# XML Pool Encryption

*Christian Geuer-Pollmann*
Institute for Data Communications Systems, University of Siegen,
Hölderlinstraße 3, 57068 Siegen,Germany
+49-271-740-2516
<geuer-pollmann@nue.et-inf.uni-siegen.de>

*Abstract: This paper describes an alternative encryption method for XML [1] which is capable to encrypt single XML Information Set [2] items. It is able to hide the size and the existence of encrypted contents. As a result, it prevents a 'traffic analysis', i.e. it's analogous counterpart for documents. In 2001, the W3C launched the XML Encryption working group which, among other things, defined how to encrypt portions of XML documents [3]. The portion must always be a subtree or a consecutive sequence of subtrees. On the other hand, XML Access Control allows more granular restrictions on what portions on an XML document a client is allowed to see: XML Access Control can remove an ancestor node from a document while leaving a descendant node in the document. This paper describes an encryption system which allows to have these 'deep children' in plaintext while having the ancestors encrypted, i.e. bringing the property from XML Access Control to XML Encryption.*

**CATEGORIES AND SUBJECT DESCRIPTORS**
E.1 *Data structures* (trees), E.3 *Data encryption*

## 1. INTRODUCTION

Today more and more applications are deployed which use XML [1] as primary data format. As security becomes a requirement for new systems, ways are needed to provide security services on the application layer. Given XML, various security mechanisms are defined which help providing end-to-end-security, like *XML Signature* [4] and *XML Encryption* [3]. Additionally, much research is done in the field of regulating access to XML documents, which e.g. resulted in standards like the *eXtensible Access Control Markup Language* [5].

### 1.1 Motivation

The motivation for this work is quite simple: The server-based XML access control mechanisms can provide some features which cannot be provided using XML Encryption. Given the mechanisms in this paper, these features can be provided using only cryptographic mechanisms, without the need of a server component.

### 1.2 Security Services

In information security, various security services are defined: For this paper, the relevant ones are

1. *confidentiality* (of data),
2. *access control* (for reading XML),
3. *traffic flow confidentiality* or *prevention of traffic analysis*

*Data confidentiality* does keep particular content information secret, i.e. the information is not made available or disclosed to unauthorized individuals or entities. Data confidentiality is usually achieved by using security mechanisms like encryption algorithms or access prevention mechanisms which use physical media protection or routing control.
*Access control* is a service which prevents the unauthorized use of a resource, e.g. reading the contents of a file. In most cases, a single point of control like a trusted enforcement engine enforces a given access control policy.
The term *traffic flow confidentiality* refers to keep the *context* of a given information exchange secret, not only the information transmitted during the communication. Traffic analysis is the inference of information from the observation of traffic flows, i.e. presence, absence, amount, direction and frequency of information exchange [7]. Mechanisms to achieve traffic flow confidentiality are e.g. *data padding* or the *creation of dummy events*. Data padding changes the size of the exchanged information, e.g. by adding data at the beginning and/or the end of a data item. Dummy events prevent that an adversary learns something about the communications frequency on the channel.

## 2. W3C XML ENCRYPTION

This part will work out two properties of W3C XML Encryption [3]:

1. It is not possible to encrypt an ancestor node (an element) while leaving any of the descendants of this node in plaintext.
2. There is information leakage between different users about their own capabilities compared to the other

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM Workshop on XML Security, Nov. 22, 2002, Fairfax VA, USA
Copyright 2002 ACM 1-58113-632-3/02/0011...$5.00.

ones.

## 2.1 Granularity of encryption

The W3C XML Encryption (XEnc) recommendation specifies a confidentiality mechanism for XML. XEnc is capable to encrypt user data like

* complete XML documents,
* single elements (and all their descendants) inside an XML document,
* the contents of an element (some or all child nodes (and all their descendants)) inside an XML document or
* arbitrary binary contents outside of an XML document.

Related to encrypting XML, XEnc allows two different granularity levels: the encryption of full subtrees (a single element and all it’s descendants) or sequences of subtrees (whereas a subtree can be a single node like a text node or also a mixed sequence of comments, elements, text and processing instructions). Including an element in the encryption process always includes the descendants of that element, too. Figure 1-1 illustrate these possiblities.

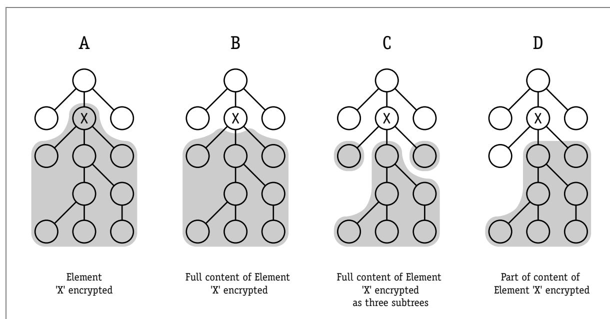


*Figure 1-1: W3C XML Encryption modes*

* Example A shows encrypting an element (and it’s descendants), which refers to the `#Element` type.
* Example B shows encrypting an elements content (`#Content` type).
* Example C encrypted the content of an element using three separate envelopes. Each envelope can have individual encryption properties.
* Example D shows that `#Content` type does not only enforce to encrypt all children of an element, but it also allows to encrypt well-balanced portions. This could even be used to split a single Text node (a sequence of multiple character information items) into encrypted and unencrypted parts[1].

