HyXAC: a Hybrid Approach for XML Access Control

Manogna Thimma
Cerner Corporation
Kansas City, MO, USA
manogna.thimma@cerner.com

Tsam Kai Tsui
NIPR
Kansas City, MO, USA
ttsui@nipr.com

Bo Luo
EECS, University of Kansas,
Lawrence, KS, USA
bluo@ku.edu

ABSTRACT

While XML has been widely adopted for information sharing over the Internet, the need for efficient XML access control naturally arise. Various XML access control enforcement mechanisms have been proposed in the research community, such as view-based approaches and pre-processing approaches. Each category of solutions has its inherent advantages and disadvantages. For instance, view based approach provides high performance in query evaluation, but suffers from the view maintenance issues.

To remedy the problems, we propose a hybrid approach, namely HyXAC: Hybrid XML Access Control. HyXAC provides efficient access control and query processing by maximizing the utilization of available (but constrained) resources. HyXAC first uses the pre-processing approach as a baseline to process queries and define sub-views. In HyXAC, views are not defined in a per-role basis, instead, a sub-view is defined for each access control rule, and roles with identical rules would share the sub-view. Moreover, HyXAC dynamically allocates the available resources (memory and secondary storage) to materialize and cache sub-views to improve query performance. With intensive experiments, we have shown that HyXAC optimizes the usage of system resource, and improves the performance of query processing.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration

Keywords

Security, XML, Access control, View

1. INTRODUCTION

The eXtensible Markup Language (XML) has become very popular for information sharing in the Internet age. It was designed to store and transport semi-structured data. Due to the increased use of XML documents over the web, the need to secure these documents has increased. In a multi-user system, where the information is being shared across users who have different access rights, it is very important to implement a security model that gives controlled access to the authorized users. XML access control was introduced to suit this purpose. XML access control research could be roughly categorized into *access control models* and *access control enforcement mechanisms*. Access control models define how access control rules are specified (e.g. how to specify "who can access which information under what circumstances"), and how such rules should be enforced (e.g. how to handle conflict rules.) Meanwhile, enforcement mechanisms implement such access control rules for XML databases. Various access control models and enforcement mechanisms have been proposed in the research community. In this paper, we focus on XML access control enforcement.

There are different categories of enforcement mechanisms proposed in the literature [22], such as built-in approaches, the pre-processing approaches, view-based approaches, and postprocessing approaches. In particular, the view-based approaches, as the most conventional mechanism, create and manage views for every user/role by making a (virtual) copy of all the data that are accessible by the user/role. All the queries are evaluated on the views of the corresponding roles. The pre-processing approaches modify incoming user queries to new "safe" queries, which request only authorized data. Such queries can then be evaluated on the XML document without any further security protection. Post processing approaches evaluate all the queries from the user on the database and get "unsafe" results. Once the results are obtained, they are pruned to discard unauthorized nodes. While all these approaches are secure, each category has its own advantages and disadvantages. In particular, the view-based approaches are considered the fastest for query processing (with materialized views) because queries are answered by smaller documents (views). However, view maintenance becomes an issue – it is non-trivial to maintain and synchronize a large number of views. On the other hand, the pre-processing approaches introduce minimum overhead for access control enforcement. However, queries are still evaluated against the original XML documents – query processing could be slow when the documents are large, especially when caching or indexing is not well supported in the XML DBMS. To the best of our knowledge, there is no approach that tries to utilize multiple XML access control enforcement mechanisms to provide a hybrid solution, which combines the advantages of mechanisms from different categories.

SACMAT'13, June 12–14, 2013, Amsterdam, The Netherlands.

In this paper, we introduce a hybrid approach for XML access control, namely HyXAC, which combines pre-processing and view-based approaches to provide secure and efficient query processing, as well as maximize the utility of available resources. The HyXAC approach first adopts the pre-processing approach (in particular, the QFilter approach [23, 24]) as the baseline to process the queries. Unlike conventional view-based approaches that construct a view for each role, HyXAC defines a sub-view for every positive access control rule (or a set of rules). A query accepted by a positive access control rule is evaluated on the corresponding sub-view. Fine-grained view management also allows sub-views to be shared across multiple roles. This eliminates the need for maintaining a view for every role, thus reducing the number of views in the DBMS. Moreover, HyXAC enables dynamic allocation of resources for materializing and caching sub-views. Based on a cost-benefit analysis, dynamic view management is introduced to achieve optimal query performance for the available resource.

The main contributions of this paper are: (1) we introduce an XML access control enforcement mechanism that exploits the advantages of both pre-processing and view-based mechanisms. (2) We are the first to propose fine-grained views that are defined for each access control rule or a set of access control rules, instead of a user/role. It becomes possible for various roles to share fine-grained views, thus reduces the redundancy of data storage and also improves query performance. (3) We further introduce a cost-benefit model for fine-grained view materialization. The dynamic view management approach maximizes the utilization of available resources to obtain best query performance.

The rest of this paper is organized as follows. The background and related works are discussed in Section 2. We present our hybrid XML access control enforcement mechanism in Section 3, and the dynamic view management technique in Section 4. We show the experimental results Section 5, and then conclude the paper.

2. BACKGROUND

2.1 Related Work

The eXtensible Markup Language (XML) is a metalanguage that could be used to define customized markup languages [8]. Due to its flexibility and descriptive power, XML becomes the *de facto* standard for information sharing over the Internet. Meanwhile, the need for access control naturally arises as the XML model gets very popular for data management. In the literature, XML access control research could be roughly categorized into *access control modeling* and *enforcement mechanisms*.

An access control model defines how access control policies are specified. Some of the earlier access control models that are widely recognized are: discretionary access control (DAC), mandatory access control (MAC), and role based access control (RBAC). An introduction of legacy database access control practices could be found at [33]. In the literature, most of the XML access control approaches adopt the RBAC [34] or attribute based access control [3] to specify the users. Meanwhile, fine-grained access control is employed so that accessibility is defined at the node level. In pioneer works in XML access control, such as [4, 5], an access request is defined as a triple: the subject, the object and the access modality, which specifies whether the subject is requesting a read or a modify access. In addition to the authorization model for XML, [18] presents an XML access control language that integrates authorization; confidentiality etc. [11] defines a 5-tuple ACR where the type is L, R, LW (local weak), or RW (recursive weak). [31] proposes a different rule-function-based access control model to improve scalability and performance.

The access control enforcement mechanism implements access control models. As we have introduced, existing XML access control enforcement mechanisms could be categorized as: engine-level, view-based, pre-processing, and postprocessing. Engine-level (a.k.a. run-time) mechanisms [39, 10, 40, 15] attach an *accessibility list* to each node in the XML tree, and check the list during query processing to return only accessible nodes. Engine-level approaches require modifications to XML engine kernels and also introduces extra storage and query processing overhead. They are currently not adopted in commercial XML engines. On the other hand, view based approaches [3, 11, 37] create a separate copy of data for every role. Each view contains all the nodes that can be accessed by the role. When the user queries, the view related to that user is loaded and the queries are answered over the view. Views are relatively small (compared to the original document), and are faster to load into memory. Query processing is also faster when queries are evaluated on a much smaller XML tree. Materialized views have been employed to improve XML query performance (e.g. [38, 1]). However, as the number of views increases, excessive storage overhead is required to store the views, and it becomes difficult to maintain and update these views. In particular, although many of the roles may have similar access control rules, the views are not shared, and hence increase redundancy of the data being stored in the views.

