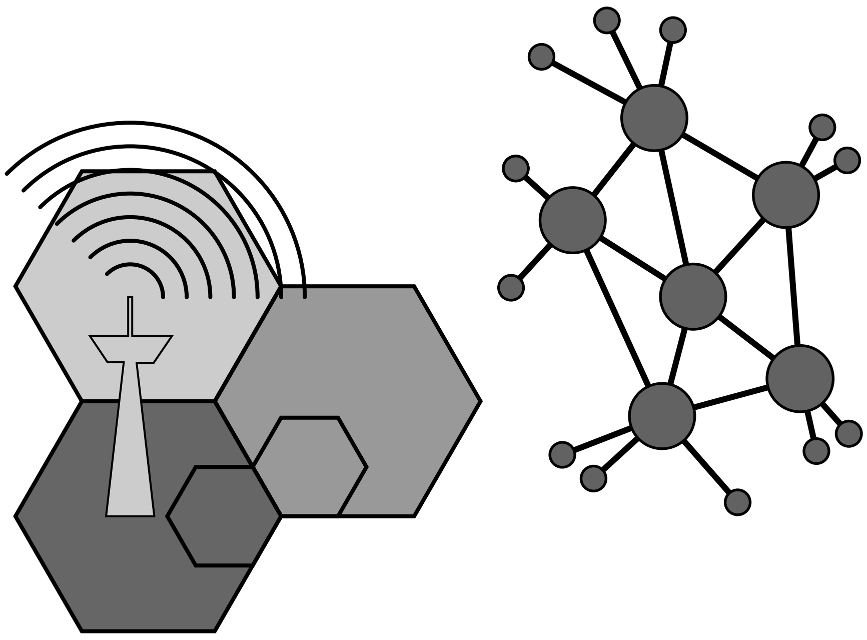
UNIVERSITÄT SIEGEN

Institut für Digitale Kommunikationssysteme

Forschungsberichte

Band 9 Herausgeber: Univ.-Prof. Dr. Christoph Ruland



Christian Geuer-Pollmann

Confidentiality of XML documents by Pool Encryption

2004

Confidentiality of XML documents by Pool Encryption

Vom Fachbereich Elektrotechnik und Informatik der Universität Siegen

zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften (Dr.-Ing.)

genehmigte Dissertation

von

Dipl.-Ing. Christian Geuer-Pollmann

1. Gutachter: *Univ.-Prof. Dr. Christoph Ruland*, Universität Siegen
2. Gutachter: *Univ.-Prof. Dr. Rüdiger Grimm*, TU Ilmenau

Vorsitzender: *Univ.-Prof. Dr. Wolfgang Merzenich*, Universität Siegen

Tag der mündlichen Prüfung: 29. Juli 2003

urn:nbn:de:hbz:467-647
urn:isbn:3-8322-2846-2

UNIVERSITÄT SIEGEN

Institut für
Digitale Kommunikationssysteme

Forschungsberichte

Herausgeber: Univ.-Prof. Dr. Christoph Ruland

Band 9

Christian Geuer-Pollmann

Confidentiality of XML documents by Pool Encryption

2004

Bibliografische Information der Deutschen Bibliothek
Die Deutsche Bibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.ddb.de/ abrufbar.

Zugl.: Siegen, Univ., Diss., 2003

ISBN 3-8322-2846-2
ISSN 1614-0508

Shaker Verlag GmbH • Postfach 101818 • 52018 Aachen
Telefon: 02407 / 9596 - 0 • Telefax: 02407 / 9596 - 9
Internet: www.shaker.de • eMail: info@shaker.de

Für Gabriele,
Georgia und Philipp

Acknowledgements

This work was written while I was research assistant at the ‘Institute for Data Communication Systems’ at University of Siegen between November 1998 and May 2003.

This is the place to dedicate a sincere “thank you” to the people I have been surrounded with.

First, I want to express my gratitude to Prof. Dr. Christoph Ruland for being my scientific mentor, for his valuable suggestions and amicable discussions as well as the excellent working conditions at the Institute for Data Communication Systems in Siegen.

My thanks further go to Prof. Dr. Rüdiger Grimm from TU Ilmenau for his interest in my work and for being my second promoter and to Prof. Dr. Wolfgang Merzenich for chairing the board of examiners.

It was a great pleasure to work together with Christoph Stepping, Oliver Jung, Luigi Lo Iacono, Tobias Lohmann, Seung Wook Jung, Sven Kuhn and especially Kai Wollenweber and Niko Schweitzer. Thanks also to Matthias Schneider the Institute’s good soul, Christine Haßler, for continuous support during the adversities of everyday life at the University.

Besides these direct contacts at the Institute, I would also like to thank the members of the W3C XML Signature Working Group (most notably Ed Simon, John Boyer, Gregor Karlinger and Merlin Hughes) for deep insights into the XML internals. To my friend Christian Lemburg a “thank you” for the invaluable reference to Joe Celko’s Adjacency List Mode.

Special thanks go to my parents Maria and Heinrich Geuer, my siblings Franz, Klaus and especially Hildegard, my parents-in-law Marlies and Eberhard Pollmann as well as my brother-in-law Andreas for their motivation, their moral backing and perpetual support and uncounted hours of baby sitting.

