A System Architecture for History-Based Access Control for XML Documents

Patrick Röder*, Omid Tafreschi**, Fredrik Mellgren, and Claudia Eckert

Darmstadt University of Technology,
Department of Computer Science,
D-64289 Darmstadt, Germany
{roeder,tafreschi,mellgren,eckert}@sec.informatik.tu-darmstadt.de

Abstract. In this paper, we present a history-based model which considers not only the content of an XML document to define access, but also how this content was created. The last aspect is an important factor for access control. Within the proposed model, the creation of documents is stored in histories, which also contain the source and destination of copied document parts. This enables us to define access depending on the origin of document parts. Applying this model in an environment where multiple users can edit documents concurrently is a challenging task, since access decisions depend on other documents, which are possibly edited at the same time. For this purpose, we present a system architecture which supports an efficient workflow and reduces the overhead for determining access rights of documents depending on other documents.

Keywords: Access Control, Document Security, XML, XPath.

1 Introduction

In the modern business world, many IT systems use XML as a standard for information storage and exchange. In such systems, security is crucial, since unauthorized access and information theft are responsible for a major part of damages caused by computer crime [9]. Access control (AC) is a central security mechanism to reduce that kind of loss. Although much work on AC in the areas of file systems or relational databases has already been done, defining access to XML documents is a different issue as stated in [7].

Consequently, a large number of models for AC for XML documents were proposed [2,5,6,8,13]. These approaches consider the content of a document to define access to its parts. This leads to a flexible way of defining policies independently of concrete instances. However, theses approaches do not regard how the content of an XML document was created. But this is important for AC, e.g., if the source of a copied part of a document is a top secret document, access to

* The author is supported by the PhD program *Enabling Technologies for Electronic Commerce* of the German Research Foundation (DFG).

** The author is supported by the German Federal Ministry of Education and Research under grant 01AK706C, project *Premium*.

that part has to be restricted, too. A similar situation arises when a document part is copied to a top secret document, e.g., a patent application. In this case, it is desirable to deny access to the document parts located in the source document to avoid information disclosure. Additionally, it is important to know who has modified a document. Consider the following example: A researcher can change the title of a section, e.g., to make a suggestion, until the title is changed by a senior researcher, who has the authority to declare a title as final. Moreover, to enable Chinese Wall policies [3], which are important in the financial domain, the knowledge about previously performed operations is required.

Since in some cases XML elements contain a large amount of data, the granularity of AC on the level of XML elements is too coarse. An XML element can be composed of text parts from different sources. In this case, the AC system must be able to consider these parts individually to increase both flexibility and usability.

For these reasons, we introduced a model in [14] that is capable of defining access based on the content of the current document, the recorded histories and the content of dependent documents. These are documents between which documents parts have been copied to or from the current document. The histories contain information about the operations that led to the current document state. Moreover, these histories also include the source and destination of copied document parts. We use this information to define access.

Applying our model in a scenario where multiple users concurrently edit multiple documents introduces four challenges. First, since access rights of one document depend on other documents, we need a method for accessing these distributed documents when calculating access rights. Second, changes to one document require the recalculation of the views of all dependent documents, which are currently viewed. The straight forward approach for this is to recalculate the views of all dependent documents after a document has been changed. However, this results in a much higher number of view recalculations compared to models which only define access depending on the current edited document. For example, editing 20 depending documents concurrently, leads to a 20 times higher number of view recalculations with the straight forward approach. Therefore, we need a method which reduces the number of these view recalculations. Third, the changes of one user to a document can revoke the access rights of other users which are currently editing dependent documents. As a consequence, access rights can be revoked during an editing process, which can lead to conflicts regarding the content of the document and the access rights. Consequently, we need a method for handling these conflicts. Fourth, aforementioned the straight forward approach causes intermediate editing steps to become relevant for access decisions of other users, which is not desired. For example, a user can change a policy relevant element of a document by first deleting it and then replacing it with an updated version afterwards. In this example, the first step can revoke the access rights of another user, whereas the second step might restore these access rights.