If an encryptor decides to encrypt a given node like an element, W3C XML Encryption constrains that *all* descendants of this node are encrypted, too.

## 2.2 Encryption for multiple recipients

### 2.2.1 Encrypting the same content

Encryption a given resource for multiple recipients can be done in several ways. The trivial case is that all recipients are allowed to see the same portion of the XML document. In that case, the content is only encrypted once, whereas the *content encryption key* is encrypted multiple times, once for each recipient. Such a document contains a single `xenc:EncryptedData` element for the encrypted content and one `xenc:EncryptedKey` element for each recipient. This element contains the content encryption key encrypted under the recipients key.

### 2.2.2 Super-Encryption

Another way to encrypt contents for multiple recipients applies when the contents overlap, i.e. when encrypted contents have to be re-encrypted.

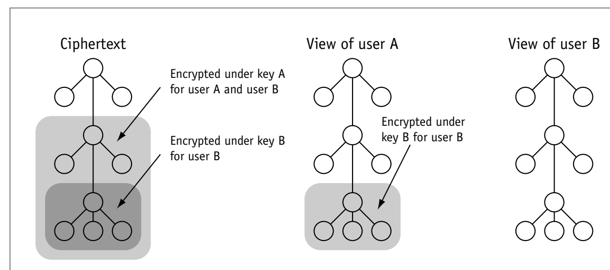


*Figure 1-2: W3C Super-Encryption: Encrypting EncryptedData*

In figure 1-2, a part of the tree is encrypted under a key B for a recipient B. The encrypted contents are, tied up with some plaintext, encrypted under key A for the recipients A *and* B. So when it comes to decryption, the recipient A can decrypt the outer envelope. After that initial decryption, recipient A encounters a second envelope which was encrypted under key B. Since key B is not available to recipient A, the second envelope remains undecrypted. Recipient A is aware of the *existence* of a part in the document which he is not allowed to decrypt; there is information leakage to recipient A that very likely more powerful users of the system exist[2]. Additionally, based on the octet size of the inner envelope, recipient A can make good estimation on how large the inner content is.
The decryption by recipient B is done in two steps: recipient B

---

1. Given DOM, XPath and the Infoset, different data models exist to describe XML; for instance, in a DOM tree, a alternating sequence of Text nodes and CDATA sections are different nodes in the DOM model, whereas the Infoset does not make this distinction but simply refers to the content of these nodes as a long character information item sequence.
2. The term ‘powerful’ refers to the ability to decrypt content. More powerful means more content can be decrypted as there are more keys available to the powerful user.

has access to both key A and key B. After decrypting the outer envelope using key A, the inner envelope B is decrypted. After both encryption steps, the full document is decrypted and available to recipient B. Recipient B is aware that the super-encryption of the inner envelope is (certainly) done in order to prevent other users from accessing the inner information. So there is information leakage to recipient B that (1) he was able to decrypt the full document and that (2) there may exist other users which are not allowed to see the contents of the inner envelope.
So information leaks to *all* users about their own decryption capabilities compared to the abilities of other users. *Both, an adversary and regular users can gain knowledge about the privileges of themselves and other users.*

## 2.3 Former location of the plaintext

When encryption a part of an XML document, the former plaintext is removed from the document and substituted by an `xenc:EncryptedData` element. The ciphertext itself can either be directly included using an `xenc:CipherValue` element or a link to an external location can be done using the `xenc:CipherReference` element. But regardless which mechanisms is choosen, the `xenc:EncryptedData` element tells an adversary where the plaintext had been before. *An adversary can see where plaintext had been.*

## 2.4 Size of plaintext

An adversary with access to the ciphertext can also make a good estimation on the size of the plaintext by inspecting the size of the ciphertext. An encrypted passphrase or credit card number is usually of a smaller magnitute than an encrypted book or catalog is, so a second place of information leakage exists. *An adversary can guess how big the plaintext had been.*

# 3. XML ACCESS CONTROL

In XML Access Control, a trusted *access control processor*, a.k.a. policy enforcement engine, decides based on a policy which portions of a document can be given to a particular user. The processor labels the tree according to the policy (and the users access rights) with '*permit*' and '*deny*' labels. After the labeling step, the document is pruned, i.e. nodes which are finally labeled 'deny' are removed from the document [8].

## 3.1 The invisible ancestors problem

One question must be discussed in more detail: "What happens if an ancestor (namely an element) is labeled 'deny (-)' but a descendant is labeled 'permit (+)' ?". There are different solutions how such a conflict is handled:

### 3.1.1 The DTD/Schema friendly solution

The authors of [8] suggest the following:

> *"Note that, in order to preserve the structure of the document, the portion of the document visible to the requester will also include start and end tags of elements with a negative or undefined label, if the elements have a descendant with a positive label."*

The advantage of this solution is that the damage to with respect to validity is limited: The elememt structure remains the same as in the original document, but all attributes from the confidential element are removed. The disadvantage is that the existence of the ancestor elements remains visible for the requester. It can be envisioned that in particular cases, even the *existence* of the ancestor element should be kept secret.
Given the following XML snippet: The `B` element in example 1-1 is labeled 'deny (-)', while all other elements are permitted to be seen by the requester:

```
<A someAttrInA="foo">
  <B someAttrInB="bar">
    <C someAttrInC="baz">
    </C>
  </B>
</A>
```