I would like to thank my beloved children Georgia and Philipp for the love which they gave me, inadvertently of how many hours I disappeared at my computer.

My deepest thanks go to my wife Gabriele for her love and patience, her encouragement and her faith in me during the years. This work would not have been written without this nearest counselor and friend of mine.

Düsseldorf, July 2003

Christian Geuer-Pollmann

Danksagung

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter des Institutes für Digitale Kommunikationssysteme an der Universität Siegen im Zeitraum zwischen November 1998 und Mai 2003.

Nun ist der Zeitpunkt gekommen, den Menschen ein herzliches Dankeschön zu widmen, die mich während dieser Zeit mit Rat und Tat unterstützt haben.

Zunächst danke ich Prof. Dr. Christoph Ruland für die wissenschaftliche Betreuung meiner Arbeit, für die wertvollen Anregungen und freundschaftlichen Diskussionen sowie für die exzellenten Arbeitsbedingungen am Institut für Digitale Kommunikationssysteme in Siegen.

Mein Dank gebührt weiterhin Prof. Dr. Rüdiger Grimm von der TU Ilmenau für das der Arbeit entgegengebrachte Interesse und die Übernahme des Korreferats sowie Prof. Dr. Wolfgang Merzenich für die Bereitschaft, den Vorsitz der Prüfungskommission zu übernehmen.

Meinen ehemaligen Kollegen Christoph Stepping, Oliver Jung, Luigi LoIacono, Tobias Lohmann, Seung Wook Jung, Sven Kuhn und insbesondere Kai Wollenweber und Niko Schweitzer danke ich für die gute Zusammenarbeit und die schöne, gemeinsame Zeit am Institut. Matthias Schneider und der guten Seele des Instituts, Christine Haßler, gebührt mein besonderer Dank für die stetige Unterstützung bei den Widrigkeiten des Universitäts-Alltags und die freundschaftliche Zusammenarbeit.

Neben den direkten Kollegen am Institut möchte ich den Mitgliedern der W3C XML Signature Working Group (insbesondere Ed Simon, John Boyer, Gregor Karlinger und Merlin Hughes) für die tiefen Einblicke in die Interna von XML mit allen seinen Facetten danken. Meinem Freund Christian Lemburg danke ich für den unbezahlbaren Hinweis auf Joe Celkos Beschreibung des Adjacency List Mode.

Meinen Eltern Maria und Heinrich Geuer, meinen Geschwistern Franz, Klaus und insbesondere Hildegard, meinen Schwiegereltern Marlies und Eberhard Pollmann sowie meinem Schwager Andreas möchte ich meinen ganz besonderen Dank sagen für ihre Motivation, die moralische Unterstützung, den immerwährenden, verlässlichen Rückhalt sowie für die Betreuung der Kinder.

Meinen geliebten Kindern Georgia und Philipp danke ich für die Liebe, mit der sie mir begegnet sind, wenn ich wieder einmal am Rechner verschwunden war.

Meiner Ehefrau Gabriele möchte ich größten Dank sagen für ihre Liebe und Geduld, ihren Zuspruch und ihr Vertrauen während der vergangenen Jahre. Ohne diese engste Beraterin, Freundin und größte Stütze wäre mir diese Arbeit nicht möglich gewesen.

Düsseldorf, im Juli 2003

Christian Geuer-Pollmann

Abstract

The eXtensible Markup Language (XML) is a widely adopted format for documents containing structured information. Structured information contains both the content (words, images etc.) and the 'markup' which indicates the role of the content, e.g. 'section' or 'price'.
XML is the foundation for a huge variety of existing and emerging applications, including user applications like vector imaging formats, web pages, enterprise application integration, database interfaces or network protocols.
Parallel to the increasing use of XML, the level of security provisions for these XML based systems rises. The World Wide Web Consortium (W3C) addressed these issues by creating the "XML Signature Syntax and Processing" and "XML Encryption Syntax and Processing" recommendations. These standards define authentication, integrity and confidentiality mechanisms for XML documents.
The XML Signature recommendation defines a method for digitally signing arbitrary portions (nodes) of an XML document. XML Signature can sign both tree structures and arbitrary sets of nodes of an XML document.
The XML Encryption recommendation specifies a method for encrypting tree structures in an XML document. The XML Encryption recommendation is constrained to protect full tree structures, i.e. there is no mechanism to protect the confidentiality of a single node in a document without affecting the descendants of that node.
The access control community transformed access control models originating in database systems to be available for XML based databases. These access control systems offer fine-grained access control enforcement on the node level, similar to the node level integrity protection of XML Signature. For example, XML Access Control systems can restrict the read access to a particular node in an XML tree while allowing access to its child nodes.
This thesis is focused on the development of a cryptography based system which can protect the confidentiality of arbitrary nodes in an XML tree. This goal is reached by combining a tree addressing scheme of databases with cryptographic mechanisms. This system is called "XML Pool Encryption".
To verify the results of this thesis, XML Pool Encryption has been implemented using the Java programming language.