The remainder of this paper is organized as follows: We summarize our model for AC for XML documents in Section 2. In Section 3, we present a system architecture that solves the four challenges mentioned above. Section 4 presents related work. We conclude and discuss future work in Section 5.

2 Model

In this section, we give an overview of our model and its components, which are explained in the following sections. We start with a description of the histories, continue with the operations defined in our model and finally present the syntax of our access rules.

2.1 Histories

We use histories to keep track of changes caused by the operations `create`, `copy`, `delete`, and `change attribute`. The operation `view` is also logged in the histories. These operations are described in detail in Section 2.2. We keep one history for every element itself including its attributes and one history for its text content. The latter history uses markup elements to divide the text into text blocks with a block ID. This mechanism enables us to keep track of sub-elements of arbitrary size. The markup elements are defined by `ac:block` elements, where `ac` is the prefix for the namespace of the access control system. We use the block IDs to reference individual text blocks in the history for the text content. If a view is created for a user, the `ac:block` elements are omitted. Keeping track of such implicitly defined sub-elements allows us to manage protection units smaller than an XML element. Technically, we use XML elements to define those sub-elements, but from a user's point of view, these sub-elements are not visible.

A new text block is created in two cases. First, we create a new text block as a result of copy operations, at both the source and the destination element. Second, we create a new text block whenever text is added to an element.

In addition to the histories, we maintain a unique element ID for each element to reference it independently of its current position within the document. Moreover, each document has a unique document ID. We use these IDs to keep track of copy operations by maintaining an *is-copy-of* relation among the elements and text blocks. Two objects are in is-copy-of relation with each other if one object is a copy of the other.

A history entry consists of an action element, which contains details on the operation and a context description. In addition to the operation, an action element can have up to two arguments that describe the action. For the actions related to attributes, we store the name of the corresponding attribute. The `change attribute` and `create attribute` operations additionally store the new value of the attribute. The `create text` and `delete text` operations store the block ID of the corresponding text block.

2.2 Operations

In this section, we describe the details of the operations supported by our model. These are `view`, `create`, `delete`, `change attribute` and `copy`. Most of the operations can be applied to elements, text and attributes. Each operation, except for `view`, has an effect on the document itself as well as on the histories. The `view` operation creates a history entry only. The `create` operation is divided into creating elements, creating attributes and creating text.

The `create element` operation creates an element without any attributes or text. In addition to the element itself, the history of the element is created. The first entry of the history for the element describes its creation. The attributes of an element are created with the `create attribute` operation, which is also logged with an entry in the history of the enclosing element. It can be required that elements have mandatory attributes. This requirement should be checked on the application level and not within the access control system. This also applies to the deletion of mandatory attributes.

The `create text` operation is used to add new text to an element. This operation has an argument that specifies the position of the new text. If this position is within an existing block, this block is split at the position where the new content should be placed and the new content is placed in-between the split blocks. The split blocks keep their original histories, whereas the new content gets a new history with one entry describing its creation. The boundaries of the split content pieces are denoted by the `ac:block` elements, as described in Section 2.1.

The `delete` operation is used to delete elements, attributes, text or parts of the text. Since elements and their attributes are checked in rules, we need to keep them after deletion. For that purpose, the context of a delete operation is captured in the element history with a delete action entry. A context is a tuple of `Date`, `Subject` and `Role`, where `Date` refers to a date including time and `Role` is the role of the `Subject` that performs the corresponding operation.

The `copy` operation is used for elements, text or parts of the text. In all cases, we apply the corresponding `create` operation to create a new instance at the destination as a copy from the source, which is registered in the destination element. Additionally, the is-copy-of relation of the elements is updated.

The `view` operation displays elements which have not been deleted. When a user wants to view a document, the `view` operation is invoked for every element of the document itself, but also for its attributes and text. In contrast to the read operation of some other systems, e.g., [1,3], the view operation does not imply a data transfer.

The `change attribute` operation allows users to change the value of a specific attribute. Since former values of an attribute can be checked by rules, we record the change with an entry in the element history.