*Example 1-1: Input document (XML Access Control)*

During pruning the tree, all attributes of the `B` element are removed, but the element itself remains in the document subset, although access to it is denied. The serialized XML result looks like in example 1-2, the tree structure corresponds to figure 1-3.

```
<A someAttrInA="foo">
  <B>
    <C someAttrInC="baz">
    </C>
  </B>
</A>
```

*Example 1-2: Result after pruning, DTD/Schema friendly solution*

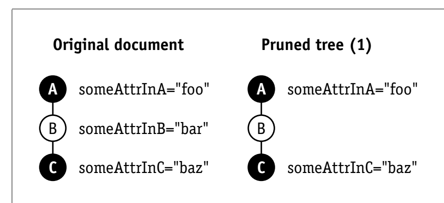


*Figure 1-3: Result after pruning, DTD/Schema friendly solution*

### 3.1.2 Real invisible ancestors

Another solution is to fully omit the ancestor like this is done in Canonical XML. No start or and tags are output if the element is labeled 'deny (-)'. This has a large impact on Schema-validity, if required elements are omitted from the serialized form. On the other hand, this is a clean way to solve the problem, because it is consistent with the idea of serializing an Infoset as defined by Canonical XML.
The result from the sample would look like in example 1-3 and

figure 1-4.

```
<A someAttrInA="foo">

    <C someAttrInC="baz">
    </C>

</A>
```

*Example 1-3: Result after pruning, element completely removed*

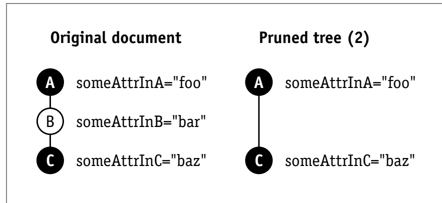


*Figure 1-4: Result after pruning, element completely removed*

Denying access to the B element results in a complete removal of the node, so that the requester does not gain any knowledge about the existence of element in the original document. This model allows to remove ancestors from the document while descendants remain accessible (readable) to the requester.

## 4. Comparison between W3C XML Encryption and XML Access Control

By comparing the results from W3C XML Encryption and XML Access Control, it can be seen that W3C XML Encryption is only capable to encrypt full subtrees, while XML Access control can remove sensitive material from the middle of the tree.
XML Encryption does not allow to encrypt an ancestor while leaving a descendant in plaintext. XML Access Control has this ability, but requires a trustworthy access control processor with access to the full document. This server-based component enforces the policy by removing nodes before giving the results to the client.
XML Encryption does not allow "deep visible children" while XML Access control does. The rest of this work will answer the question on how the goal of deep visible descendants with invisible (encrypted) ancestors can be reached only with cryptographic mechanisms and how the information leakage ("Where are encrypted contents?") can be prevented.

## 5. Pool encryption

### 5.1 Basic idea

The basic idea of pool encryption is to encrypt each node separately and to move all encrypted nodes from their original position in the document into a *pool of encrypted nodes*. This pool can be either inside the document where the plaintext nodes originated from or it can be in one or multiple different documents. When it comes to decryption, the nodes 'magically' find their way back into the appropriate positions.

Each node is encrypted with an unique *node key*. To transfer the node keys to the recipient, the node keys are grouped in a *set of node keys* and this pool is encrypted under the recipents key. Depending on the privileges of a particular recipient (called *decryptor*), different views to the document can be defined by choosing which set of node keys is given to the decryptor.
Figure 1-5 illustrates the basic concept: Given the document tree below, some nodes are to be encrypted, while others remain unencrypted. The selected nodes (namely E, F, J, M, N, O, P, U and V) are to be encrypted in the subsequent steps (marked black):

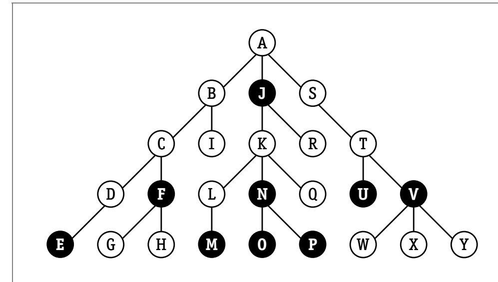


*Figure 1-5: Input document with selected nodes*

In the encryption step, the selected nodes are removed from their original position in the plaintext document. The plaintext of the node (i.e. all necessary information set data) is bundled together with the nodes position information. This tuple is encrypted individually under a randomly choosen node key and the ciphertext of the encrypted nodes is collected in a pool of encrypted nodes. The illustration below does not specifically show the encryption itself, but only the result of the moving operation. During this step, orphaned child nodes which lost their parent node are made childs of their grandparent (or the next unencrypted ancestor if the grandparent is also encrypted). For instance, the node J is encrypted, so K looses its parent; therefore, A becomes the new parent of K.
The pool of encrypted nodes can be either be placed in the original document (dashed line in figure *1-6*) or in a separate XML document.