Zusammenfassung

Die eXtensible Markup Language (XML) ist ein weit verbreitetes Format für Dokumente, die strukturierte Information enthalten. Strukturierte Information umfasst sowohl den eigentlichen Inhalt (z.B. Wörter, Bilder, etc.) sowie Auszeichnungsinformation, um die Rolle der Inhalte zu umschreiben, z.B. "Überschrift" oder "Preis".
XML bildet die Grundlage für eine große Anzahl existierender und im Entstehen begriffener Anwendungen, wie z.B. Vektorgrafik-Formate, Web Seiten, Enterprise Application Integration Systeme, Datenbank Schnittstellen oder Netzwerkprotokolle.
Parallel zur steigenden Verbreitung von XML werden immer mehr Vorkehrungen zum Schutz der auf XML basierenden Systeme notwendig. Das World Wide Web Consortium (W3C) hat sich dieser Notwendigkeit angenommen, indem die "XML Signature Syntax and Processing" und die "XML Encryption Syntax and Processing" Empfehlungen verabschiedet wurden. Diese Standards definieren Mechanismen für Authentisierung, Integrität und Vertraulichkeit von XML Dokumenten.
Die XML Signature Recommendation definiert einen Mechanismus, um beliebige Teile eines XML Dokumentes (Nodes) digital zu signieren. XML Signature kann sowohl Baumstrukturen als auch beliebig geformte Knotenmengen eines XML Baumes schützen.
Die XML Encryption Recommendation definiert einen Mechanismus für das Verschlüsseln von Baumstrukturen innerhalb eines XML Dokumentes. W3C XML Encryption ist hierbei auf die Verschlüsselung kompletter Baumstrukturen beschränkt, d.h. es existiert keine Möglichkeit, die Vertraulichkeit für einzelne Knoten im Dokument zu gewährleisten, ohne dass die Kinder dieser Knoten ebenfalls geschützt werden.
Für die Zugriffskontrolle von XML basierten Daten wurden Zugriffsschutzmodelle aus dem Datenbankbereich überarbeitet. Diese Systeme bieten die Durchsetzung fein granularer Zugriffskontrolle auf Knotenebene, ähnlich dem Integritätsschutz beliebiger Knoten bei XML Signature. So ist es beispielsweise möglich, den Lesezugriff auf einen Knoten zu verweigern, während die Kinder dieses Knotens weiterhin lesbar bleiben.
Im Mittelpunkt dieser Arbeit steht die Entwicklung eines auf kryptografischen Verfahren basierenden Systems, welches die Vertraulichkeit für beliebige Knoten eines XML Baumes gewährleistet. Dieses Ziel wurde durch die Kombination eines Schemas für die Adressierung von Baumstrukturen mit kryptografischen Verfahren erreicht. Dieses System wird "XML Pool Encryption" genannt.
Zur Überprüfung der Resultate dieser Arbeit wurde XML Pool Encryption in Java implementiert.

Table of Contents

List of figures

1 Introduction

During the last years, the eXtensible Markup Language (XML) has been widely adopted as ‘lingua franca’ for the Internet. The World Wide Web Consortium (W3C) created a whole set of specifications surrounding the basic XML v1.0 language specification.
These new specifications include the reformulation of HTML using XML (XHTML), vector graphics formats (Scalable Vector Graphics SVG), metadata formats for policy documents, e.g. the Platform for Privacy Preferences Project (P3P), multimedia languages (SMIL) and control languages for voice recognition systems (VoiceXML). The Simple Object Access Protocol (SOAP) or XML Protocol (XMLP) enables the transport of documents through the Internet and remote procedure calls (SOAP-RPC).
Besides the application level languages, the W3C created a framework of base specifications, e.g. linking languages (XLink, XPointer and XPath), transformation languages (XSLT) and security specifications.
These security specifications include the “W3C XML Signature Syntax and Processing” Recommendation for digital signatures, the “W3C XML Encryption Syntax and Processing” Recommendation for the encryption of XML documents and the “XML Key Management Specification” to define interfaces to Public Key Infrastructures via XML based protocols. Both W3C XML Signature and W3C XML Encryption are finished standardization activities, i.e. the working groups have delivered a Technical Recommendation.
An XML document is a tree structure, i.e. it consists of XML nodes which form the tree. Each element node in the document is the root of a subtree. The leaf element nodes are subtrees which consist of a single element node. An arbitrary node set is an arbitrary compounded set of nodes without constraints to the composition, i.e. the presence of a node in the node set does not imply that the node’s child nodes are also present in the subset.
The XML Signature Recommendation enables users to sign parts of an XML document, including the whole document tree, subtrees in the document, or arbitrary node sets.
The XML Encryption Recommendation enables users to encrypt parts of an XML document, including the whole document tree or subtrees in the document. The design of W3C XML Encryption does not allow the encryption of arbitrary node sets. The encryption of a confidential node implies that the child nodes of this confidential node are encrypted, too.
XML Access Control Processors are systems which enforce access control policies for XML documents and parts thereof. These systems allow the definition of fine-grained access control policies, i.e. to define an access control policy for each node in an XML document’s tree. For instance, XML Access Control policies can define that the access to a particular element node is denied while access to the child nodes of this element node is permitted. Therefore, XML Access Control enables the access control for arbitrary node sets in an XML document.