Another important category is pre-processing approaches, including virtual view approaches. They check the queries against access control rules before evaluating them in the XML engine. Only safe queries are evaluated, while unsafe queries are either rejected or rewritten. The static analysis approach [27, 28] creates automata for queries, access control rules and the XML schema, and compares the automata to decide if a query is accepted or denied. Meanwhile, the security-view approaches [12, 19, 14] publish a schema (DTD) that only contains accessible portion of the XML document for users to write queries. User queries on the security view are then translated to equivalent queries on the original XML document and evaluated in the XML engine. QFilter [23] is another pre-processing mechanism that uses NFA structures developed from access control rules to check queries and decide if they are accepted or denied. If a query is neither completely accepted nor rejected, QFilter rewrites it to a *safe query* that only yield accessible data. Since HyXAC employs QFilter as the baseline, we will introduce more details of this approach in the next subsection. Moreover, other mechanisms in this category include: access condition table [29], secure query rewrite [26], policy matching tree [30], etc. The pre-processing approaches pass safe queries to the XML engine, therefore, no additional security check is necessary. This feature allows such approaches to be adopted by any XML engine without requiring additional changes to the query processor.

Last but not least, postprocessing approaches (e.g. [7]) evaluate the original queries against the entire XML docu-

ment to get unsafe answers. A postprocessing mechanism is employed to take access control rules and prune all the access-denied nodes from the results. This approach could be useful for streaming XML data or subscription services.

2.2 Preliminaries

2.2.1 Access control model

XML access control modeling is not the focus of this paper, hence, we adopt a relatively simple access control model used in [24]. In this model, every access control rule (ACR) consists of a 4-tuple:

$$R = \{subject, object, action, sign\}$$

where the *subject* denotes the role who is authorized (or denied) to access the data; the *object* is a set of XML nodes to be accessed; the *action* is the operation that can be performed on the object by the given subject (ex: read, write, update etc.); and the *sign* specifies whether the action can be performed on the object or not (ex: +, −: "+" specifies "access granted" and "−"specifies "access denied"). The "−" takes precedence, when there is a conflict between rules concerning the same node/set of nodes. When there is no explicitly defined rule for a node then the access to that node is denied. In this model, the object is specified using XPath [2]. XPath is a query language used for selecting nodes in a XML document. A more powerful XML query language, XQuery [6], also uses XPath to access data. In this paper, like in many other XML access control enforcement mechanisms [24, 27], we use a subset of XPath that includes child ("/") and descendent-or-self ("//") axes, wildcards ("*"), and predicates ("[]"). As an example, the following rule:

$$R_1 = \{\text{assistant}, //\text{person/name}, read, +\}$$

defines that "users of the `assistant` role are allowed to read the `<name>` child of `<person>` nodes".

2.2.2 The QFilter approach

Our HyXAC approach uses the QFilter for query preprocessing. Hence, we briefly introduce the QFilter approach. QFilter is an NFA-based implementation for XML access control. It first reads access control rules (ACR) as input, builds an NFA structure using the ACR to represent the XPath expressions that are defined as valid (i.e. access granted) by the ACR. For an incoming query Q, it is processed over the NFA to obtain one of three possible results: (1) the query is accepted as is – all of the requested nodes are accessible to the user; (2) the query is denied – none of the requested nodes is accessible to the user; and (3) the query is rewritten into a safe query Q' – some of the requested nodes are inaccessible to the user, and the new query does not request those nodes. In case (1) and (3), the output query is transmitted to the underlying XML engine to be evaluated. Let us look at an example:

Example 1: We use the popular XMark DTD and document [35], which simulates an online auction scenario. The XMark document stores item information, auction (open and closed) information as well as user information. The "Assistant" (AS) role has the following access control rules defined:

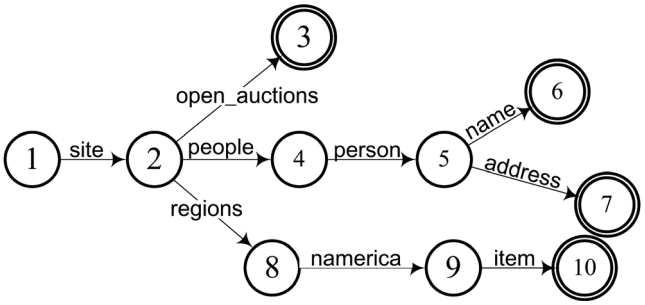


Figure 1: Example of QFilter.

```
R1: {AS, /site/open_auctions, read, +}
R2: {AS, /site/people/person/name, read, +}
R3: {AS, /site/people/person/address, read, +}
R4: {AS, /site/regions/namerica/item, read, +}
```

The corresponding QFilter NFA is shown in Figure 1. The accept states of the QFilter correspond to positive rules. For instance, accept state 3 corresponds to rule R1.

A new query "`/site/open_auctions/interval`" will be accepted at state 3[1]. Another query "`/site/regions/*/item`" will be rewritten into "`/site/regions/namerica/item`", which only requests for accessible nodes. □

For more details of the QFilter approach, please refer to [23, 24]. Moreover, it has been extended to handle multi-user scenarios in [21, 20]. Please note that although HyXAC employs QFilter for query pre-processing and sub-view definition, the concepts and mechanisms for fine-grained views, sub-view sharing and dynamic view management are all original contributions of HyXAC.

3. HYXAC: HYBRID XML ACCESS CONTROL

Conceptually, the HyXAC model is introduced in two stages. In the first stage, we add fine-grained view management to NFA-based access control enforcement, to create sub-views for distinct access control rules, and allow sub-views to be shared among roles. In the second stage, the views are dynamically materialized and cached, to get maximum query evaluation performance for limited resources.

3.1 The HyXAC framework

In this section, we introduce our new model named Hybrid XML Access Control (HyXAC). HyXAC is a hybrid model produced by the combination of a pre-processing approach (QFilter) and the view based approach.

The QFilter used in our approach is similar to the QFilter described in the previous section. For now, we only consider positive access control rules. First, a set of access control rules are used to construct the NFA structure of QFilter. When an access control rule is added, the last XPath step from the XPath will result an accept state to be added to the NFA. For instance, in Figure 1, state 6 is created and set to be an accept state, when QFilter construction process reaches the XPath step "`/name`". Therefore, every accept

[1]The *recursive* semantics is employed in the 4-tuple model – granting access to a node inherently grants access to the entire subtree. For more details, please refer to [24].