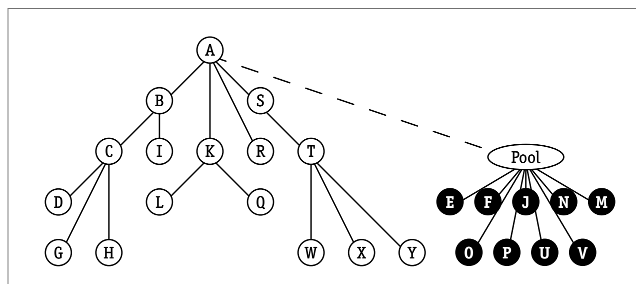


*Figure 1-6: Document with encrypted nodes*

Depending on which decryption keys are available to the decryptor, a specific set of encrypted nodes can be decrypted. In this example, the decryptor is given five node keys to decrypt the nodes N, O, P, U and V. The keys for the nodes E, F, J and M are not available to the decryptor, so these nodes cannot be decrypted. The decrypted nodes are placed back into the docu-

ment, restoring their original position.
After decryption, a view to the document has been established which differs from both the unencrypted original document (which contained *all* nodes) and the encrypted document (in which all confidental nodes have been removed). The original, unencrypted form cannot be reconstructed because the decryptor was *not* given access to *all* necessary decryption keys (figure 1-7).
The overall process of an encryption looks like in figure 1-8.

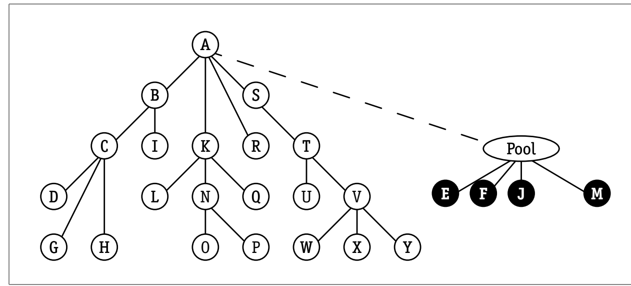


*Figure 1-7: Document after (partial) decryption*

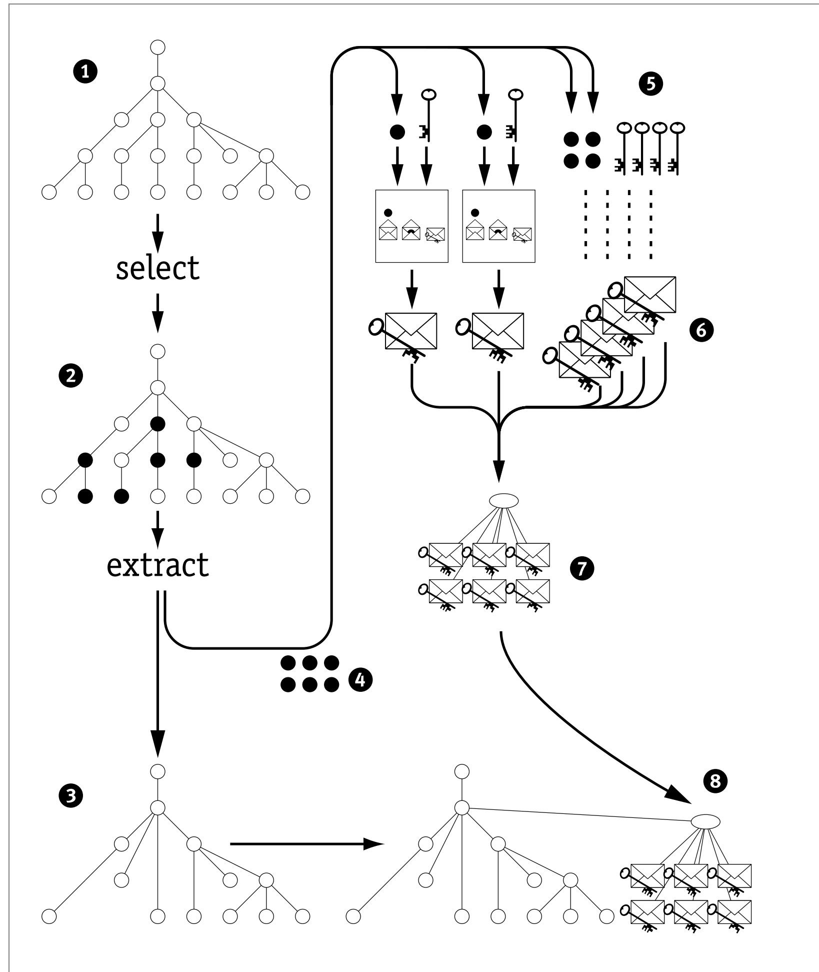


*Figure 1-8: Overall encryption process*

From the input tree (1), the nodes which are to be encrypted are selected (2). The selected nodes are extracted from the tree prior encryption, so that the pruned tree (3) and the extracted nodes (4) are separated. Each extracted node is encrypted with an individual node key (5), resulting in separate encrypted nodes (6). These nodes are bundled in a pool of encrypted nodes (7). The encrypted pool is merged into the pruned tree (7) or can be left as a separate entity in an second document.

## 5.2 Key management

For simplicity, the above example does not show how the node keys are transported to the recipients. Additionally to the pool of encrypted nodes, the node keys are transported to the recipient(s). This is done in a straightforward fashion: Each encrypted node is identified by a *node ID* which is attached to the ciphertext, e.g. a 128 bit identifier. A node key and a node ID are grouped into a tuple. To give a decryptor access to two encrypted nodes, both tuples for the respective nodes are grouped together in a *key pool*; this key pool is encrypted under the recipients key. All *encrypted key pools* (for the various recipients) are grouped in a *pool of encrypted key pools* which can be added to the pool of encrypted nodes[1].