Table 1-1 shows which security mechanism can be used to provide a security service for a given XML structure:

Security mechanism	**full XML document**	**subtrees of an XML document**	**Arbitrary XML node sets**
integrity and authentication	W3C XML Signature	W3C XML Signature	W3C XML Signature
confidentiality	W3C XML Encryption	W3C XML Encryption	
access control	XML Access Control	XML Access Control	XML Access Control

Table 1-1: Security mechanisms by secured data

The above table indicates that there exists no confidentiality mechanism for arbitrary node sets.
The aim of this thesis is to develop a confidentiality mechanism for arbitrary node sets, which does not require an online access control processor to enforce given confidentiality requirements. The confidentiality mechanism developed in this thesis is called “XML Pool Encryption”.
XML Pool Encryption is a cryptographic system which is able to encrypt arbitrary confidential nodes in an XML document. The problem is that the encryption of a single node has dramatic impact on its child nodes, as the context of the child node changes. XML Pool Encryption takes confidential nodes out of the document, encrypts them and collects the encrypted nodes in a so called pool. The users of the system can decrypt the encrypted nodes and restore them so that the original document structure is rebuilt.
The main task during the development of XML Pool Encryption was to find a sufficient description of a confidential node’s position, so that the document can be restored correctly.
The idea to create XML Pool Encryption appeared during the author’s work on standardization and implementation of the W3C XML Signature and W3C XML Encryption specifications.
The structure of this thesis is as follows:
Chapters 2 to 5 contain an introduction to the background of the problem.
Chapter 2 introduces general concepts of security services and security mechanisms, including data confidentiality, traffic flow confidentiality, integrity, authentication, access control and plausible deniability.
Chapter 3 gives an introduces to the relevant XML standards, including XML v1.0, XML Namespaces, the XML Information Set, the Document Object Model DOM and the XML Path language XPath.
Chapter 4 describes Canonical XML and the W3C XML Signature Recommendation as an example how security mechanisms and XML are combined to provide integrity and authentication.
Chapter 5 introduces current confidentiality mechanisms, including W3C Encryption and XML Access Control.
Chapter 6 is the motivation for the work of this thesis, summarizing the relevant properties of W3C XML Encryption and XML Access Control and defining requirements for XML Pool Encryption.

Chapter 7 develops the XML Pool Encryption system. A sketch of the basic idea, the introduction of relevant terms and the concepts and design principles can be found at the beginning of chapter 7. After this introduction, chapter 7 introduces the Adjacency List Node to represent trees and defines the modification of the Adjacency List Mode to enable XML Pool Encryption. The exact procedures to perform encryption and decryption are described. After proving the correctness of the encryption and decryption process, key management issues and encryption details are defined. The introduction of a mechanism for traffic flow confidentiality and an approach to the editing of encrypted documents concludes the chapter.
Chapter 8 describes properties of XML Pool Encryption as a comparison between the results of chapter 7 with the requirements defined in chapter 6.
Chapter 9 summarizes this work.
Appendix A contains a description of the XML Pool Encryption and XML Signature implementations.

2 IT Security Services and Mechanisms

In the definition of computer systems and networks, usually the term "*entity*" is used to refer to persons as well as computer processes running on behalf of a person or even computer processes which run without human interaction. While humans process *information*, computer systems process data which is a particular *representation* of *information*. For instance, a number can be represented in the binary form using a bit string of a defined length. Other information is represented by sequences which are octet strings in a computer system, which are encoded in a defined character set.

bit: One of the two symbols '0' or '1'.

bit string: An ordered sequence of bits.

octet: A bit string of length 8.

octet string: An ordered sequence of octets.

block: String of bits of a defined length.

The use of IT systems introduces risks and threats which have to be addressed. These risks can be managed by reducing the possible impact of a given threat or by reducing the probability of occurrence. Reducing the probability is usually done by using security services. "Security services ensure adequate security of the systems or of data transfers" [ITU-T X.800 | ISO 7498-2] and define the properties and functionality of security mechanisms, which address the threat.
In the following sections, the relevant security definitions and concepts are summarized.

Transmission versus storage. This thesis is about confidentiality for documents. A document may be transmitted over a communication channel (e.g. the Internet) or it may be stored on a storage medium (e.g. a hard drive or a database system). Transmitting a document via a communication channel lets the document travel in space; storing the document on a medium and loading it later lets the document travel in time.
Unless otherwise noted, both cases are considered to be equal in this thesis: for example, the term 'sender' refers either to the entity which sends the document via the communication channel or to the entity which stores the document on the storage medium.

2.1 Security services

The “ISO Security Architecture for OSI Systems” defines the term “security service” as follows:

security service: “A service, provided by a layer of communicating open systems, which ensures adequate security of the systems or of data transfers.” [ITU-T X.800 | ISO 7498-2]

In the following sections, different security properties are introduced, e.g. ‘confidentiality’, ‘data integrity’ or ‘data origin authentication’. For each of these properties, a corresponding security service exists which provides entities with a service ensuring the property:

confidentiality security service: A service, provided by a layer of communicating open systems, which ensures adequate confidentiality of systems or of data transfers. (For the definition of ‘confidentiality’, see page 7).

traffic flow confidentiality security service: A service, provided by a layer of communicating open systems, which ensures adequate traffic flow confidentiality of data transfers. (For the definition of ‘traffic flow confidentiality’, please refer to page 15).

data integrity security service: A service, provided by a layer of communicating open systems, which ensures adequate data integrity of systems or of data transfers. (For the definition of ‘data integrity’, please refer to page 18).

peer entity authentication security service: A service, provided by a layer of communicating open systems, which ensures adequate peer entity authentication of systems or of data transfers. (For the definition of ‘peer entity authentication’, please refer to page 21).

data origin authentication security service: A service, provided by a layer of communicating open systems, which ensures adequate data origin authentication of data transfers. (For the definition of ‘data origin authentication’, please refer to page 21).