2.3 Rules

In this section, we define a syntax for AC rules, which can express policies that depend on the content of the current document, the recorded history information and the content of dependent documents. Generally speaking, a rule has the typical structure of subject, operation, object and mode. The `mode` field of a rule defines whether it is positive (allow) or negative (deny). The default semantics of our model is deny: if the access to the object is neither allowed nor denied by a rule, then the object is not accessible. If conflicts occur, we take the rule with the superior role and finally apply "deny takes precedence over allow". We use roles [15] to model the subjects to gain a higher level of abstraction and therefore more flexibility compared to directly listing individual subjects.

Instead of listing individual objects in rules in an ACL-like manner [10], we describe objects by their properties, e.g., location within a document or attribute values. For this purpose, we use XPath patterns [4] to describe the objects for which a rule is applicable. We use XPath, since its clearly defined semantics makes the interpretation of the resulting rules unambiguous. Moreover, XPath has a predefined set of mechanisms that can be used for our purpose, which simplifies the implementation of our model.

We define two types of rules. The first type of rule defines permissions for the unary operations `create`, `view`, `delete` and `change attribute`. The objects of an AC rule are defined by an XPath pattern. The second type of rule defines permissions for the binary `copy` operation, which requires the specification of a source and a destination object. We use two XPath patterns for this. The syntax of both types of rules is listed in Figure 1.

Unary rule		*Copy rule*	
Element	Description	Element	Description
Role	Role	Role	Role
Operation	Operation	Operation	"Copy"
Object	XPath	Object	XPath
		Destination	XPath
Mode	allow | deny	Mode	allow | deny

Fig. 1. Syntax of AC rules

2.4 Accessing History Information with XPath

We use XPath patterns in rules to define access depending on histories. As a consequence, we need a mechanism to access the histories within an XPath pattern. Therefore, we extend the function library of XPath by a set of functions, which we collect in the following six groups. The namespace of our functions is indicated by the prefix '`ac:`'. In the context of XPath, we speak of a node instead of an object.

Getting Copies of a Node. This group of functions is related to the is-copy-of relation of nodes among each other. It is required to express rules that define access depending on the source of an object or on the locations to where an object was copied.

The function `ac:copies` returns all nodes that are in is-copy-of relation with the current node, whereas the function `ac:predecessors` returns all nodes of which the current node is a copy. Finally, the function `ac:successors` returns all nodes that are copies of the current node. All three functions also return nodes that are in indirect is-copy-of relation to the current node, e.g., `ac:successors` also returns the copies of the copies of the current node.

Getting Attribute Values. The function `ac:attribute-values` returns a chronologically sorted list of tuples of an attribute value and the context corresponding to the change of the attribute value. It is required to define rules, which inspect former values of an attribute. For example, the rule {`researcherB`, `View`, `deny`, `/Report[count(ac:attribute-values('funded-by')[value='Company A']) > 0]/*`} states that subjects in the role `researcherB` are not allowed to view reports that were funded by `'Company A'` in the past or at present.

Getting Related Nodes Depending on Time. This group of functions retrieves nodes addressed relatively to the context node that existed within a specified time interval. In the XPath terminology, the element to be checked against the pattern is called the *context node*. XPath offers functions to retrieve nodes addressed relatively to the context node, but without the specification of a time interval, since XPath only considers the current state of a document. This time interval is required to select related nodes depending on time, since nodes can be deleted. Therefore, each of these functions can have a time interval as parameter, e.g., `ac:children-at(t1, t2)` returns all nodes that were children of the context node in the time interval between `t1` and `t2`. To inspect a single point in time, `t2` can be omitted. The functions of this group are `ac:parent-at`, `ac:following-at`, `ac:preceding-sibling-at`, `ac:preceding-at`, `ac:following-sibling-at`, `ac:children-at`, `ac:descendant-at`, `ac:root-at` and `ac:self-at`.