A recipient can decrypt his portion of the document (reconstruct his view) by locating his encrypted key pool, decrypting it (similar to the mechanisms like in W3C xenc, e.g. with his private key), extracting the different (nodeKey/nodeID) tuples, locating the encrypted nodes which correspond to the node IDs and decrypting the nodes and the position using the node keys. Afterwards, the nodes are inserted back into the document using the decrypted node position.

## 5.3 Position of a node in the tree

The biggest problem in the above scenario is to find a good representation for the position of each node. To make decryption possible, the decryptor needs the position information of a decrypted node, i.e. on which place the node was in the original tree. This isn't trivial because the decryptor may have only access to a reduced subset of the original tree.
The absolute position of a node could be described by it's ancestor nodes (i.e. the depth in the tree) and the position relative to it's siblings. If the position information is expressed in terms of ancestors and the decryptor does not see the direct ancestors (e.g. because they cannot be decrypted), the re-insertation of nodes into the tree becomes impossible.
One (insufficient) way to represent the position information using an XPath based expression would be

```
/A[1]/K[1]/N[1]/P[1]
```

to describe the position of the `P` node. The problem with this representation is that the model allows that also ancestor nodes are encrypted, i.e. that some nodes in the XPath are not available in the document. For instance, if the `N` node is encrypted, the above path cannot be evaluated.
A powerful and extensible scheme for describing the position of a node will be described in the following section.

### 5.3.1 The "Adjacency List Mode"

In the article "Trees in SQL" [9], JOE CELKO describes a scheme how tree structures can be stored in flat tables like SQL data bases, i.e. how a tree can be converted into a flat table and restored back from the table information. He called this scheme *Adjacency List Mode* (ALM). Given the tree in figure 1-9.

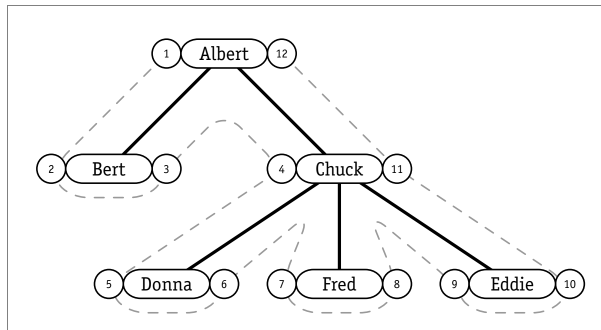


*Figure 1-9: Sample tree for the „Adjacency List Mode"*

1. sorry for this pool-o-mania, it's simply a grouping process ;-)

The tree contains six nodes (`Albert`, `Bert`, `Chuck`, `Donna`, `Fred` and `Eddie`). The algorithm for defining the position of each node has to traverse the full tree. For each node, the position information consists of two integer values, the *left* and the *right value*. Each time a node is visited for the first time, a variable $X$ is incremented by $1$ and the value of $X$ is assigned to the nodes left value. Each time a node is visited the second time, the variable $X$ is incremented by $1$ and the value of $X$ is assigned to the nodes right value.
Starting with an initial value of $X = 0$, the node `Albert` is the first on in the traversal. `Albert` is visited for the first time, $X$ is incremented by $1$ and the new value is assigned to the nodes left value: $Albert_{left} = 1$. `Bert` is visited for the first time, $X$ is incremented by $1$ and the new value is assigned to the nodes left value: $Bert_{left} = 2$. `Bert` has no further descendants, so the algorithm visits Bert for the second time, $X$ is incremented by $1$ and the new value is assigned to the nodes right value: $Bert_{right} = 3$. This method is applied to all nodes in the tree. The dashed grey line outlines the order in which the labels are assigned to the left and right values of the respective nodes, i.e. the way of the traversal.
The tabular representation of the tree in figure 1-9 is listed in table 1-1.

| Node | left | right |
|---|---|---|
| Albert | 1 | 12 |
| Bert | 2 | 3 |
| Chuck | 4 | 11 |
| Donna | 5 | 6 |
| Fred | 7 | 8 |
| Eddie | 9 | 10 |

*Table 1-1: Sample table for the „Adjacency List Mode"*

Given table 1-1, it is simple to determine the position of two given nodes relative to each other. The range which is spanned by the left and the right value determine the position in the tree. Figure 1-10 shows the ranges for all nodes.

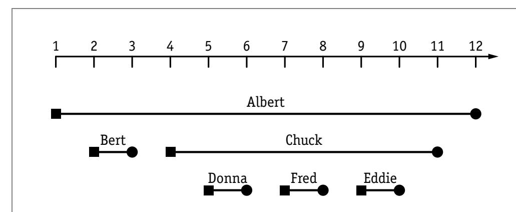


*Figure 1-10: Sample ranges for the „Adjacency List Mode"*

It can be seen that the range of `Albert` includes the ranges of all other nodes, so `Albert` is in the `[ancestor axis]` of *all* other nodes, i.e. `Albert` is the root node. The ranges of `Bert` and `Donna` have no common interval, so there is no ancestor/descendant relationship between them. `Bert` is in the `[preceeding axis]` of `Donna` and `Donna` is in the `[following axis]` of `Bert`. So the left and right values

can be used to determine whether a given node is on the `[self axis]`, `[ancestor axis]`, `[descendant axis]`, `[preceeding axis]` or `[following axis]` of a second node.

