle this problem, [15] proposes a method to compress XML views. However, view-independent enforcement mechanisms are sometimes more desirable.

Letting XML engines enforce access control at the node-level is a view-independent enforcement mechanism, but the complexity of managing and maintaining authorizations can be too significant to make this enforcement mechanism practical. The idea is to associate an access-control-list with each node of the XML document. The major complexities are: (1) whenever a user is created or removed, or an authorization is granted or revoked, the XML engine has to "refresh" its access control lists; (2) the query processing overhead can be substantial; (3) this enforcement mechanism is useless when XML data are managed by a RDBMS, as many real world applications do; (4) how to manage the authorization inheritance relationships among data nodes? [3] addresses this issue by mitigating the problem (2), but cannot solve the other two problems; (5) when XML documents are huge, using XML engines to enforce access control may not be cost-effective.

To further reduce the overhead of the XML engine, [9] proposed an automata-based static analysis that identifies XML queries that are either "entirely" authorized or "entirely" prohibited before the queries are submitted to an XML engine. Therefore, if a query Q is completely prohibited by access control rules, then there is no need to submit Q to an XML engine, and Q can be simply thrown away outright. Conversely, if one is certain that Q does not have any conflicts with access control rules, Q may be processed as if it is a regular query without security concern. However, for the "partially" authorized XML queries, [9] still relies on an XML engine to filter out the data nodes that users do not have authorizations to read or write. The proposed solution in [8] removes this problem so that query processing as well as security enforcement are optimized regardless of the query or access control types. That is, our solution is independent of the underlying XML engine or the usage of views, solving the above enforcement problems naturally.

3 A Framework for XML Access Control Enforcement

3.1 XML Security Model

Since the focus of our framework is on *how to enforce access controls*, rather than on *how to define a security model* itself, the choice of a particular XML security model that we use in this paper is insignificant. Nevertheless, to simplify the presentation of the paper, let us first define a model as follows.

In short, we adopt an XML access control model from [4] and incorporate role-based access control from [12] to make our access control mechanisms more pragmatic. In this 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 (or XQuery) is used for specification of queries as well as identification of nodes. The node-level authorization is specified via access control rules (ACR), each of which is a 5-tuple: $ACR = \{subject, object, action, sign, type\}$, where (1) *subject* is to whom an authorization is granted (e.g., user or role); (2) *object* is part of an XML data specified by an XPath expression; (3) *action* consists of read, write, and update[1]; (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 authorization is not explicitly specified in ACR are considered to be "access denied". It is possible for a node to have more than one relevant access control rules. If conflicts occur among such rules, denial takes precedence. When an answer returned from databases does not contain any security-violating data in it, the answer is called *safe answer (SA)*, and *un-safe answer (UA)* otherwise. Similarly, if a query produces only safe answers, then the query is called *safe query (SQ)*, and *un-safe query (UQ)* otherwise.

3.2 Building Blocks

We view the XML access control mechanism as the interplay of three building blocks – *data*, *query*, and *access control rule* as follows:

- **Data (D)** indicates the XML data (or document) that contains the answers users are looking for. Often the data are stored in native XML engines or RDBMS, but the choice of storage system is irrelevant to the discussion of this paper.

- **Query (Q)** describes the information that users want, and can be viewed as a conceptual pointer to the desired data in D. In XML domain, query is often written in either XPath or XQuery language. When a Q is issued by a user, Q has the same security role as what the user has.

---

[1] In this paper, we focus on the read action since write/update operations for XML model are still being designed by W3C.
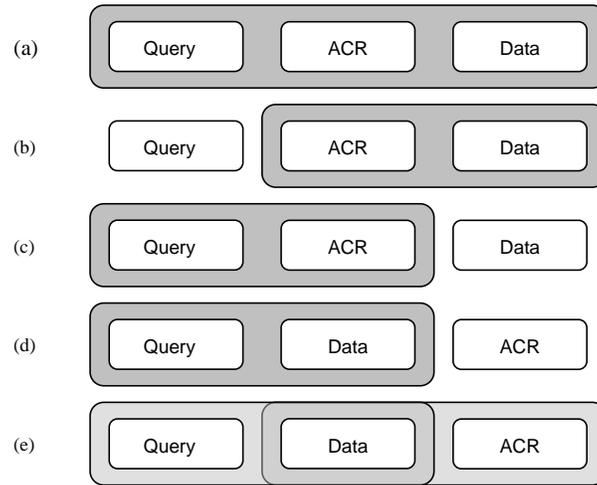
Fig. 1. Different combinations of building blocks in the framework.