Getting the Context of a History Entry. This group of functions offers access to the context of a specific history entry. Each function returns an element consisting of subject, role and time. These functions are `ac:creation-context` and `ac:deletion-context`.

Getting Accessed Nodes. This group of functions is used to get all nodes which have been accessed by a specified user or by a user in a certain role. For example, these functions are required to express Chinese Wall policies [3]. The functions are `ac:created`, `ac:viewed`, `ac:changed-attribute` and `ac:deleted`. Each function refers to a specific operation, e.g., `ac:viewed` returns viewed nodes. In addition, the function `ac:accessed` returns all accessed nodes independently of the operation. All functions have two parameters that define conditions on the returned nodes. The first parameter `user` specifies to return only nodes that have been accessed by the specified user. Analogously, we define the parameter `role`. Both parameters can be set to `any` to indicate to return nodes accessed by any user or in any role. Optionally, each parameter can be set to `current`. In this case, the current user or his current role is used for the check. For example, `created(any, current)` returns all nodes which have been created by users who were active in the same role as the one in which the current user is active in.

Getting Specific Nodes of Current Rule. We define three functions for accessing specific nodes within an XPath pattern. The function `ac:current--node` returns the node in question for which the XPath pattern is evaluated. This function is required when the pattern's context changes to a document that is different from the document for which the pattern was initiated. The function `ac:src-node` retrieves the source node in question when checking a copy rule. In a similar fashion, the function `ac:dest-node` returns the destination node of a copy rule. The last two functions are necessary to define copy rules which compare the source and destination objects with each other.

3 System Architecture

In this section, we present a system architecture for applying history-based AC in an environment where multiple users can edit documents concurrently. Its components are explained in the following sections. Additionally, we describe the algorithms and protocols that are required for the interaction between the components.

3.1 Architecture Overview

Our system architecture and its components are depicted in Figure 2. Our system uses four databases. The document database (Doc DB) contains all documents of the system. The rule database (Rule DB) contains the AC rules, which specify allowed or denied accesses to the documents and their parts. The copy database (Copy DB) stores the is-copy-of relation of the objects. Since the is-copy-of relation can be depicted by a graph, we speak of an edge when we refer to a single is-copy-of relation between two objects. Finally, the user database (User DB) stores the credentials of the users of the system as well as the corresponding roles including their hierarchy.

The user interface (UI) presents documents to the user and offers operations that can be performed on the documents. If the user invokes such an operation, the corresponding request is sent to the document processor (DP), which

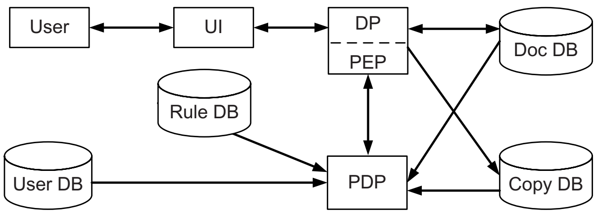

Fig. 2. System architecture

performs the requested operation if it is permitted. Inside the DP, the policy enforcement point (PEP) intercepts each operation and asks the policy decision point (PDP) whether the requested operation is allowed. The PDP uses the four databases to decide whether to allow or deny the requested operation. This architecture allows us to access distributed documents when a rule is evaluated and therefore it represents a solution for the first challenge mentioned in the introduction and. In the following, we explain the workflow for editing a document to illustrate the processes within our architecture.

3.2 Workflow

A document must be opened before it can be viewed or edited. Therefore, the UI offers a command to open a document. This command is sent to the DP, which loads a copy of the document from the document database. We refer to this process as *check-out*, since it has semantics similar to the check-out command of a version control system [16]. After the check-out, the user can edit the document by applying the operations of our model. The changed content of an opened document including the corresponding histories becomes relevant for access decisions of other documents after it is checked-in. Up to then, the content of the opened document is only relevant for access decisions concerning that document itself. The document and the corresponding histories are kept as a local copy in the DP. To *check-in* a document, the user must invoke the corresponding command of the UI. Then, the DP stores the copy of the document back to the document database.