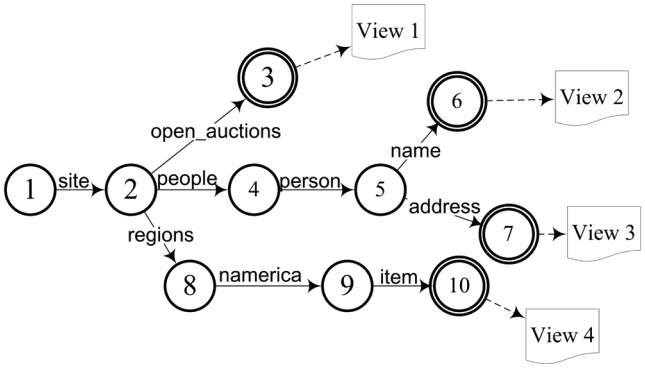


Figure 2: Example of sub-views in HyXAC.

state in the NFA corresponds to one access control rule. In HyXAC, we introduce fine-grained view management. That is, we create a sub-view for this rule, which stores the *object* node(s) of the access control rule, i.e., the subtree rooting at the XPath in the object field of the rule. However, this sub-view is not necessarily materialized. In each accept state of the NFA, the ID and status of the corresponding view is stored.

During query processing, each query is first processed in the NFA exactly the same as the QFilter approach [24]. When the last token from the query reaches an accept state in the NFA, it implies that the query, or a re-written query, is accepted by the QFilter. This query could be answered by the raw XML document, without yielding any inaccessible data. However, when the corresponding sub-view is materialized, we can exploit the view to answer the query, for better query evaluation performance.

Example 2: Let us revisit Example 1 in the previous section. If we employ HyXAC in this scenario, 4 sub-views will be created for the "Assistant" role, as shown in Figure 2. For instance, View 1 stores the subtree for "`/site/open_auctions`". Meanwhile, State 3 stores the ID of the view (View 1), as well as the status of the view: materialized, or cached in memory.

The query "`//person[@pid='18']/name`" will be accepted at State 6. If view 2 is materialized, we can answer the query using the view, which is smaller to load into memory and manipulate.

3.2 View sharing

In the conventional view-based access control enforcement mechanisms, a view is created for every role. In the above example, all four sub-views will be merged into one view, which will be used to answer queries from role "AS". However, many of the roles would have overlaps in access control rules, hence, there is a lot of redundant data across all the views.

To handle multiple roles, Multi-Role QFilter (MRQ) has been introduced in [21]. In MRQ, rules from all roles are put together to create one NFA, in which every state is attached with an access-list and an accept-list.

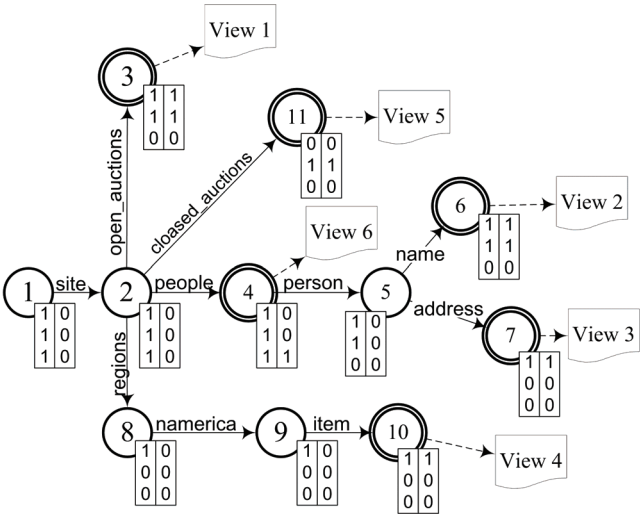


Figure 3: Example of multiple roles in HyXAC.

Example 3: We add two roles: Auction Manager (AM) and User Manager (UM), to the scenario in Example 1, with the following access control rules[2]:

```
R5: {AM, /site/open_auctions, read, +}
R6: {AM, /site/closed_auctions, read, +}
R7: {AM, /site/people/person/name, read, +}
R8: {UM, /site/people/person, read, +}
```

The MRQ for all three roles are shown in Figure 3. In particular, each state is attached with an access list and an accept list. For instance, the lists for State 4 indicate that this state is accessible to queries from all three roles, but only accept queries from role "UM". In HyXAC, 6 sub-views are created for the accept states (i.e. rules). Please note that View 6 actually contains Views 2 and 3, so that we may not need to materialize the descendent views if View 6 is materialized. We will discuss this issue later. □

The redundant storage of data is one of the drawbacks of the view-based approaches. Creating a view for every accept state (i.e. every distinct access control rule) rather than creating one for every role, answers the problem of redundant storage. Roles having identical access control rules would share accept states in the MRQ, and hence share the sub-views. For instance, if we employ conventional view-based approaches for the roles in Example 3, three views will be created, as shown in Figure 4. More duplicate data will be observed when we have more roles.

In practice, if the sub-views are materialized, they could be used to answer queries, for better query evaluation performance. In particular, fine-grained views are relatively smaller than conventional views, so that they are even faster to load into memory. In the next section, we will discuss how to select a subset of views to materialize to maximize the performance with limited resource. Before we move to sub-view management, we discuss some practical issues in HyXAC.

[2]QFilter supports wildcards and predicates, hence, they are also supported in HyXAC. For the simplicity of the illustrations, we only included simple path expressions in our examples.

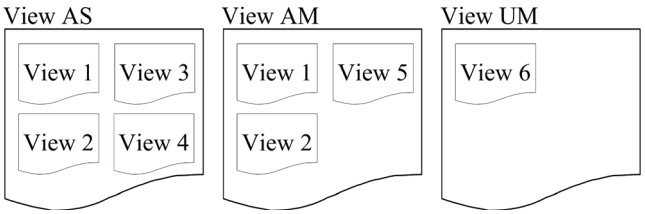


Figure 4: Conventional views for the three roles in Example 3.

3.3 Discussions

Handling negative rules. So far, we only considered positive rules – rules that grant access to a set of nodes. In [24], a separate NFA is constructed for negative rules – all the output from the positive NFA are processed in the negative NFA, and the results are connected by a *deep except* operator. In HyXAC, negative rules could be handled in two different ways: (1) same as [24], or (2) with views.

1. As introduced in [24], a separate NFA could be generated to capture negative rules. Queries accepted by the positive NFA are further processed by the negative NFA, and a "deep-except" operator is used to connect the outputs from two NFAs. The result query could be directly answered by the sub-view that is constructed from the positive rule. However, this solution is less efficient since: (1). the views are larger than the need to be, since inaccessible XML nodes are included; and 2. evaluation for queries with the deep-except operator could be slow, since recursion is needed.

2. Another approach is to remove inaccessible XML nodes from sub-views, so that queries (w/o deep-except) answered by the sub-view would not yield any inaccessible data. Consider that access is forbidden by default, negative rules are only used to revoke access to a subset of nodes that have been granted access by positive rules. Hence, each positive rule may correspond to some negative rules, while "dangling negative rules" could be ignored, since the object nodes are inaccessible to the user by default. In HyXAC, each view is created from a positive rule (or a set of positive rules), hence, we can remove the nodes identified by the corresponding negative rules from the view. In this way, when queries reach the accept state and are to be answered by the view, there will be no need to go through the negative NFA, since the view only contains authorized data.