- **Access Control Rules (ACR)** is a list of 5-tuple access control rule, describing the security policy of some roles. When a portion of data in D that do not violate policies of ACR are returned, it is a "safe" answer.

Note that D, Q and ACR are independent components, and thus can be located independently and processed separately. Figure 1 illustrates various combinations of the three building blocks, where gray box implies that building blocks in it are (1) co-located (in a spatial sense); and/or co-processed (in a temporal sense). For instance, (a) can be interpreted as: all three building blocks must be (1) co-located in a single system; and/or (2) processed at the same time. Below, we will consider both aspects of the framework.

- (a) indicates a scenario where all three building blocks are co-located in a single system. For instance, conventional RDBMS supports relational access control via the embedded support of GRANT/REVOKE. In such a setting, Q is issued against both D and ACR which are stored together;

- (b) is a slight modification of (a) in that Q can be issued remotely while ACR must be stored together with D in a system. Typical example of this scenario includes the client-server model such as web-based database interface. On the other hand, from the temporal aspect, (b) illustrates the view-based XML access control mechanism where ACR and D are processed first (yielding a safe view), and then Q is evaluated against the view. Whichever case it is, the data provider must be able to support XML access control mechanism;

- In the spatial sense, (c) indicates a scenario where one party holds Q and ACR, while D is stored elsewhere. For instance, D is provided by a data provider while ACR is provided by a data mediator who connects end users

with raw data sources with marginal fees. Once acquiring an adequate security role from the mediator by paying the fee, end users can issue a query to D. On the other hand, in the temporal sense, (c) implies that Q and ACR can be pre-processed prior to D. Therefore, for optimization, one can "merge" Q and ACR such that new output Q' can be processed against D more efficiently;

- (d) shows a scenario where only ACR is stored elsewhere. Since Q and D are stored together, conventional databases without access control support can be used to first evaluate Q against D. When ACR itself carries security-conscious information and has to be stored securely, this approach can be adopted; and

- Lastly, (e) is a conceptual merge of (b) and (d). Since the final "safe" answers are those data that can pass through constraints of Q as well as ACR, one can do intersection of two data sets – one from evaluating Q against D, and the other from enforcing ACR against D.

3.3 Operators

By viewing the three building blocks of the previous section as "operands", here, we present a few core "operators", thus forming an Algebra in a sense.

- **D' = evalQuery(Q, D)**. This operator takes a query Q issued by a user and a data D, and returns a data D' as the answer. if either input Q is a "safe query" or D is a "safe data", then the output D' is also a "safe data".

- **D' = evalRule(ACR, D)**. This operator applies the 5-tuple access control rules ACR against D, and produce a set of data D' (i.e., D'_1, ..., D'_n) as return, one for each role. That is, each returned D' is a portion of data in D that a role is entitled to access. Note that D' can be a virtual concept. For instance, D'_1 and D'_2 can exist as augmented taggings to D', instead of being physically-returned data.

- **Q' = merge(Q, ACR)**. This operator "re-writes" the input query Q using ACR such that parts of Q that violate the security policies specified in ACR are pruned. The output query Q' is thus a conceptual merge of Q and ACR. For instance, suppose a manager "John" issues the following query `Q://dept[@loc='East']//salary` when ACR has only 1 rule in it: `<manager, //member/salary, +, read, LC>` (i.e., managers can read any member's salary information). Then, $merge(Q, ACR)$ would generate: `Q'=//dept[@loc='East']//member/salary`.

Note that the proposed three operators can be nested in an arbitrary manner, together with the traditional set operators (i.e., $\cap$, $\cup$, and $-$). Now, let us see how different scenarios of Figure 1 can be captured using the operators.