The check-in and check-out concept is more efficient and offers a higher usability compared to directly working on the policy-relevant version of a document. The first concept is more efficient, because changed content must be propagated less often, i.e., only when a document is checked-in compared with immediately after each change. This also reduces the overhead for recalculating permissions. The usability is also higher, because of the transaction semantics of the approach. With this concept a user can decide when the changing of a document is done, instead of having potentially unwanted intermediate states to get relevant for access decisions. With this concept we give a solution for the second and fourth challenge mentioned in the introduction.

Check-Out. When a user invokes the command to check-out a document, the DP first loads a copy of that document from the Doc DB. The Doc DB maintains a list for each document that denotes by which users the corresponding document is currently opened to support concurrent access to documents. The PDP executes Algorithm 1 to create a view. This algorithm removes nodes from the document for which the user in question has no view permission and deleted nodes. For that purpose, the algorithm adds a marker to each node which is set initially to “default”, where a node can either be an element, an attribute or a text block. Next, we sort all rules by their role and their mode. More special roles are priorized over less special roles and deny rules are placed before allow rules.

Then, we remove inapplicable rules. For each of the remaining rules, the corresponding XPath pattern is evaluated. The result of this step is a set of nodes that match with the current XPath pattern, which defines the applicable objects of the rule. For each of these nodes, the marker is set according to the mode field of the current rule. If all nodes have a marker different from "default" we stop inspecting rules. Finally, we remove every node with a marker set to "default", every node with a marker set to "deny" and deleted nodes. After that, the PDP sends the view to the DP, which creates history entries for the view operation and forwards the view to the UI.

Algorithm 1. Create View

```
Input  : $rules_{all}$, $role_{curr}$, role_hierarchy, doc
Output: doc
1  add marker to every node of doc
2  set marker of every node of doc to "default"
3  sort $rules_{all}$ by role (special first) and mode (deny first)
4  for each $rule_i$ of $rules_{all}$ do
5      if operation of $rule_i$ is not "view" or role of $rule_i$ is not inferior or equal to
       $role_{curr}$ then
6          continue with next iteration of loop
7      $nodes_{result}$ ← evaluate XPath of $rule_i$ for doc
8      for each $node_j$ of $nodes_{result}$ do
9          if marker of $node_j$ is "default" then
10             set marker of $node_j$ to mode of $rule_i$
11     if all markers of doc are different from "default" then
12         exit loop
13 for each $node_j$ of doc do
14     if marker of $node_j$ is "default" or "deny" or the node is deleted then
15         remove $node_j$ and subtree below from doc
16 return doc
```

Editing. To edit a document, the user first selects an operation offered by the UI. This operation is sent to the DP, where the PEP intercepts the operation to check whether it is allowed. For this purpose, the PEP sends the requested operation together with the current document to the PDP, which evaluates the rules to answer the request of the PEP. For this purpose, the PDP performs the Algorithm 2.

The algorithm for rule evaluation sorts all rules like the previous algorithm. Then, it checks the applicability of each rule by inspecting its role and its operation. For each rule, the XPath pattern is evaluated to check whether it matches with the object in question. In case of a copy operation, the XPath pattern for the destination is evaluated, too. If the rule is applicable, its mode is returned. After evaluating all rules, the algorithms returns "deny", if none of the rules was applicable. The PDP sends the result of this algorithm back to the DP. If the result is deny, the DP does not perform the requested operation and informs the user via the UI. If the result is allow, the DP performs the requested operation.