Example 4: Let us revisit Examples 1 and 3. Assume that a negative rule is added for the role Assistant:

```
R9: {AS, /site/regions/*/item/payment, read, -}
```

The rule says that AS cannot read the `<payment>` child of the `<item>` nodes. With the negative rule, View 4 will be updated to eliminate all `<payment>` nodes. Next, a user of the AS role issues the following query:

```
/site/regions/namerica/item[quantity>15]
```

The query will be accepted (as is) by the positive NFA (shown in Figure 3) at state 10. In conventional solutions [24], the query need to further go through the negative NFA and the output will be appended to the original query with a *deep-except* operator, which is computationally more expensive. With HyXAC, View 4 could be materialized, and the accepted query from positive NFA will be directly answered the View 4. □

As we can see, in HyXAC, negative rules could be handled by the sub-views, to provide better flexibility and efficiency.

Contained views and combined views. For access control rules from different roles, the nodes referred in the object fields could have parent-child or ancestor-descendent relationships. Therefore, the corresponding views may contain each other. As shown in Example 3, the object in `R8` is the ancestor of the objects in `R2`, `R3` and `R7`. Hence, view 6 would contain views 2 and 3. In this case, we may choose to (1) drop views 2 and 3, and use view 6 to answer queries from States 6 and 7 (note that this will not jeopardize security since the queries generated by QFilter are safe). (2) Keep all three views. It is up to the database administrator to pick an option. In particular, approach (2) requires additional storage, but provides better query evaluation performance for queries from States 6 and 7.

Meanwhile, we may also choose to combine fine-grained views, if we observe that the accept list for the corresponding accept states are identical. That says, when some sub-views are accessible for the same set of roles, they could combined. In Example 3, Views 1 and 2 are both accessible for AS and AM, but not UM, hence, we could combine those two sub-views.

Queries requiring multiple views. Not all queries could be answered by sub-views. Especially, some twig queries may require access to multiple views. In Example 3, a user from the AS role may submit this query:

```
/site/people/person[name='Alice']/address
```

This query will be accepted at State 7, however, the corresponding view (view 3) does not have "`<name>`" nodes to answer the twig query. Moreover, the XML engine may not be able to join views 2 and 3 – attribute "`id`" of "`<person>`" nodes is required for join, but it may not be preserved in the views. Therefore, for queries that access nodes outside of the destination views, they need to be answered by the original document, or by the views that correspond to the ancestor states of the NFA.

The portion of queries that require multiple views highly depends on the query pattern and design of access control rules. For instance, the design of `R2` and `R3` in Example 1 is highly likely to cause queries that cannot be answered by a sub-view. Such problems could be avoided by carefully designed access control rules, as well as forcing sub-views at ancestor states in the NFA. In particular, for *logically-related rules*, it is suggested to push the sub-view to their ancestor states in the NFA, so that twig queries could be answered by the sub-view. In the above example, a sub-view could be generated for NFA state 5 (Figure 2) to handle such twig queries. However, this move may slow down query processing by introducing larger views, which take more time to load, and decrease flexibility in view management. In real-world applications, we may often observe such logically-related rules, especially with complex ACR. An optimal solution is to observe the pattern of the queries, and merge views that are often requested in the same query. On the other hand, logically-unrelated rules, such as `R3` and `R4` in the example, would not cause such problem.

4. DYNAMIC VIEW MANAGEMENT

In Section 3, we have demonstrated the HyXAC framework with fine-grained views. In this section, we will look at the dynamic implementation of the HyXAC model. The main goal of HyXAC model is to improve the query performance under constrained situations where there are limited resources available. In particular, not all the views can be materialized and stored in system memory because of memory constraints. We select the views to be materialized to maximize the query evaluation performance. We first analyze the baseline without fine-grained views, so that queries are answered by the document. We then present the case where fine-grained views are materialized and stored on the hard drive. Finally, we introduce a cost-benefit model for dynamic view caching (i.e. loading to memory).

4.1 The Baseline

In the conventional pre-processing approach (e.g. [24]), safe queries are evaluated over the original XML document. In the first stage, all the queries are passed through the QFilter to get safe queries. Let the time taken for processing a query by QFilter be t_{QF,q_i}. Assuming that there are N safe queries that passed through the QFilter, then, the time taken for processing all the N queries by the QFilter will be $\sum_{i=1}^{N} t_{QF,q_i}$. The N queries are further evaluated against the original XML document on an XML engine. The total query processing time would be: $\sum_{i=1}^{N}(t_{L(D)} + t_{D,q_i})$, where $t_{L(D)}$ denotes the time to load the XML document (we assume that documents are loaded on-the-fly), and t_{D,q_i} denotes query processing time. Note that, the document load time linearly increases with the size of the document, and it is usually significantly larger than the query evaluation time. Assume that the document does not stay in memory – it is loaded for every query, the average end-to-end query processing time for all N queries would be:

$$\overline{T}_B = \frac{\sum_{i=1}^{N}(t_{QF,q_i} + t_{D,q_i})}{N} + t_{L(D)} \quad (1)$$

The first part (QFilter processing time and query evaluation time) is relatively moderate, but the second part is generally more expensive, because loading a document from hard drive is relatively time consuming.

4.2 HyXAC with fine grained views

Nowadays, cost for secondary storage, in particular, hard drives, are getting extremely low. On the other hand, the fine-grained view management in HyXAC minimizes the overlaps between sub-views. Hence, we could assume that all the sub-views are materialized but not pre-loaded in memory – each sub-view is stored in an XML file on hard drive (we will discuss dynamic view materialization later in the section). In HyXAC query processing, all queries from the NFA will be answered by the materialized sub-view that corresponds to the accept state. Hence, the time to load a sub-view to the memory could be denoted as $t_{L(V_k)}$, where $k = GetView(q_i)$ is a function that returns the ID of the view that answers q_i. In practise, the view ID is stored in the accept state. Let the time taken to answer query q_i over this sub-view be t_{V_k,q_i}. In this approach, we first assume that the memory is cleared of the view once the query is answered. Then the sub-view that answers the next query is loaded into memory again. In HyXAC, the end-to-end query processing time for a single query q_i is given by:

$$T_{H,q_i} = t_{QF,q_i} + t_{L(V_{GetView(q_i)})} + t_{V_{GetView(q_i)},q_i}$$

For N queries, the average query processing time is:

$$\overline{T}_H = \frac{\sum_{i=1}^{N}(t_{QF,q_i} + t_{L(V_{GetView(q_i)})} + t_{V_{GetView(q_i)},q_i})}{N}$$

Assume that n_k queries are answered by view V_k, the above equation could be decomposed into:

$$\overline{T}_H = \frac{\sum_{i=1}^{N}(t_{QF,q_i} + t_{V_{GetView(q_i)},q_i})}{N} + \frac{\sum_{k=1}^{M}(t_{L(V_k)} \times n_k)}{N} \quad (2)$$

where M is the total number of views and $N = \sum_{k=1}^{M} n_k$.