– **Figure 1(a) and 1(b)**. Since all of our operators are binary, both scenarios (a) and (b) can be captured as "$evalQuery(Q, evalRule(ACR, D))$". Note that since $evalRule(ACR, D)$ returns a list of D', instead of a single D', in order to use it in a nested fashion, there needs to be another operator that lets pick one of the D' such as "$evalQuery(Q, pickOne(evalRule(ACR, D)))$". However, for simplicity, we omit this operator.

– **Figure 1(c)**. The operation "$evalQuery(merge(Q, ACR), D)$" captures the scenario (c). Note that how individual operator is "implemented" in practice is not discussed yet, and to be explored in Section 4. For instance, $merge(Q, ACR)$ operator is implemented in two ways in [8], called primitive and QFilter.

– **Figure 1(d)**. The operation "$evalRule(ACR, evalQuery(Q, D))$" captures the scenario (d). Note the potential inefficiency stems from the fact that $evalQuery(Q, D)$ is processed first so that intermediate (possibly un-safe) data must be carried to the second step of $evalRule(ACR, D)$. In this scenario, the first $evalQuery(Q, D)$ may need to do extra task of keeping ancestor tags or predicates. For instance, after the `Q:/a/b` returns `<b>` nodes, when an access control rule has `/a[c]/b`, it cannot be checked since necessary tags are already stripped out.

– **Figure 1(e)**. This scenario can be captured as the operation "$evalQuery(Q, D) \cap evalRule(ACR, D)$" if the domain compatibility of the $\cap$ operator is provided. Consider the following case: `Q://a` and `ACR:<admin, //b, +, read, RC>` (i.e., administrators can read all `b` elements and their descendants). Furthermore, suppose the first operator $evalQuery(Q, D)$ returns an answer $\{a_1, a_5, a_7\}$, while the second operator $evalRule(ACR, D)$ returns a subtree rooted at b_2 that contains $\{a_3, a_5, a_7, a_{10}\}$ as sub-elements. In this case, the first sub-answer has the type of `<a>` while the second sub-answer has the type of `<b>`, and therefore, their domain is not compatible. However, two elements of `<a>` – a_5 and a_7 – must be returned as the final answer since they satisfy both constraints of Q and ACR. How to achieve this intelligent intersection is beyond the scope of this paper, and for instance explored in [8].

4 Current XML Access Control Enforcement Mechanisms under the Framework

In this section, we discuss the current XML access control approaches, and show/compare how they are architected under our framework.

4.1 Available Approaches

– **RDBMS-style Approach.** Typical RDBMS uses the role-based access control (RBAC) model where users are assigned a certain role which has predetermined GRANT/REVOKE privileges. Access control rules are stored

in the access control tables (ACT), along with data. Therefore, architecturally, they typically adopts the scenario of Figure 1(a) (although query can be issued remotely using database interfaces such as ODBC). To out best knowledge, there is no commercially available native XML databases with full access control support at this point.

– **Instance-tagging Approach.** When ACR and D are available together like in Figure 1(b), one can traverse entire XML data tree, and tag each (element and attribute) node by its corresponding security information. [3], for instance, uses this approach although their focus is on optimizing the query evaluation, not the access control mechanism itself. With the two rules `<user1, //a//c, read, +, LC>` and `<user2, //b//c, read, +, LC>`, the `<c>` elements in the tree would have taggings, specifying that they are readable by both "user1" and "user2". Assuming there is some kind of index on this tagged information, then secure query evaluation can be provided. That is, when "John" with a "user1" role issues a query "//c", databases can retrieve all `<c>` elements under `<a>`, but not under `<b>` using the index. In some sense, this approach is related to the subsequent view-based approach.

– **View-based Approach.** By adopting the architecture of Figure 1(b), view-based approach takes advantage of the fact that ACR and D are either co-located or co-processed. By processing $evalRule(ACR, D)$ first, therefore, this approach produces a set of data, D'_1, ..., D'_n for each role, thus creating a number of "views". Since each view contains only "safe" data for that particular role, query can be processed on this view without any further special care, making the query processing very efficient. The examples of view-based approaches recently proposed include [15, 4, 1], and is one of the most popular XML access control mechanisms. Depending on the details of the algorithms, the views can be maintained either physically or virtually.