Algorithm 2. Evaluate Rules

```
Input  : $rules_{all}$, $role_{curr}$, role_hierarchy, op, doc, obj, $doc_{dest}$, $obj_{dest}$
Output: deny | allow
1  sort $rules_{all}$ by role (special first) and mode (deny first)
2  for each $rule_i$ of $rules_{all}$ do
3      if operation of $rule_i$ is not op or role of $rule_i$ is not inferior or equal to
       $role_{curr}$ then
4          continue with next iteration of loop
5      if op is "copy" then
6          $nodes_{result} \leftarrow$ evaluate XPath for source of $rule_i$ for doc
7      else
8          $nodes_{result} \leftarrow$ evaluate XPath of $rule_i$ for doc
9      if obj is not contained in $nodes_{result}$ then
10         continue with next iteration of loop
11     if op is "copy" then
12         $nodes_{result} \leftarrow$ evaluate XPath for destination of $rule_i$ for $doc_{dest}$
13         if $obj_{dest}$ is not contained in $nodes_{result}$ then
14             continue with next iteration of loop
15     return mode of $rule_i$
16 return "deny"
```

Check-In. A user can invoke the check-in command of the UI to save his changes to an opened document doc_a, which is currently stored only within the DP, to the Doc DB. As a result of this, the checked-in version of the document becomes relevant for the access decisions of other documents, which also includes concurrently opened versions of doc_a. For these documents the permissions must be recalculated, which possibly revokes permissions of currently edited documents. The concurrent editing of a document can also lead to conflicts, where the editing of one user to doc_a is incompatible to the editing of another user, who also has edited doc_a. For these reasons, we have to perform two steps when a document is checked-in. In step one, we have to solve conflicts between the concurrent versions of a document. In step two, we must update the permissions of other affected documents whose permissions depend on the saved document.

To perform step one, we first retrieve the list of concurrently edited versions of doc_a, which is maintained by the Doc DB for each opened document. Next, we must merge all concurrently edited versions of doc_a to one consistent version. We apply a conflict resolution strategy to solve conflicts between concurrently edited documents. It depends on the scenario to define a specific strategy. One possible strategy is to resolve conflicts manually. An automatic strategy can accept or reject changes depending on the role of the subject that performed the changes or depending on the time the changes were performed, since this information is available in the corresponding histories. After the conflicts are solved, the temporarily stored edges, which correspond to the accepted operations, are saved to the Copy DB.

To perform step two, we first inspect the Copy DB to retrieve the opened documents that might depend on doc_a. These documents have at least one node, that is in is-copy-of relation with a node of doc_a. Then, we recalculate the permissions of these documents for their current users. In some cases, permissions of edited nodes are revoked. In these cases, the UI asks the user whether he wants to reject the current changes or keep them and accept being unable to make further changes. These two steps provide a solution for the third challenge mentioned in the introduction.

3.3 Implementation

We have implemented all components of our system architecture in Java version 1.5. We have extended the XPath function library of the Saxon XSLT and XQuery processor[1] version 8.8 with the functions defined in Section 2.4. The implementation supports all operations defined in Section 2.2. In addition, it is able to evaluate and enforce our AC rules defined in Section 2.3. We have verified the feasibility of our model by evaluating the performance of our implementation. For example, the calculation of a view for a document with 2000 nodes takes less than a second. This performance is sufficiently fast, since our check-in and check-out concept avoids additional view recalculations after every operation on depending documents. Instead, we must update views only when a depending document is checked-in. More details about the implementation and performance evaluation can be found in [12].

4 Related Work

The model proposed in [2] supports selective authorizations to parts of documents based on the semantic structure of XML documents. Authorizations can be defined for different nodes together with propagation options. Regarding these aspects, the model is very similar to our work. However, the supported operations and their semantics are different, since our approach is able to differentiate between objects with different histories. The support of copying data differs from our work, since the model proposed in [2] supports only a push of different views of a document to different sets of users, whereas our model allows us to define which elements of one document may be reused in other documents. Similar approaches can be found in [5,6,8,13], where [8,13] consider access control rules for the read operation only. All these approaches consider the XML element as the smallest unit of protection, in contrast to our approach, which is capable of handling parts of the text.

Iwaihara et al. allow to define access based on the version relationship of documents and elements among each other [11]. They define six operations including *copy*, which is similar to our copy operation, but can only be applied to elements or subtrees and not to text content or parts of the text content. In contrast to our model, the modification of the text content of an element is modeled by the operation *update* only, which describes that the entire content of a node is

[1] See `http://saxon.sourceforge.net/`

replaced with a new content. Concerning AC, Iwaihara et al. only consider read and write operations and do not define a copy operation as part of their privileges. Consequently, they can not express which transfers among documents are permitted or denied. Moreover, they do not have the concept of splitting copied elements to have different history information for parts from different sources.