The first part of $\overline{T}_H$ is slightly faster than the first part in $\overline{T}_B$ (Equation 1), since evaluating the query on a view is faster than evaluating the query on a document. The second part is significantly faster than $t_{L(D)}$ in Equation 1 – the time for loading the document is linear to the size of the document, and the view is much smaller compared with the document.

As we can see, HyXAC with fine grained views will perform better than traditional method. However, we still expect to improve it further. In the equation, the query processing by QFilter and the evaluation of the query over the views are optimized. Meanwhile, the second part (loading the view from secondary storage) can be enhanced. We came up with a new solution which can improve the average query processing time.

4.3 HyXAC with dynamic view caching

In database management systems, we always have a chunk of memory that could be used to cache data. Frequently queried tables or XML documents are temporarily kept in memory to expedite query processing. In this subsection, we introduce dynamic sub-view caching techniques to HyXAC, for better end-to-end query performance. In particular, we introduce a cost-benefit model for dynamic sub-view caching. First, we analyze the cost and benefit of caching a view V_k:

Cost. When the size of the view V_k is S_{V_k}, the cost of caching this view is a function of the size: $C_{V_k} = C(S_{V_k})$. In most cases, the cost increases linearly with the size: $C_{V_k} \propto S_{V_k}$. However, there are certain scenarios where the cost varies based on certain additional factors. For instance, in database-as-a-service scenarios, the price of renting the resources may vary. In this paper, we consider the size of the view as the cost: $C_{V_k} = S_{V_k}$.

On the other hand, the total affordable cost (e.g. total available memory) is limited to C_{max}. We assume that we cannot afford to cache all views: $\sum_{k=1}^{M} C_{V_k} > C_{max}$.

Benefit. Caching a view in memory saves the time required for loading the view every time a query needs to be answered by that view. This eliminates the loading time $t_{L(V_k)}$. So the query processing time reduces to:

$$T_{HD,q_i} = t_{QF,q_i} + t_{V_{GetView(q_i)},q_i}$$

while $b_{v_k} = \Delta t_{q_i} = t_{L(V_k)}$ is the benefit for q_i. Assuming that there are n_k queries being answered by view V_k, the

total benefit of caching the view in memory is $b_{v_k} \times n_k$. Please note that, ultimately, benefit should be modeled as the "improvement of user satisfaction". Research from usability community has shown that the frustration of the user may not increase linearly with the waiting time. For instance, when wait time gets longer, user's frustration may increase exponentially. In this paper, we adopt the simple linear model, however, the b_{V_k} in our cost-benefit model could be easily altered to fit into more complicate usability models.

With the cost-benefit model, the problem becomes: to select a subset of views, so that the total cost is less than or equal to C_{max} , while maximize the total benefit. This is a classic *0/1 Knapsack Problem*, which could be described as: given a set of items, each has a weight and a value, fill the knapsack with a subset of items, so that the combined weight is under the capacity, while total value is maximized.

In our scenario, we define a view loading vector $L = [l_1, l_2, ..., L_M]$, where $l_k \in \{0, 1\}$ indicating whether the view V_k is cached. The total cost is then denoted as:

$$C = \sum_{k=1}^{M}(C_{V_k} \times l_k) = \sum_{k=1}^{M}(S_{V_k} \times l_k) \quad (3)$$

And the total benefit is denoted as:

$$B = \sum_{k=1}^{M}(b_{V_k} \times n_k \times l_k) = \sum_{k=1}^{M}(t_{L(V_k)} \times n_k \times l_k) \quad (4)$$

Formally, our problem is to find a loading vector L, so that $C \leq C_{max}$, while B is maximized.

The knapsack problem is known to be NP-hard. Many polynomial time approximation approaches have been proposed in the literature [16, 25]. In this paper, we employ the classic *Greedy Approximation* [17], which has a $O(n)$ time complexity (assuming that the benefit-cost ratios (BCR) are already sorted), while provide good approximation when c_i is relatively small compared with C_{max}. Please note that any approximation approach of the knapsack problem could be employed in HyXAC. We choose greedy algorithm in this paper for its simplicity in presentation and relatively good performance; so that we do not deviate from the main contribution of HyXAC.

The benefit-cost ratio (BCR) is defined as the ratio of the benefit of caching a view, relative to the cost, i.e., benefit for unit cost. The BCR for caching view V_k is defined as:

$$BCR = \frac{b_{V_k} \times n_k}{C_{V_k}} = \frac{t_{L(V_k)} \times n_k}{S_{V_k}} \quad (5)$$

S_{V_k}, denoting the size of view V_k, could be easily measured. On the other hand, $t_{L(V_k)}$ is approximately linear to the size of V_k. It could be assessed with a simple experiment too. However, n_k, denoting the number of queries hitting view V_k, is not always available. We discuss three cases:

Case 1: known query pattern. Assume that we have observed the incoming query pattern (i.e., we know the fraction of queries that hit each view), and the future queries follow the same pattern as the observed queries. That is, n_k is known, and could be used to predict the number (or portion) of queries hitting view V_k in the future. In this case, we can employ the greedy approximation algorithm illustrated in Algorithm 1 to get the view loading vector $L = [l_1, l_2, ...l_M]$, in which $l_i = 1$ indicates that view i would be cached in memory, and $l_i = 0$ indicates that view i would not be cached.

Algorithm 1 Greedy approximation for HyXAC view caching with know query pattern

Require: M: the total number of sub-views
Require: $Cmax$: the maximum cost
Require: $c[1, .., M]$: the cost of sub-views
Require: $n[1, .., M]$: number of queries hitting sub-views
Require: $b[1, .., M]$: the benefit of the sub-views
1: **return** $l[1, .., M]$: the view-loading vector
2: **for** $i = 1$ to M **do**
3: $s[i] = b[i] \times n[i]/c[i]$;
4: **end for**
5: sort $s[i]$ in descending order, with index array $ind[1, ...M]$
6: $C = 0$;
7: **for** $i = 1$ to M **do**
8: $l[ind[i]] = 0$
9: **if** $C + c[ind[i]] < Cmax$ **then**
10: $l[ind[i]] = 1$
11: $C = C + c[ind[i]]$
12: **end if**
13: **end for**

Note that, in line 2, the index array contains the original view ID of the sorted views, e.g. $ind[1]$ denotes the view ID of the view with highest benefit-cost ratio. The greedy algorithm starts from Line 4: we use a loop to go through the views, from the highest BCR to the lowest. In each iteration, we check whether there is capacity to load the current view. If yes, we load the view and move to the next.

Case 2: unknown query pattern. In the case that we do not know the distribution of the queries, we make the assumption that queries come in a unified pattern. That is, the number of queries answered by view V_k is proportional to the size of the view: $n_k \propto S_{V_k}$. Therefore, we can consider the total number of queries answered by a view V_k to be $n_k = \rho \times S_{V_k}$, where ρ is the number of queries answered by unit size. The benefit-cost ratio is now modified as:

$$BCR = \frac{b_{V_k} \times \rho \times S_{V_k}}{C_{V_k}} = \rho \times t_{L(V_k)} \quad (6)$$

Therefore, the greedy approximation will start from the largest view. In each iteration, the largest "affordable" view is set to be cached, until no more affordable views are available.