Since the I/O and space costs for constructing views are amount to evaluating $evalRule(ACR, D)$, it is dependent on the number of roles in ACR and the size of D. However, often, this view construction is performed off-line, and thus the cost issue becomes less important. When the space cost becomes a major issue due to large number of views (e.g., million roles in Internet environment), then one may mitigate the problem using the compression-based techniques suggested in [15]. However, this approach still has to take extra burden to maintain the views. When update occurs to either ACR or D, synchronization must be performed to views. Overall, the view-based approach is fast in answering user queries but may have to pay high I/O and storage cost, and the extra complexity of view maintenance. Another drawback of this approach is that since ACR must be processed against D first, the database engines must be aware of the security aspect. That is, one cannot implement this approach using off-the-shelf databases that do not have built-in security support.

– **Pre-processing Approach.** Scenarios depicted in Figure 1(c) allows the handling of Q and ACR prior to D. Since the D are de-coupled from ACR, databases do not need to understand ACR. To exploit this property, one can probe only Q and ACR to do optimization. Known approaches in this category include two proposals from [8] and a proposal from [9].

In [8], we proposed *primitive* and *QFilter* as a pre-processing approach. The primitive approach simply merges Q and ACR with $\cap$ operator to construct $Q' = Q \cap ACR^2$. This Q' is then passed to the XML database capable of handling the set operator. Although simple to implement, its performance is highly dependent on the capability of underlying XML database. To remedy the problem of the primitive approach, QFilter tries to produce a more "optimized" Q' by pruning unnecessary parts as early as possible. It performs the "intersection" of ACR and Q using the extended non-deterministic finite automata (NFA). Informally, suppose we have the query `Q://dept[@loc='East']//salary` when ACR has 2 rules in it: `<manager, //member/salary, +, read, LC>` (i.e., managers can read any member's salary), and `<manager, //member[@proj-type='secret']/salary, -, read, LC>` (i.e., managers cannot read member's salary if they work for a secret project). Then, the primitive approach would produce an output query Q' as: "`//dept[@loc='East']//salary` $\cap$ `//member/salary` $-$ `//member [@proj-type='secret']/salary`." However, the QFilter approach would instead produce Q' as:

"`//dept[@loc='East']//member[@proj-type<>'secret']/salary`",

which is often processed much faster. More importantly, since the new query Q' fully preserves both constraints of Q and ACR, even if Q' is processed by normal databases that do not support access controls, the output of $evalQuery(Q', D)$ is the "safe" answer. The details of the QFilter algorithm to achieve this optimization is beyond the scope of this paper, and can be found in [8].

On the other hand, [9] proposed another approach called *static analysis*, which is a hybrid of pre-processing and internal XML database security check. The idea is to recognize two cases in the pre-processing stage: "access-fully-granted" and "access-fully-denied". That is, in our framework, (1) access-full-granted occurs when $evalQuery(Q, D) \subseteq evalRule(ACR, D)$. Since all answers returned from Q are fully allowed by ACR, then $Q' = Q$ holds. This means that the original user query can be processed by databases without any special care; and (2) access-fully-denied occurs when $evalQuery(Q, D) \cap evalRule(ACR, D) = \emptyset$. That is, all the answers that the user is asking for are prohibited to access by ACR. In this case, there is no point of sending any query to databases, and thus system simply returns null to the user right

---

[2] In reality, the primitive algorithm is a bit more complicated to take care of the subtle differences in the semantics of "$+/-$" sign and "LC/RC" type.

away. Compared to the QFilter approach, the static analysis method lacks of the capability to handle the case: $evalQuery(Q, D) \cap evalRule(ACR, D) \neq \emptyset$, i.e., some parts of the answers that the user is asking for are blocked, but other parts are accessible. Mainly due to this reason, [8] demonstrated the QFilter method can outperform the static analysis method by significant margin.

– **Post-processing Approach.** Figure 1(d) illustrates the post-processing scenario, where Q is applied to D first (where no security enforcement is engaged), and then ACR is examined second. Since the first and second step may be temporally and spatially far apart, the risk of carrying unnecessary intermediate data in the middle can hamper this approach significantly. One may use data filter techniques such as YFilter [6] in the second step to remove the forbidden contents from the unsafe answers. The cost to construct YFilter depends on ACR only, thus could be performed off-line efficiently. The final step of data filtering is the major performance bottleneck if the intermediate data contains a large volume of forbidden data in them. In additions, the post-processing filters often require the intermediate answers (after $evalQuery(Q, D)$) to retain full path to the nodes that query requested. However, current XML database engine such as Galax returns only requested nodes without their ancestors. Therefore, to implement XML access controls using the post-processing approach, one has an extra burden to recover all the ancestor tags to the root.