2.2 Confidentiality

When people (or more generally “*entities*”) write down information, sometimes they need protection of the information so that it cannot be read by everybody. Therefore, methods to hide this information and to reveal it only to authorized people are needed.

2.2.1 Definitions

The "ISO Security Architecture" [ITU-T X.800 | ISO 7498-2] and the "ISO Confidentiality framework" [ITU-T X.814 | ISO 10181-5] contain various useful definitions of confidentiality and related terms:

confidentiality: "The property that information is not made available or disclosed to unauthorized individuals, entities, or processes." [ITU-T X.800 | ISO 7498-2]

plaintext: "Intelligible data, the semantic content of which is available." [ITU-T X.800 | ISO 7498-2] The terms 'cleartext' and 'plaintext' are used synonymous throughout this document.

ciphertext: "Data produced through the use of encryption. The semantic content of the resulting data is not available." [ITU-T X.800 | ISO 7498-2]

confidentiality protected environment: "An environment which prevents unauthorized information disclosure either by preventing unauthorized data inspection or by preventing unauthorized derivation of sensitive information through data inspection. Sensitive information may include some or all of the data attributes (e.g. value, size, or existence)." [ITU-T X.814 | ISO 10181-5]

confidentiality protected data: "Data within a confidentiality protected environment. A confidentiality protected environment may also protect some (or all) of the attributes of the confidentiality protected data." [ITU-T X.814 | ISO 10181-5]

confidentiality protected information: "Information all of whose concrete encoding (i.e. data) are confidentiality protected." [ITU-T X.814 | ISO 10181-5]

The first part of the definition for the confidentiality protected environment is very intuitive: inspection of the plaintext by unauthorized entities is to be forbidden.
It can be necessary to protect attributes associated with the plaintext, rather than the only plaintext itself. Such attributes include the plaintext's size, its position in a document or even the existence of the plaintext. For instance, the existence of a particular file in a storage system or the existence of a communication connection between two entities can disclose information to an attacker. Preventing the disclosure of the existence of communications is called 'traffic flow confidentiality' and is discussed in "Traffic Flow Confidentiality" on page 15.
The *size* of the data can be also confidential. The number of octets of a protected file in a storage system or the transferred amount of data between two entities also discloses information to the attacker.

2.2.2 Ways to disclose information

In an information processing system, information is represented by data items which are stored or processed in the system or transferred between system entities. Disclosure of sensitive information happens by deriving it from the data. This can happen in various ways [ITU-T X.800 | ISO 7498-2]:

- by reading the value of the data, i.e. direct access to the plaintext octets,
- by having access to attributes of the data, e.g.
  - the existence (or non-existence) of the data,
  - the size of the data or duration of transfer,
  - the identity of the data owner (or entities involved in a transfer) or
  - relevant time attributes, e.g. date of creation, date of last update, date of last read access or time of transfer
- by deriving information from context information, i.e. other data object associated with the data in question,
- by observing the dynamic variations of the representation and
- by observing the reaction of the entities involved in a communication after sending or receiving a given piece of data.

2.2.3 Types of confidentiality security mechanisms

A confidentiality security mechanism provides protection against the unauthorized disclosure of information. The “ISO Confidentiality Framework” defines two general types of confidentiality security *mechanisms*:

1. The read access to the data can be restricted and prevented.
   - Access restrictions are usually enforced by *access control mechanisms*, defined in the “ISO Access control framework” [ITU-T X.812 | ISO 10181-3]. The access control mechanism grants access to the information only to authorized entities.
   - Access prevention can be done using physical access prevention to storage locations and transfer mediums, e.g. through physically shielded cables, tamper resistant (hardened) smart cards or security guards to protect entry to the information processing buildings.
2. A transformation algorithm and the so called *hiding confidentiality information* (HCI) are used to transform the plaintext data into a form which is only accessible to those who possess the corresponding critical *revealing confidentiality information* (RCI). Such transformations include:

- *Encryption* (also called *encipherment*) mechanisms which render the value of the data (the semantics) unreadable.
- *Data padding* mechanisms which hide the size of the data. In order to be effective, a second confidentiality security mechanism (such as encryption) is necessary to prevent an attacker from distinguishing plaintext data from padding data.
- *Spread spectrum* mechanisms can be used to hide the existence of a communication channel.
- *Quantum cryptography* is a relatively young discipline which utilizes the 'Heisenberg uncertainty principle of quantum mechanics'. Quantum cryptography creates communication channels in which an eavesdropper destroys the transmitted information with any attempt to eavesdrop the channel. This enables the communicating entities involved in a quantum based communication to detect the presence of the attacker.

2.2.4 Cryptographic algorithms for confidentiality security mechanisms

Throughout this document, two confidentiality security mechanisms will be dominant: access control and encryption. Access control will be described in a later section.