Given a context node C and node X, it has to be defined on which of these five axes the node X is located (relative to C). After labeling the tree according to the above algorithm, this classification can be done this way:
Four parameters from table 1-2 are available. The classification

| Symbol | Type |
|---|---|
| $C_L$ | Left value of the context node |
| $C_R$ | Right value of the context node |
| $X_L$ | Left value of the node which is to be classified |
| $X_R$ | Right value of the node which is to be classified |

*Table 1-2: Used symbols*

is done as in table 1-3. The traversal algorithm guarantees that the left value of a node is always smaller than its right value. The Adjacency List Mode (ALM) always converts a *full and complete* tree into a table structure and vice versa. This property allows an easy labeling scheme: To assign the left and right values to the nodes, the counter is incremented by 1 in each step. The ALM is not able to label a partial tree.

| Axis | condition 1 | condition 2 |
|---|---|---|
| $X$ is on the `[self axis]` of $C$ | $C_L = X_L$ | $C_R = X_R$ |
| $X$ is on the `[ancestor axis]` of $C$ | $X_L < C_L$ | $C_R < X_R$ |
| $X$ is on the `[descendant axis]` of $C$ | $C_L < X_L$ | $X_R < C_R$ |
| $X$ is on the `[preceding axis]` of $C$ | $X_R < C_L$ | |
| $X$ is on the `[following axis]` of $C$ | $C_R < X_L$ | |

*Table 1-3: Classification by document order*

When applying this algorithm to a tree, a full labeling traversal is performed. There is no need to attach the left/right values to the nodes in the tree, because the full tree is converted to the table on-the-fly. The tree is not altered by attaching the labels directly to the tree.
When it comes to encrypting (parts of) an XML tree, this labeling scheme has an important drawback: Removing encrypted nodes from the tree messes the labeling algorithm and decryption becomes impossible. Using the scheme to increase the counter by 1 and labeling the tree, the example looks like in figure 1-11.

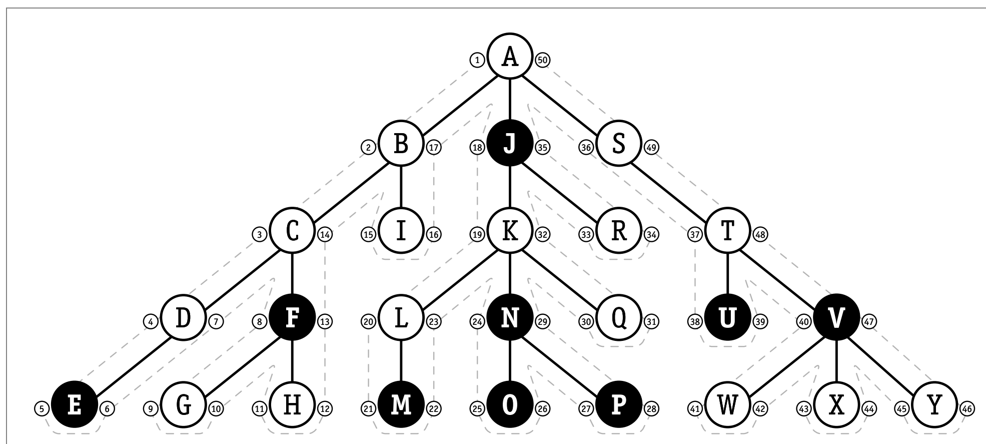


*Figure 1-11: Labeled plaintext document*

After pruning the tree, all nodes which have been selected for encryption are removed from tree and the remaining numbering scheme looks like figure 1-12.
It can be seen that the created number sequence (1, 2, 3, 4, 7, 9, 10, 11, 12, 14, ..., 50) cannot be reconstructed by the decryptor. In the pruned document, the decryptor needs access to the labels of the unencrypted nodes in order to perform the decryption correctly. The numbering cannot be simply reconstructed by re-labeling the document prior decryption, because the numbering scheme is not an 'increment-by-one' sequence. In the above document, the labels would have to be transferred *into* the document. This can be accomplished by adding specific at-