Let us emphasize that all of the aforementioned approaches are well captured in our framework, usually in a slight different orderings. For instance, note the slight difference of the view-based, pre-processing, and post-processing approaches:

– View-based: $SA = evalQuery(Q, evalRule(ACR, D))$
– Pre-processing: $SA = evalQuery(merge(Q, ACR), D)$
– Post-processing: $SA = evalRule(ACR, evalQuery(Q, D))$

Note that at the end, users always get the "safe answer" (SA) back. Figure 2 depicts details of three XML access control enforcement mechanisms using our framework.

4.2 Qualitative Comparison

In this section, let us do a close examination on the three (important) categories: view-based of Figure 1(b), pre-processing of Figure 1(c), and post-processing of Figure 1(b). End-to-end processing time of these approaches are illustrated in Figure 3. We observe that typically an XML access control mechanism involves three separate operations: (1) off-line service preparation, (2) on-line query processing, and (3) service maintenance.

– **Off-line Service Preparation.** This step is typically devoted on tasks to help speed-up the subsequent query processing step, and done off-line.
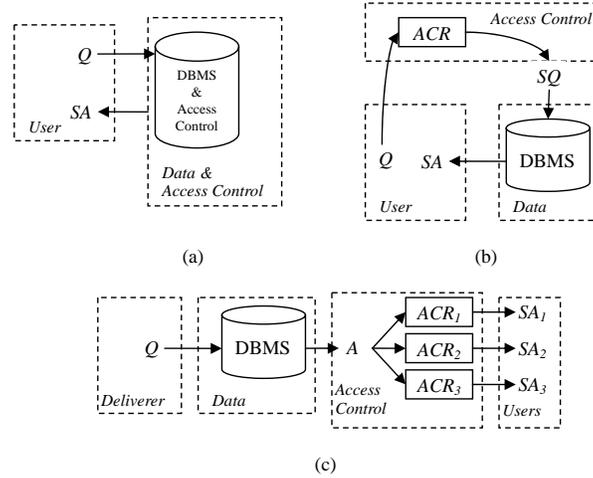
Fig. 2. Illustration of three XML access control approaches under our framework: (a) view-based, (b) QFilter, and (c) YFilter

Obviously, view-based approach would need to generate views per roles in this step. Similarly, the pre-processing approach like QFilter or static analysis method spends this time on constructing needed data structures (e.g., NFA). For the post-processing approach, one can build up some kind of index on ACR (e.g., given a "role", quickly retrieve all relevant rules from ACR) so that later post-filtering process can run faster. Note that in this stage, Q from users are not known, and both ACR and D are the sole resources. Therefore, often the cost for service preparation depends on the size of ACR and D. Moreover, when the preparation requires non-trivial probing of ACR such as QFilter case, the complexity of ACR also does affect the cost. However, overall, since these tasks are done off-line, they do not contribute much to the performance of whole XML access control mechanisms, and thus omitted in our experimental comparisons of Section 5.

– **On-line Query Processing.** Once Q is issued, the task of evaluating Q while ensuring security policies in ACR is done in this step, and must be done on-line (unless the submitted query is part of batch-process). The end output of this task must be the "safe answers". Thus, the end-to-end on-line query processing time is the time-line between Q and SA in Figure 3.