2.2.4.1 Encryption mechanisms

Encryption mechanisms are transformations which transform plaintext into ciphertext and vice versa. *Plaintext* is the intelligible data (readable content) that has to be rendered unreadable. *Ciphertext* is the encrypted plaintext for which no semantic content is available. An encryption mechanism transforms plaintext into ciphertext. A decryption mechanism transforms ciphertext back into plaintext.
In order to be able to re-use a particular algorithm in many systems, the algorithm is parameterized by a *key*.

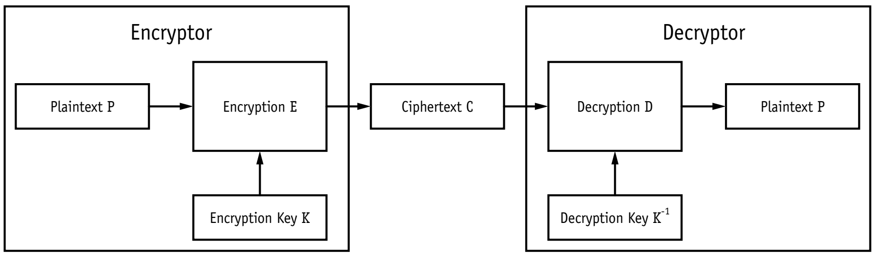


Figure 2-1: Generic encryption system

key: "A sequence of symbols that controls the operations of encryption and decryption." [ITU-T X.800 | ISO 7498-2] The ISO standard uses the terms 'encipherment' and 'decipherment', but for consistency of this document, encryption and decryption are used.

For encrypting a plaintext P, the encryption algorithm E is parameterized with a confidentiality encryption key K. The ciphertext C is decrypted using the confidentiality decryption key K^{-1}.
Encrypting a plaintext P under the key K produces the ciphertext C and is denoted as $C = E_K(P)$. The decryption of C under the key K^{-1} reproduced the plaintext P and is denoted as $P = E_{K^{-1}}(C)$.
Cryptography knows two different types of encryption systems: *symmetric* encryption systems and *asymmetric* encryption systems:

2.2.4.2 Symmetric encryption systems

symmetric encryption system: "Encryption system based on symmetric cryptographic techniques that use the same secret key for both the encryption and decryption algorithms". [ISO 18033-1]

symmetric cryptographic technique: "Cryptographic technique that uses a shared secret key.
(Examples of symmetric cryptographic techniques include symmetric ciphers and Message Authentication Codes (MACs). In a symmetric cipher, the same secret key is used to encrypt and decrypt data. In a MAC scheme, the same secret key is used to generate and verify MACs.)" [ISO 18033-1]

Given these definitions, a *symmetric encryption system* has the property that both the encryption key and the decryption key have the same value, i.e. $K = K^{-1}$. Symmetric keys are also called *secret key.*

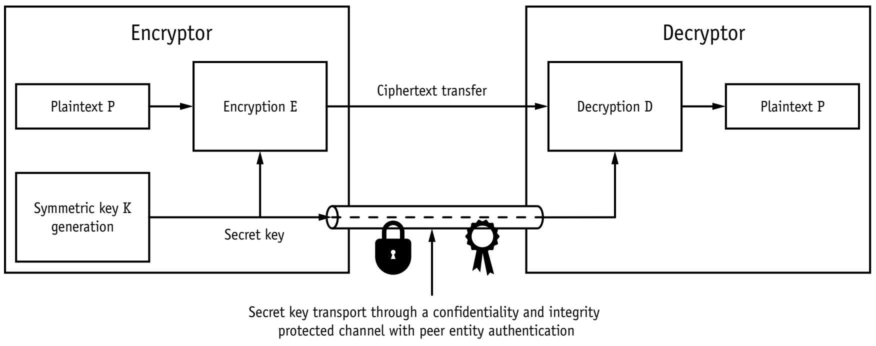

Encryptor
Decryptor
Plaintext P
Encryption E
Ciphertext transfer
Decryption D
Plaintext P
Symmetric key K generation
Secret key
Secret key transport through a confidentiality and integrity protected channel with peer entity authentication

Figure 2-2: Example of a symmetric encryption system

secret key: "A key that is used with a symmetric cryptographic algorithm. Possession of a secret key is restricted (usually to two entities)." [ITU-T X.810 | ISO 10181-1]

symmetric cryptographic algorithm: “An algorithm for performing encryption or the corresponding algorithm for performing decryption in which the same key is required for both encryption and decryption.” [ITU-T X.810 | ISO 10181-1]

The secret key can be generated in various ways:

- ❑ by the encrypting entity (called *encryptor*) like shown in figure 2-2,
- ❑ by the *decryptor*,
- ❑ the key is generated by a trusted third party (TTP) like a key distribution center (KDC) or
- ❑ it can be derived from parameters in a key agreement protocol which is performed by both encryptor and decryptor.

If the key is not computed using a key agreement protocol, the key must be transported through a secure channel which is confidentiality and integrity protected. Additionally, the recipient(s) of the secret key must know the source of the key, i.e. data origin authentication for the transported key is necessary.
The ciphertext itself can be transported through an unprotected channel.

2.2.4.3 Symmetric encryption algorithms

Algorithms which perform symmetric encryption are grouped into two classes:

- ❑ *block ciphers* and
- ❑ *stream ciphers*.