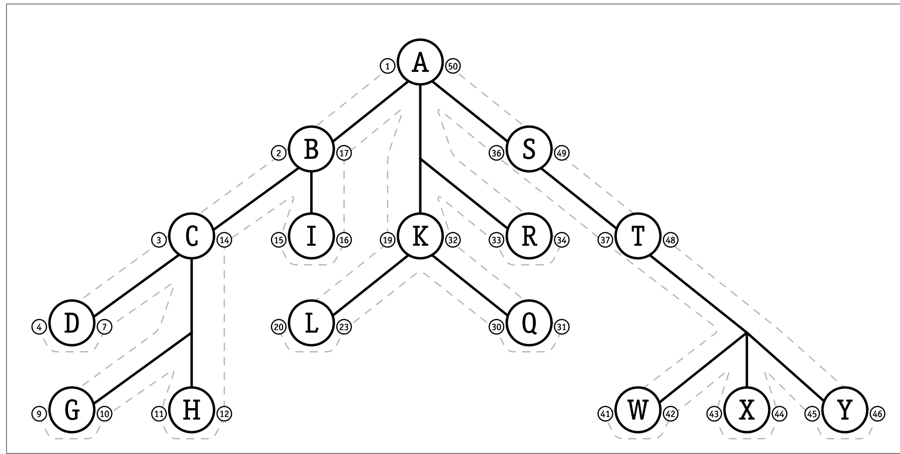


*Figure 1-12: Labeled and pruned plaintext document*

tributes for the left and right value to the document[1]:

```
<A  xmlns:pe="http://xmlsecurity.org/#poolenc"
          pe:L="1" pe:R="50">
  <B      pe:L="2" pe:R="17">
    <C    pe:L="3" pe:R="14">
      <D pe:L="4" pe:R="7" />
      <G pe:L="9" pe:R="10" />
      <H pe:L="11" pe:R="12" />
    </C>
    <I    pe:L="15" pe:R="16" />
  </B>
  <K      pe:L="19" pe:R="32">
    <L    pe:L="20" pe:R="23" />
    <Q    pe:L="30" pe:R="31" />
  </K>
  <R      pe:L="33" pe:R="34" />
  <S      pe:L="36" pe:R="49">
    <T    pe:L="37" pe:R="48">
      <W pe:L="41" pe:R="42" />
      <X pe:L="43" pe:R="44" />
      <Y pe:L="45" pe:R="46" />
    </T>
  </S>
</A>
```

*Example 1-4: Serialized plaintext with left/right values in attributes*

In the pruned document, the left and the right values are included by using attributes in a particular namespace to store these values so that the decryptor has access to them.
The most obvious drawback of this approach is the ‘pollution’ of the documents infoset with a large number of attribute and namespace nodes; the size of the pruned document is increased significantly.
The second disadvantage is more subtle: The simple ‘increase-by-one’ scheme for incrementing the X counter enables an attacker to make good assumtions in which places a node has been removed (because it was encrypted) and where no encrypted nodes have been as there is no space to do re-insert one. For instance, the Y element cannot have child nodes after decryption, because the difference between the right and the left value is 1.

### 5.3.2 The Modified Adjacency List Mode

A more advantagous way to label the tree is to increment the counter in larger (defined) steps. This is done by modifying the label algorithm sightly:
The tree labeling algorithm is done similar to the original ALM. Two traversals are performed. The first traversal assignes labels to the unselected[2] nodes, the second traversal labels the selected nodes.
The algorithm uses a parameter, the *step size* $S$. $S$ defines in which steps X is incremented. In the original ALM, $S = 1$.
In the first traversal run, only unencrypted nodes are labeled by incrementing the counter $X$ using the step size $S$. The selected nodes are skipped during this first traversal. After the first traversal, all unselected nodes (which remain unencrypted) have been assigned a left/right pair.
In a second step, the remaining nodes are labeled. Due to the property that $S \gg 1$, there are ‘gaps’ between the left/right values of two unencrypted nodes. The left/right values $V$ of unselected nodes always are always aligned on the values $V = n \cdot S, n \in \aleph$. This gap is used to assign interstitial[3] values to the selected nodes.
Note: The above description uses two independent traversals; it is possible to easily label the tree in a single traversal step.
Figure 1-13 shows a simplified tree which is completely labeled. The step size is chosen $S = 1000$, the white nodes are plaintext nodes, the black nodes will be encrypted subsequent-

1. The example tree in figure 1-12 consists only of elements. The XML source code adds whitespace for indentation to show the depth of the tree. These whitespace text nodes do not show up in the figure, so strictly speaking, both figures do not match.

2. a ‘selected’ node will be encrypted later, an ‘unselected’ node remains in the document.
3. *interstitial*: (german: „Zwischengitterplatz“) In crystallography, an interstitial is a place in the crystals lattice where foreign atoms can be inserted, e.g. by doping a semiconductor.

ly:

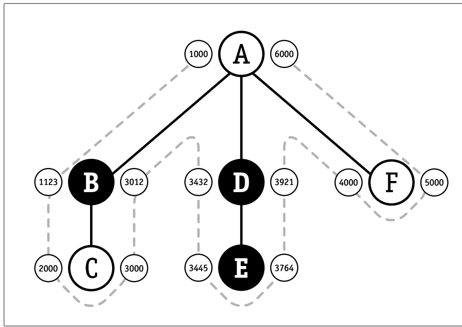


*Figure 1-13: Labeled tree (Modified ALM)*

The plaintext nodes A, C and F are assigned the values 1000, 2000, 3000, 4000, 5000 and 6000. The selected nodes are assigned values which do reside between these values (1123, 3012, 3432, 3445, 3764 and 3921), so after pruning the tree (figure 1-14), it cannot be derived whether nodes have been between the unencrypted nodes. It will be discussed later how this segmantation os done.

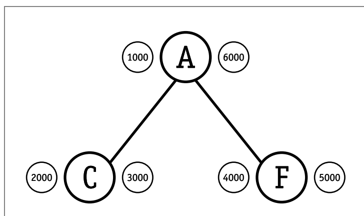


*Figure 1-14: Pruned tree (Modified ALM)*

The value of the step size $S$ defines how many nodes have place between two unencrypted nodes. The lower bound for the step size is $S > 2$ , in order to allow at least a single encrypted node to fit into the interval. Generally speaking, to allow $n$ encrypted nodes to be descendants of an unencrypted node, the size must be choosen $S > 2n$ .

## 5.4 Dummy Nodes

Without further countermeasures, an adversary can determine how many nodes have been removed from the original document by counting the encrypted nodes in a pool. Given the security service *prevention of traffic flow analysis*, a similar service can be defined for encrypted trees: An attacker should not be able to gain knowledge about how many nodes have been in the original document. It should be hidden from attackers and also from all legitimate users how the original structure has been. Given all node keys, a legitimate user can decrypt the full document, but he'll never know that he reached this state.