For the view-based approach, the query processing can be efficient since there is no need for additional security check (i.e., each view contains only safe data for the role, after all). For the pre-processing approach, the performance largely depends on the quality of the re-written query from the pre-processing. For instance, if the primitive method generates a re-written query Q' as "$s_1 \cap ... \cap s_n - t_1... - t_m$" ($n, m \gg 1$), then the evaluation
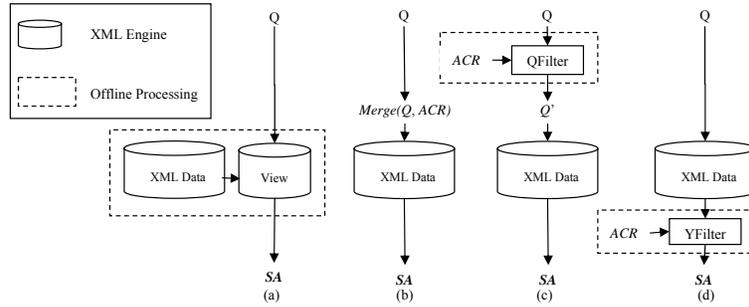
Fig. 3. Processing flow of XML access control mechanisms: (a) view-based approach, (b) primitive pre-processing, (c) QFilter-based pre-processing, and (d) YFilter-based post-processing.

Table 1. Qualitative comparison of different XML access control mechanisms.

Approach	Preparation	Processing	Maintenance
View-based	Medium	Good	Medium
Pre-processing	Good	Medium/Good	Good
Post-processing	Good	Bad/Medium	Good

of the Q' can be quite slow. Other pre-processing approaches like QFilter or static analysis method improve it drastically via early-pruning of access-full-granted or access-fully-denied cases and via improved query re-writing in $merge(Q, ACR)$. For the post-processing approach, the security check is pipelined after the query evaluation, and thus can be disadvantageous in terms of performance. Post-filtering time is highly dependent on the size of unsafe answer set.

– **Service Maintenance.** In general, any service preparations done off-line need to be maintained when update occurs. For instance, when D is changed (e.g., new sub-tree is inserted to D), view-based approach needs to (incrementally) re-construct relevant views. However, the changes to D do not affect the pre-processing or post-processing approach. On the other hand, when ACR is changed, it affects the pre-processing (e.g., an NFA needs to be updated) and post-processing approach (e.g., index on ACR needs to be updated).

The summary of the qualitative comparison of three scenarios of Figure 1 is summarized in Table 1. Note that the query processing cost of the post-processing approach heavily depends on the size of intermediate un-safe data and/or the complexity of rules in ACR.

Table 2. Summary of roles and rules.

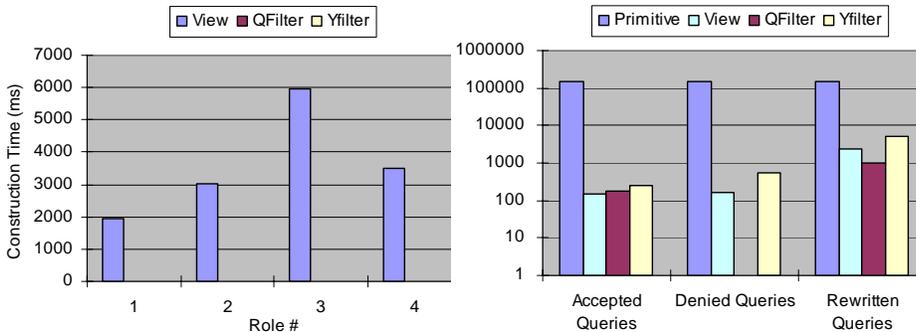| Role	Policy	Size (KB)	# of + rules	# of − rules
#1	Can view all information, except auction.	1,525	6	0
#2	Can view all category, north America item, and user information except for their private ones.	1,279	8	2
#3	Can view all the closed auctions, basic item and user information except for their private ones.	1,256	8	2
#4	Can view all the open auctions and basic item information.	1,352	6	0

Fig. 4. Performance comparison: (a) Service preparation time (ms); and (b) Query evaluation time (ms).

5 Performance Evaluation

Now we validate the analysis of Section 4 with the experimental results. We use Galax 0.3.1 [14] as the underlying XML engine, and XMark [13] schema and data set. Overall, we experimented with: (1) for Q, user-denied and synthetic XPath queries. Depending on the complexities of queries, we identified 8 categories; (2) for ACR, user-defined and synthetic access control rules in the range of $0 - 500$ rules ; and (3) for D, the sizes ranges 500KB – 2.5MB. Among all these, here we present a simple case of: 32 rules (of 4 roles) in ACR and 200 synthetic queries against 2.5MB data. Note that other results not shown here are still consistent with the presented case, and are available in [8].