A *block cipher* processes blocks of plaintext to create blocks of ciphertext. A block is a string of n bits. Such a block cipher is called n *bit block cipher*. Typical algorithms which are used in today’s systems are

- ❑ 3DES (Triple DES (Data Encryption Standard), also known as TDEA – Triple Data Encryption Algorithm),
- ❑ IDEA (International Data Encryption Algorithm),
- ❑ AES (Advanced Encryption Standard)
- ❑ and various other block ciphers like the other AES candidates (e.g. Blowfish or RC6).

The plaintext bit sequence is segmented into n bit blocks, as the block cipher needs n bit as input. To allow cases where the input length is not a multiple of n bit, a *padding mechanism* is usually used with a block cipher. The padding algorithm defines an unambiguous way how each plaintext is extended to a length of a multiple of n bit. This is even done if the length of the plaintext is already a multiple of n bit. So if a padding mechanism is used, the length of the ciphertext is larger than the length of the plaintext. After

decrypting with the block cipher, the padded bits are removed from the decrypted data.
Mechanisms closely related to block ciphers are modes of operation (MoO or modes). Modes define how the inputs and outputs of consecutive block cipher operations are combined. This is done to ensure that the same plaintext block results in different ciphertext blocks throughout the ciphertext stream and to chain the blocks together to prevent substitution attacks.
A *stream cipher* combines a sequence of plaintext symbols with a sequence of keystream symbols, one symbol at a time, and using an invertible function (for single bits as a symbol, the function is usually an exclusive or between keystream bit and plaintext bit). Typical stream cipher algorithms are RC4, A5 are n bit block ciphers which operate in a specific mode to create a key symbol stream.

2.2.4.4 Asymmetric encryption systems

asymmetric encryption system: "Encryption system based on asymmetric cryptographic techniques whose public transformation is used for encipherment and whose private transformation is used for decipherment." [ISO/IEC 9798-1]

asymmetric cryptographic technique: "Cryptographic technique that uses two related transformations, a public transformation (defined by the public key) and a private transformation (defined by the private key). The two transformations have the property that, given the public transformation, it is computationally infeasible to derive the private transformation." [ISO/IEC 11770]

In an *asymmetric encryption system*, the encryption key K (also called *public key*) and the decryption key K^{-1} (also called *private key*) have distinct values, but these both values do have a mathematical relationship which is defined by the underlying cryptographic algorithm:

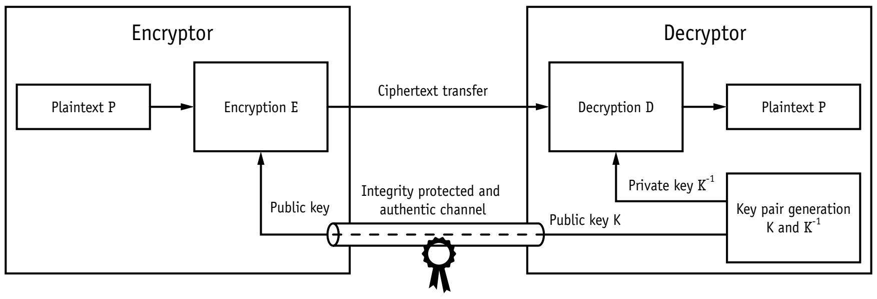


Figure 2-3: Example of an asymmetric encryption system

private key: “A key that is used with an asymmetric cryptographic algorithm and whose possession is restricted (usually to only one entity).” [ITU-T X.810 | ISO 10181-1]

public key: “A key that is used with an asymmetric cryptographic algorithm and that can be made publicly available.” [ITU-T X.810 | ISO 10181-1]

A *key pair* consists of a public and the corresponding private key. The generation of a key pair can be performed by different parties:

- ❑ The key pair can be generated by the decryptor.
  In that case, the decryptor can publish the public key in a directory service or directly send the public key to the encryptor. This case is shown in figure 2-3 on page 12. *Note that the channel for transporting the public key does not have to be confidentiality protected.*
- ❑ The key pair can be generated by a trusted third party.
  In that case, the private key must be transmitted to the decryptor via a confidentiality protected channel.

The private key must be protected by the decryptor. Regardless which entity undertakes the key pair generation, the public key must be made available to the encryptor. The encryptor must be confident that the public key belongs to the decryptor. This can be achieved using digital certificates (if a trusted third party is available) or by transport through integrity protected channels with data origin authentication enabled.

2.2.4.5 Asymmetric encryption algorithms

The most common used asymmetric encryption algorithm is the RSA algorithm, named after its inventors Rivest, Shamir and Adleman [RSA78]. RSA is based on the difficulty of factoring large integers.