Case 3: dynamic decision. A more optimized solution will be to make view-caching decisions decanically. That is, at time t, use the view patterns observed during a time window $[t - t_0], t$, and apply Algorithm 1 to select views to be cached. Views may be unloaded from memory if they are not hit by queries during the time period, while frequently queried views are "promoted" to reside in memory. In the implementation, a counter (FIFO queue) is attached at every accept state, to store the number of queries in a unit time. Every period of t_0, the decision algorithm queries the accept states to get the query distribution to decide which views to be cached.

Algorithm 2 Greedy approximation for HyXAC view caching with unified query pattern

Require: M: the total number of sub-views
Require: $Smax$: the maximum cost (total available size)
Require: $s[1,..,M]$: the size of sub-views
1: **return** $l[1,..,M]$: the view-loading vector
2: sort $s[i]$ in descending order, with index array $ind[1,...M]$
3: $S = 0$;
4: **for** $i = 1$ to M **do**
5: $l[ind[i]] = 0$
6: **if** $S + s[ind[i]] < Smax$ **then**
7: $l[ind[i]] = 1$
8: $S = S + s[ind[i]]$
9: **end if**
10: **end for**

4.4 Dynamic view materialization.

So far, we have assumed that the sub-views are all materialized and stored on hard drives. As we have introduced, with fine-grained view management, it is expected that overlaps between views are significantly reduced, when compared with conventional views. However, there still exist overlaps that cause redundancy in storage, such as Views 2, 3, 6 in Example 3 (Figure 3). As we introduced in previous section, contained views (e.g. Views 2, 3) do not need to be materialized, since the container view (View 6) could be employed to answer the queries. To further eliminate redundant storage in HyXAC, we only need to materialize the frequently queried views.

We can employ the cost-benefit analysis method, as introduced above, for dynamic view materialization. In view materialization, the benefit will be the difference between HyXAC query processing time ($\overline{T}_H$ in Equation 2) and baseline query processing time ($\overline{T}_B$ in Equation 1). The cost will be the storage for the corresponding view. Again, this becomes a knapsack problem, which could be approximated with polynomial-time solutions.

Moreover, if we combine both view materialization and view caching in one dynamic process, it becomes the classic *generalized assignment problem*. The generalized assignment problem could be described as: given N items and M bins, for each item x_i against each bin b_j, there is a weight $w_{i,j}$ and a value (also called profit) $v_{i,j}$. The goal is to assign items into bins so that (1) the total weights in each bin is under the capacity ω_j of the bin, and (2) the total value is maximized. The dynamic view materialization and caching problem in HyXAC is a generalized assignment problem with 2 bins ($M = 2$). The generalized assignment problem itself is known to be NP-hard and APX-hard. Various approximation methods have been proposed in the literature, e.g. [36, 32]; a survey is available at [9]. In HyXAC, as $M = 2$ and $w_{i,j}$ is much smaller than ω_j, a simple greedy approximation is expected to achieve relatively good result. In this paper, we do not go into further details of the 2-bin optimization problem, so that we do not deviate from the focus on access control.

5. EXPERIMENTAL RESULTS

Experiment settings. To demonstrate the effectiveness of our approach, we have performed intensive experiments to compare HyXAC and conventional approaches. First, we use an implementation of QFilter in [24]. We further implement HyXAC on top of QFilter in Java. We use Galax [13] for query evaluation, and use Galax's Java API to communicate between HyXAC and Galax.

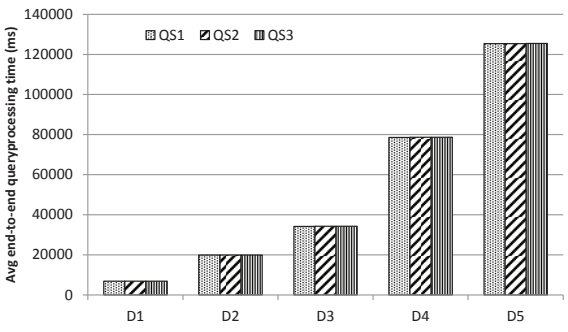


Figure 5: End-to-end query processing time for baseline approach.

We use the XMark benchmark [35] to generate XML documents of various sizes. We construct two sets of access control rules: each set contains approximately 10 rules, with RS1 grants access to a larger portion of the XML trees (portions of accessible nodes vary for different documents). Meanwhile, we construct 3 sets of XPath queries, with 250, 500, and 1000 queries (with different probabilities for wildcards in the XPath), respectively. Every query is submitted on behalf of both roles, and the average query processing time is recorded. Note that we do not include "access-denied" queries since they are rejected by the NFA and are not processed by the XML DB. Meanwhile, a very small portion of accepted twig queries require multiple sub-views in the evaluation, we eliminated such queries as well.

Baseline. The baseline is the conventional pre-processing approach. We use Multi-Role QFilter (MRQ) to process incoming queries. The output safe queries are then submitted to Galax, which loads the XML documents from hard drive, and evaluate the queries against the document. We used 5 documents: approximately 5 MB (MBytes), 11 MB, 25 MB, 43 MB and 52 MB, respectively. We assume that the available memory space is sufficient to load the queried document in its entirety, but cannot hold all of them in long-term. That is, documents are loaded on-the-fly during query processing. End-to-end query processing times are shown in Figure 5. The results confirm our assumption that query processing times are dominated by document loading and parsing times, and appear to be linear to the size of the documents.

HyXAC with fine grained views. In this experiment, we assume that fine-grained views are constructed and stored on hard drive, for every accept state in the MRQ constructed in the baseline approach. The queries are answered by the views, instead of the documents. Therefore, the document loading in the baseline approach is replaced by view loading. It is undoubtable that HyXAC significantly outperforms baseline when the document size increases. To demonstrate the advantage of HyXAC even with a smaller document (so that the constant overhead of MRQ and view management becomes relatively significant), we use the 5MB

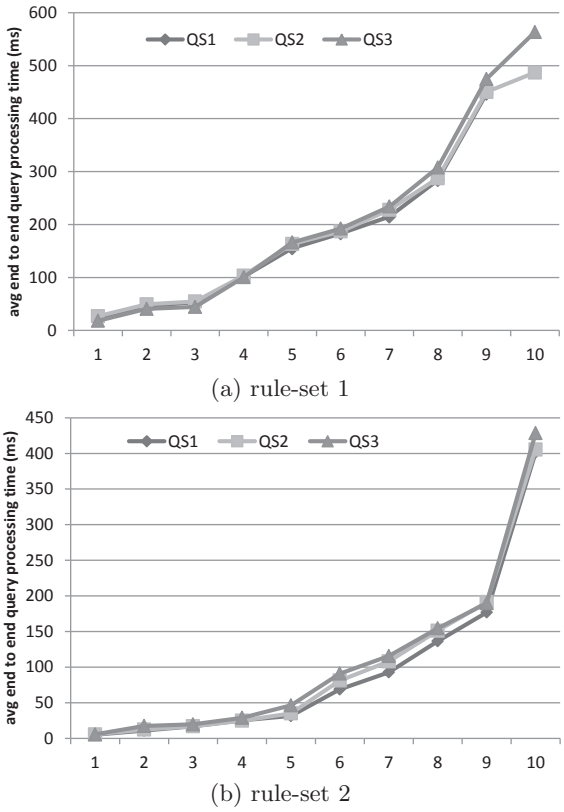


Figure 6: End-to-end query processing times for HyXAC with fine-grained views.