Based on the available node keys, three different classes of attackers are defined:

1. Attackers without access to *any* decryption key.
2. Attackers with access to a *reduced set* of decryption keys.
3. Attackers with access to the *all* decryption keys.

An attacker *without any decryption key* has only access to the pruned tree. Depending on how the encrypted nodes are organized, the attacker has access to a particular set of encrypted nodes. From this set, the attacker can count how many nodes are in the set. The step size parameter from the labeling scheme allows to calculate how many nodes can exist in the original document.

An attacker with access to a *reduced set* of decryption keys can decrypt some nodes from the pool and therefore reconstruct parts of the document. After the decryption, the attacker can count how many nodes remain undecrypted in the pool (or more exactly, in the pools he is aware of). In contrast to the previous attacker, he knows some left/right values of decrypted nodes, so he can make a better assumtion on how many nodes have place in particular areas of the tree.

A decryptor with full access to *all* decryption keys can fully reconstruct the original tree. But is such a decryptor an attacker? It seems that this decryptor already has access to anything, but that's not the case: It can be hidden from this decryptor *that* he has already full access.

The left/right values assigned to encrypted nodes are randomly choosen (within the required interval). Therefore, a decryptor cannot determine whether he already decrypted all nodes. To support this uncertainty, the encryptor can add *dummy nodes* to the pool of encrypted nodes. A dummy node is the analogy to the *data padding* and *dummy events* from the traffic flow confidentiality. No decryptor is given the key required to decrypt a dummy node, therefore all decryptors must assume that the encrypted envelope containing the dummy node contains a node which they are not allowed to see.

The doubt whether a decryptor can see the full document or not is also increased by the possibility of pools which he is not aware of.

## 5.5 Collaboration of decryptors

Each decryptor who is allowed to decrypt a particular node is given the node key for that node. All node keys for a decryptor are grouped in the key collection owner by this decryptor. A key collection is mathematically a *set*. Two decryptors which are allowed to decrypt a given node do have the same node key in their respective key collections. Multiple decryptors with (partly) disjunctive key collections can collaborate to decrypt a larger part of the tree: By merging different key collections, a larger set is composed which decrypts a larger part of the tree: Given that user $A$ has the keys $k_1$ and $k_2$ so that the encrypted nodes $N_1$ and $N_2$ can be decrypted. User $B$ has the keys $k_1$ and $k_3$ so that the encrypted nodes $N_1$ and $N_3$ can be decrypted. By merging their key sets, the nodes $N_1$ , $N_2$ and $N_3$ can be decrypted.

# 6. CONCLUSIONS

The presented system is able to encrypt XML at the level of the information set. In contrast to W3C XML Encryption, even ancestors can be encrypted while descendants remain unencrypted. Using cryptographic mechanisms, it provides the same facilities like XML Access Control (w.r.t. *read operations* on a document).

The design of the system pays special attention to information

leakage. It is possible to encrypt (parts of) a document in a way so that no traces of the encryption process remain in the plaintext. It can hide the origins of ciphertext, the size of ciphertext and even the existence of ciphertext. Enforcing the use of dummy nodes can even promote esoteric security features like *plausible deniability* (which has not been described in this paper). Of course, unmindfully use of the system can destroy DTD or Schema validity, but this is the case for all systems which modify documents like XML Encryption or Access Control.

# A. References


**1** W3C, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, *Extensible Markup Language (XML) 1.0 (Second Edition)*, W3C Recommendation 6 October 2000 `http://www.w3.org/TR/2000/REC-xml-20001006`

**2** W3C, John Cowan, Richard Tobin, *XML Information Set*, W3C Recommendation 24 October 2001 `http://www.w3.org/TR/2001/REC-xml-infoset-20011024`

**3** W3C, Takeshi Imamura, Blair Dillaway, Ed Simon, Donald Eastlake, Joseph Reagle, *XML Encryption Syntax and Processing*, W3C Candidate Recommendation 02 August 2002 `http://www.w3.org/TR/2002/CR-xmlenc-core-20020802`

**4** W3C, Donald Eastlake, Joseph Reagle, David Solo, Mark Bartel, John Boyer, Barb Fox, Brian LaMacchia, Ed Simon, *XML-Signature Syntax and Processing*, W3C Recommendation 12 February 2002 `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212`

**5** Simon Godik, Tim Moses, *XACML - The OASIS eXtensible Access Control Markup Language (XACML)*, Working Draft 14, 14 June 2002, `http://www.oasis-open.org/committees/xacml/`

**6** Paul Grosso and Daniel Veillard, *XML Fragment Interchange*, W3C Candidate Recommendation, 12 February 2001 `http://www.w3.org/TR/2001/CR-xml-fragment-20010212`

**7** International Organization for Standardization, Walter Fumy, *ISO/IEC JTC 1/SC 27 --- Information Technology -- Security techniques: Glossary of IT Security Terminology*, March 1998, `http://www.din.de/ni/sc27/doc6.html`

**8** E. Damiani, S. De Capitani di Vimercate, S. Paraboschi and P. Samarati, *Design and Implementation of an Access Control Processor for XML Documents*, Proceedings of Ninth International World Wide Web Conference, Amsterdam, May 2000 `http://www9.org/w9cdrom/419/419.html`

**9** Joe Celko, Trees in SQL, Oktober 2000 `http://www.intelligententerprise.com/001020/celko1_1.shtml`