5 Conclusions and Future Work

In this paper, we have summarized our model for defining access control for XML documents and presented a system architecture that enables us to apply the model in a scenario where multiple users concurrently edit documents in an efficient way. The proposed system architecture maintains the rules, the documents and the history, so that this information is accessible for access decisions of the PDP. We introduced the check-in and check-out approach, which reduces the overhead of recalculating permissions for dependent documents. We specified the workflow for editing a document by explaining the algorithm for the calculation of permissions and the algorithm for the creation of views. We are currently using the implementation of our model to study its usability in different application scenarios.

References

1. Bell, D., LaPadula, L.: Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, MITRE Corp, Bedfort, MA (1973)
2. Bertino, E., Ferrari, E.: Secure and Selective Dissemination of XML Documents. ACM Transactions on Information and System Security 5(3), 290–331 (2002)
3. Brewer, F.D., Nash, J.M.: The Chinese Wall Security Policy. In: IEEE Symposium on Security and Privacy, IEEE Computer Society Press, Los Alamitos (1989)
4. Clark, J., DeRose, S.: XML Path Language (XPath) Version 1.0. W3C recommendation, W3C (1999), `http://www.w3.org/TR/1999/REC-xpath-19991116`
5. Damiani, E., Capitani, S.D., Paraboschi, S., Samarati, P.: Securing XML Documents. In: Zaniolo, C., Grust, T., Scholl, M.H., Lockemann, P.C. (eds.) EDBT 2000. LNCS, vol. 1777, pp. 121–135. Springer, Heidelberg (2000)
6. Damiani, E., di Vimercati, S.D.C., Paraboschi, S., Samarati, P.: A Fine-Grained Access Control System for XML Documents. TISSEC 5(2), 169–202 (2002)
7. Fundulaki, I., Marx, M.: Specifying Access Control Policies for XML Documents with XPath. In: SACMAT 2004. Proceedings of the ninth ACM Symposium on Access Control Models and Technologies, ACM Press, New York (2004)
8. Gabillon, A., Bruno, E.: Regulating Access to XML Documents. In: Working Conference on Database and Application Security, pp. 299–314. Kluwer Academic Publishers, Dordrecht (2002)
9. Gordon, L.A., Loeb, M.P., Lucyshyn, W., Richardson, R.: 2006 CSI/FBI Computer Crime and Security Survey. Technical report, CSI (2006)
10. Graham, G.S., Denning, P.J.: Protection - Principles and Practice. In: Spring Joint Computer Reference, vol. 40, pp. 417–429 (1972)

11. Iwaihara, M., Chatvichienchai, S., Anutariya, C., Wuwongse, V.: Relevancy Based Access Control of Versioned XML Documents. In: SACMAT 2005. Proceedings of the tenth ACM Symposium on Access Control Models and Technologies, Stockholm, Sweden, pp. 85–94. ACM Press, New York (2005)
12. Mellgren, F.: History-Based Access Control for XML Documents. Master's thesis, Technische Universität Darmstadt (June 2007)
13. Murata, M., Tozawa, A., Kudo, M.: XML Access Control using Static Analysis. In: ACM Conference on Computer and Communications Security, ACM Press, New York (2003)
14. Röder, P., Tafreschi, O., Eckert, C.: History-Based Access Control for XML Documents. In: ASIACCS 2007. Proceedings of the ACM Symposium on Information, Computer and Communications Security, ACM Press, New York (2007)
15. Sandhu, R.S., Coyne, E.J., Feinstein, H.L., Youman, C.E.: Role-Based Access Control Models. IEEE Computer 29(2), 38–47 (1996)
16. Tichy, W.F.: RCS - A System for Version Control. Softw. - Practice and Experience 15(7), 637–654 (1985)