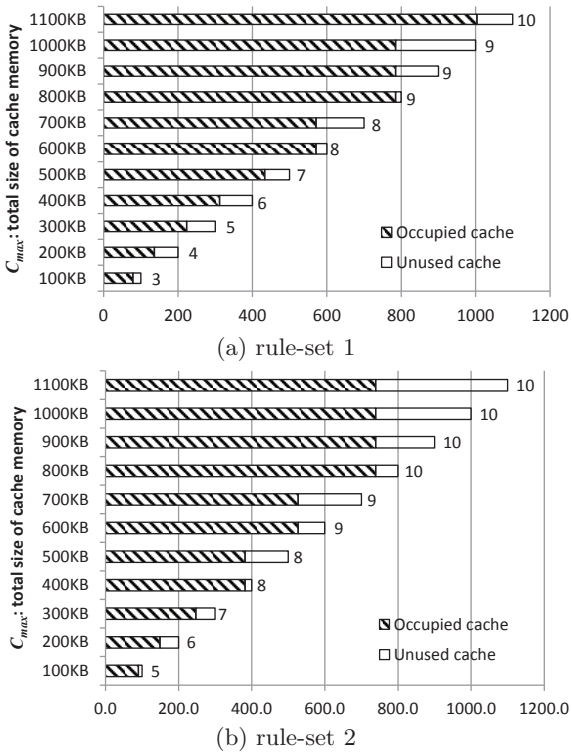


Figure 7: HyXAC with dynamic view management: cache memory utilization ratio. Numbers of cached sub-views are shown on the right of each bar.

XML document in the rest of the experiments. Role 1 has access to approximately 20% of the document (by size), while role 2 has access to approximately 15% of the document.

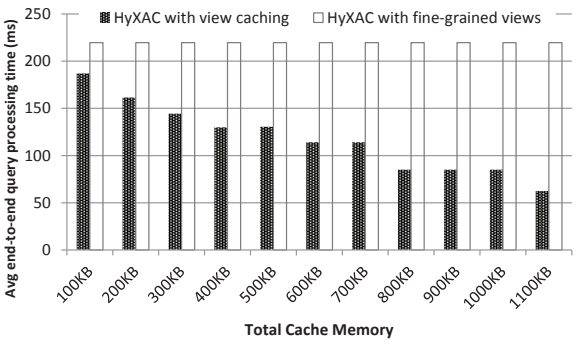
For each role, we sort the sub-views based on their costs (i.e. sizes), and record the average query processing times for queries hitting each sub-view. End-to-end query processing times for both roles are shown in Figure 6. Again, query processing time is mostly dominated by view loading and parsing times.

As we can see from the results, end-to-end query processing time is approximately linear to the size of the document. As the sizes of sub-views are significantly smaller than the original document, we have observed 12x to 600x improvement of query processing performance (compared with original QFilter approach). Meanwhile, if we utilize conventional view-based approaches, the view for each rule-set would be the combination of all sub-views for the RS. Compared with conventional views, the basic HyXAC approach could achieve approximately 3x-150x performance improvement.

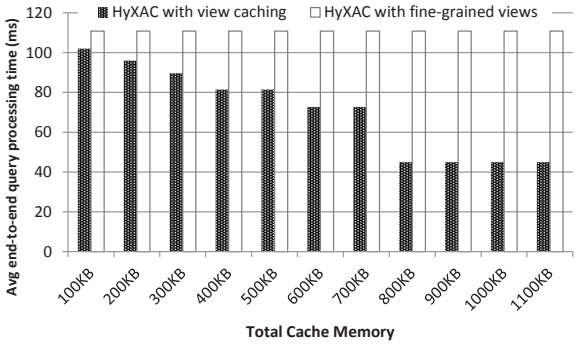
HyXAC with dynamic views caching. In this experiment, we assume that a small chunk of memory (C_{max} KB) is available to cache the sub-views. We assume that the query pattern is known, and then employ the cost-benefit model to identify the views to be cached. We use the greedy approximation introduced in Algorithm 1. We conduct the experiments for two roles separately, to better illustrate incremental view caching. Figure 7 demonstrates the memory utilization ratio, when the available cache memory varies from 100KB to 1100KB. Please note that 4 out of 10 views are shared between both roles, and 1400KB of cache is enough to host all views for two roles.

Next, we assess the query evaluation performance, with dynamic view caching. We submit queries from all query sets on behalf of each role, and record the average end-to-end query processing time. Again, the greedy approximation algorithm in Algorithm 1 was implemented to identify views to be cached, while the available memory for view caching varies from 100KB to 1100KB. The results are shown in Figure 8. Note that the white bars demonstrate the average query evaluation time when none of the views are cached, just for comparison. As expected, query evaluation performance improves when more views are cached in memory. In particular, when all the views are loaded in memory, the query processing time denotes the time for XML engine to evaluate the XPath queries and generate results, which cannot be further optimized outside the XML engine.

Discussions. In the experiments, we assumed that the XML engine does not have intelligent memory management – no built-in document caching capabilities. This is true for Galax and many other open source XML database engines, to the best of our knowledge. For instance, in Galax, `Galax.loadDocument()` needs to be explicitly invoked to load the document in memory. Meanwhile, for XML engines that could cache frequently queried XML documents, HyXAC still has significant advantages due to fine-grained view management. Smaller sub-views in HyXAC provides more flexibility for document caching: (1) memory utilization would be improved due to the smaller size of the views; (2) cache



(a) rule-set 1



(b) rule-set 2

Figure 8: HyXAC with dynamic view management: average end-to-end query processing time with various cache memory size.

hit ratio is expected to be better again due to fine-grained views – it becomes possible to remove unused portion of the XML document from memory; and (3) cache update becomes more efficient since I/O is faster with smaller sub-views.

6. CONCLUSION

In this paper, we present HyXAC, a hybrid XML access control enforcement mechanism. HyXAC first employs the pre-processing approach QFilter to process queries. With fine-grained view management, a sub-view is defined for each access control rule (i.e. an accept state in the NFA), and queries accepted by the access control rule are answered by the corresponding sub-view. In this way, views are not defined in a per-role basis, so that overlaps among views are minimized, and sub-views are shared among roles. Furthermore, the dynamic view management mechanism allocates the available resources (memory and secondary storage) to materialize and cache sub-views to improve query performance. Experimental results show that HyXAC optimizes the usage of system resource, and improves the performance of query processing.