We did not really implement the view-based approach. Instead, we simulate it by evaluating ACR on data to create answer set as views, and then evaluate queries on these views. Same as the situation in the YFilter approach, answer set of evaluating ACR on the document do not have ancestor tags and special care was taken to trace back the ancestor axis to recover full path. However, this step is not included in the comparison below.

Figure 4(a) shows the service preparation time of view-based, QFilter, and YFilter. The view-based approach takes the longest time while both QFilter and YFilter approaches are quite fast (appears to be 0 in the graph). The end-to-end query processing time is shown in Figure 4(b) (in logarithmic scale) for the 200 synthetic queries of the role #1. One can clearly observe:

– The primitive pre-processing approach performs the slowest since the underlying XML engine (i.e., Galax)'s performance degrades as the number of set operators such as $\cap$ or $\cup$ in the re-written query Q' increases when it evaluates $Q' = merge(Q, ACR)$. On the contrary, YFilter-based post-processing approach turns out to be faster than the primitive pre-processing since the intermediate data after evaluating $evalQuery(Q, D)$ was significantly small, incurring little cost to post-processing task. However, when the size of intermediate (unsafe) data increases, the post-processing approach often becomes slower than the primitive pre-processing.

– The view-based and QFilter pre-processing approaches are the fastest. For fully-accepted queries (i.e., $evalQuery(Q, D) \subseteq evalRule(ACR, D)$), "$Q' = Q$" holds, and thus the view-based approach is faster than even the QFilter approach, as it evaluates the query on a smaller data set of "views". For fully-denied queries (i.e., $evalQuery(Q, D) \cap evalRule(ACR, D) = \emptyset$), the QFilter approach takes almost no time since the query is rejected outright without being sent to databases for evaluation. For re-written queries (i.e., $evalQuery(Q, D) \cap evalRule(ACR, D) \neq \emptyset$), the QFilter approach exhibits a better performance mainly due to its good query rewriting algorithm utilizing pre-constructed NFA. Often, QFilter rewrites general paths having "*" or "//" into more specific paths, which tend to be processed faster in evaluation. In additions, due to the existence of "*" and "//" in both Q and ACR, Q' may include some paths which are not allowed by the schema, and those can be easily detected and ruled out by the underlying XML engine. As a result, while evaluating $evalQuery(Q', D)$ and $evalQuery(Q, V)$ yields the same safe answers, the former tends to perform faster.

6 Conclusion

In this paper, we proposed a flexible framework that can capture most of the current XML access control enforcement mechanisms using the same set of building blocks (query Q, access control rules ACR, and data D) and operators ($evalQuery(Q, D)$, $evalRule(ACR, D)$, $merge(Q, ACR)$). Using the framework, we have identified various architectural settings of access control scenarios. Especially, by focusing on three representative approaches – view-based, pre-processing, and post-processing, we showed and compared the pros and cons of each scenario. Furthermore, by examining many existing XML access control mechanisms, we identified which belongs to which category, providing easy and intuitive platform to understand and compare different proposals. Finally, experimental validations to confirm our qualitative comparison are presented.

In short, pre-processing approach such as QFilter or static analysis method is promising due to its low maintenance cost and high performance.

References

[1] E. Bertino and E. Ferrari. "Secure and Selective Dissemination of XML Documents". *IEEE Trans. on Information and System Security (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. http://www.w3.org/TR/2000/REC-xml-20001006.

[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". *IEEE Trans. on Information and System Security (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 Trans. on Knowledge and Data Engineering (TKDE)*, 6(2):275–292, 1994.

[8] B. Luo, D. Lee, W.-C. Lee, and P. Liu. "QFilter: Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting". Technical report, Penn State University, Jan. 2004. (Submitted for publication).

[9] M. Murata, A. Tozawa, and M. Kudo. "XML Access Control using Static Analysis". In *ACM Conf. on Computer and Communications Security (CCS)*, Washington D.C., 2003.

[10] F. Rabitti, E. Bertino, and G. Ahn. "A Model of Authorization for Next-Generation Database Systems". *ACM Trans. on Database Systems (TODS)*, 16(1):89–131, 1991.

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

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

[13] 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.

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

[15] 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.