7. ACKNOWLEDGEMENTS

Bo Luo is partially supported by NSF OIA-1028098, and University of Kansas General Research Fund (GRF-2301677). The authors would like to thank the anonymous reviewers for their valuable comments that helped improve the quality of the paper.

8. REFERENCES

[1] A. Balmin, F. Özcan, K. S. Beyer, R. J. Cochrane, and H. Pirahesh. A framework for using materialized xpath views in xml query processing. In *VLDB*, pages 60–71. VLDB Endowment, 2004.

[2] A. Berglund, S. Boag, D. Chamberlin, M. F. Fernández, M. Kay, J. Robie, and J. Simeon. "XML Path Language (XPath) 2.0". W3C Working Draft, Nov. 2003. http://www.w3.org/TR/xpath20.

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

[4] E. Bertino, S. Castano, E. Ferrari, and M. Mesiti. Specifying and enforcing access control policies for XML document sources. *World Wide Web*, 3(3):139–151, 2000.

[5] E. Bertino and E. Ferrari. Secure and selective dissemination of XML documents. *ACM Trans. Inf. Syst. Secur.*, 5(3):290–331, 2002.

[6] S. Boag, D. Chamberlin, M. F. Fernández, D. Florescu, J. Robie, and J. Simeon. "XQuery 1.0: An XML Query Language". W3C Working Draft, Nov. 2003. http://www.w3.org/TR/xquery.

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

[8] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, F. Yergeau, and J. Cowan. XML 1.1 (Second Edition). W3C Recommendation, Aug. 2006. http://www.w3.org/TR/2006/REC-xml11-20060816/.

[9] D. Cattrysse and L. Van Wassenhove. A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, 60(3):260–272, 1992.

[10] S. Cho, S. Amer-Yahia, L. V. Lakshmanan, and D. Srivastava. "Optimizing the Secure Evaluation of Twig Queries". In *VLDB*, Aug. 2002.

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

[12] W. Fan, C.-Y. Chan, and M. Garofalakis. Secure xml querying with security views. In *SIGMOD*, pages 587–598, 2004.

[13] M. Fernandez and J. Simeon. Galax, 2009. http://galax.sourceforge.net/.

[14] J. Foster, B. Pierce, and S. Zdancewic. Updatable security views. In *Computer Security Foundations Symposium, 2009. CSF '09. 22nd IEEE*, pages 60 –74, july 2009.

[15] M. Jiang and A. W.-C. Fu. Integration and efficient lookup of compressed xml accessibility maps. *IEEE Transactions on Knowledge and Data Engineering*, 17(7):939–953, 2005.

[16] H. Kellerer and U. Pferschy. A new fully polynomial time approximation scheme for the knapsack problem. *Journal of Combinatorial Optimization*, 3:59–71, 1999.

[17] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer, 2004.

[18] M. Kudo and S. Hada. Xml document security based on provisional authorization. In *ACM CCS*, pages 87–96, 2000.

[19] G. Kuper, F. Massacci, and N. Rassadko. Generalized xml security views. In *SACMAT*, pages 77–84, 2005.

[20] F. Li, B. Luo, P. Liu, D. Lee, and C.-H. Chu. Automaton segmentation: a new approach to preserve privacy in xml information brokering. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, pages 508–518, 2007.

[21] F. Li, B. Luo, P. Liu, D. Lee, P. Mitra, W.-C. Lee, and C.-H. Chu. In-broker access control: Towards efficient end-to-end performance of information brokerage systems. In *IEEE SUTC'06*, pages 252–259, 2006.

[22] B. Luo, D. Lee, W.-C. Lee, and P. Liu. "A Flexible Framework for Architecting XML Access Control Enforcement Mechanisms". In *VLDB Workshop on Secure Data Management in a Connected World (SDM)*, Toronto, Canada, Aug. 2004.

[23] B. Luo, D. Lee, W.-C. Lee, and P. Liu. "QFilter: Fine-Grained Run-Time XML Access Control via NFA-based Query Rewriting". In *ACM CIKM' 2004*, Washington D.C., USA, Nov. 2004.

[24] B. Luo, D. Lee, W.-C. Lee, and P. Liu. Qfilter: Rewriting insecure xml queries to secure ones using non-deterministic finite automata. *The VLDB Journal*, 20(3), 2011.

[25] M. Magazine and O. Oguz. A fully polynomial approximation algorithm for the 0Íc1 knapsack problem. *European Journal of Operational Research*, 8(3):270 – 273, 1981.

[26] S. Mohan, A. Sengupta, and Y. Wu. Access control for xml: a dynamic query rewriting approach. In *ACM CIKM*, pages 251–252, 2005.

[27] M. Murata, A. Tozawa, M. Kudo, and S. Hada. Xml access control using static analysis. In *ACM CCS*, pages 73–84, 2003.

[28] M. Murata, A. Tozawa, M. Kudo, and S. Hada. Xml access control using static analysis. *ACM Trans. Inf. Syst. Secur.*, 9(3):292–324, 2006.

[29] N. Qi and M. Kudo. Access-condition-table-driven access control for xml databases. In P. Samarati, P. Y. A. Ryan, D. Gollmann, and R. Molva, editors, *ESORICS*, volume 3193 of *Lecture Notes in Computer Science*, pages 17–32. Springer, 2004.

[30] N. Qi and M. Kudo. Xml access control with policy matching tree. In *ESORICS 2005, 10th European Symposium on Research in Computer Security*, pages 3–23, 2005.

[31] N. Qi, M. Kudo, J. Myllymaki, and H. Pirahesh. A function-based access control model for xml databases. In *ACM CIKM*, pages 115–122, 2005.

[32] G. Ross and R. Soland. A branch and bound algorithm for the generalized assignment problem. *Mathematical programming*, 8(1):91–103, 1975.

[33] R. Sandhu and P. Samarati. Access control: principle and practice. *Communications Magazine, IEEE*, 32(9):40 –48, sept. 1994.

[34] R. S. Sandhu, E. J. Coyne, H. L. Feinstein, and C. E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–47, 1996.

[35] 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.

[36] D. Shmoys and É. Tardos. An approximation algorithm for the generalized assignment problem. *Mathematical Programming*, 62(1):461–474, 1993.

[37] A. Stoica and C. Farkas. Secure xml views. In E. Gudes and S. Shenoi, editors, *DBSec*, volume 256 of *IFIP Conference Proceedings*, pages 133–146. Kluwer, 2002.

[38] X. Wu, D. Theodoratos, and W. H. Wang. Answering xml queries using materialized views revisited. In *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, pages 475–484, 2009.

[39] T. Yu, D. Srivastava, L. V. S. Lakshmanan, and H. V. Jagadish. Compressed accessibility map: Efficient access control for XML. In *VLDB*, pages 478–489, China, 2002.

[40] H. Zhang, N. Zhang, K. Salem, and D. Zhuo. Compact access control labeling for efficient secure xml query evaluation. *Data Knowl. Eng.*, 60(2):326–344, 2007.