2.2.5 Key Management

key management: “The generation, storage, distribution, deletion, archiving and application of keys in accordance with a security policy.” [ITU-T X.800 | ISO 7498-2]

In this section, the generation and distribution of keys is described. The encryption system examples in figure 2-2 on page 10 and figure 2-3 on page 12 illustrate examples on where keys can be generated and how they can be distributed.
The example in figure 2-2 implicitly assumes that (1) the sender generates the symmetric secret key and that (2) a confidentiality protected, integrity protected and peer entity authentication enabled communication channel exists between both parties.

key establishment: “A process or protocol whereby a shared secret becomes available to two or more parties, for subsequent cryptographic use. Key establishment may be broadly subdivided into ‘key transport’ and ‘key agreement’.” [MOV96]

key transport: “A key transport protocol or mechanism is a key establishment technique where one party creates or otherwise obtains a secret value, and securely transfers it to the other(s).” [MOV96]

Both example figures do illustrate key transport mechanisms. In figure 2-2 on page 10, a secret key is generated by one party (the sender in the example). This key is securely transferred to the other party (the receiver). This secure transfer could be done by encrypting the secret key under the public key of the receiver. RSA-based key transport schemes like RSAES-PKCS1-v1_5 and RSAES-OAEP-ENCRYPT described in [KaSt98] fall into this category.

key agreement: “A key agreement protocol or mechanism is a key establishment technique in which a shared secret is derived by (or more) parties as a function of information contributed by, or associated with, each of these, (ideally) such that no party can predetermine the resulting value.” [MOV96]

Key agreement schemes can be Diffie-Hellman based protocols like described in [Resc99] or based on symmetric techniques, e.g. like the Kerberos system [KoNe93].

2.2.6 Pseudo random bit generation

Many cryptographic algorithms require random bits, e.g. for the generation of keys like secret keys or as time variant parameters for the use during cryptographic protocols (nonces). A random number generator (RNG) is useable for cryptographic use if the generated random bits cannot reproduced other than by chance. Additionally, the generated bits must withstand commonly established statistical tests.
The ISO Project 1.27.31 currently (May 2003) works on a not yet finished standard ISO/IEC 18031 “Information technology - Security techniques - Random bit generation”. This draft document defines the following terms:

computationally infeasible: “A problem, which is deemed impractical to solve. Theoretically, this means requiring computational resources, growing faster than any polynomial in size of the input. Pragmatically, it means requiring computational resources 2^s, where s is a sufficiently large security parameter of a cryptographic system.”

deterministic: “This term defines a characteristic of an algorithm. Given a same set input will result in a known output.”

non-deterministic bit stream: "For the purposes of this standard, non-deterministic is defined as an output stream of bits produced as a result of some unpredictable phenomena or activity."

pseudo-random bit generator: "A deterministic algorithm which when given some form of a bit sequence length k outputs a sequence of bits of length $l > k$, computationally infeasible to distinguish from true random bits."

random bit generator: "A device or algorithm that produces a stream of bits where those sequences are statistically close to having a uniform distribution."

random number: A sub-string of bits that have been converted to some interpretable number in a predetermined interval (e.g., 0011_2 equals 3_{16}).

A random number generator requires a physical source of randomness, e.g. a radioactive radiation source or another quantifiable physical phenomenon. Building a random source using solid physical sources is very expensive; for that reason, current systems gather random bits from multiple measurable events (timing of user keystrokes, mouse movements, CPU utilization, hard drive timings and network traffic characteristics) as input seed and use a cryptographically secure pseudo-random bit generator to calculate pseudo-random bits out of this input. Today, strong mixing functions are used as pseudo-random bit generators. Such strong mixing functions are e.g. hash functions like SHA-1 or RIPEMD160 or block ciphers like AES or 3DES.

2.3 Traffic Flow Confidentiality

One form of confidentiality is the so called "*traffic flow confidentiality*", also known as "*prevention of traffic flow analysis*". Traffic flow confidentiality is defined in the "ISO Security architecture" [ITU-T X.800 | ISO 7498-2]:

traffic analysis: "The inference of information from observation of traffic flows (presence, absence, amount, direction and frequency)." [ITU-T X.800 | ISO 7498-2]

traffic flow confidentiality: "A confidentiality service to protect against traffic analysis." [ITU-T X.800 | ISO 7498-2]
"Traffic flow confidentiality provides for the protection of the information which might be derived from observation of traffic flows." [ITU-T X.814 | ISO 10181-5]

traffic padding: "The generation of spurious instance of communication, spurious data units and/or spurious data within data units." [ITU-T X.800 | ISO 7498-2]

2.3.1 Security mechanisms for traffic flow confidentiality

The “ISO Confidentiality framework” introduces two mechanisms to prevent traffic flow analysis: (1) *data padding* and (2) *dummy events*:

2.3.1.1 Confidentiality provision through data padding

> *“The purpose of this mechanism is to prevent knowledge of the information represented by the size of a data item. This mechanism increases the size of data items so that the size of a padded data item bears little relation to its original size. One way to do this is to add random data to the beginning or the end of the data item. This must be done in a way that the padding is recognizable as such by authorized entities but is indistinguishable from the data by unauthorized entities. In order to achieve this, data padding can be used in conjunction with cryptographic transformations.”* [ITU-T X.814 | ISO 10181-5]

According to that definition, data padding changes the size of each data item, regardless whether the data item is payload data or a dummy event.

2.3.1.2 Confidentiality provision through dummy events

> *“The purpose of this mechanism is to prevent inferencing based on the rate that a given event occurs. An instance of this mechanism can be found in network layer security protocols that seek to hide the volume of traffic exchanged over untrusted links.*
> *This mechanism produces pseudo events, e.g., bogus protocol-data-units (PDU) that only authorized parties can identify as such. This mechanism can be used to counter covert channel attacks that perform signaling based on variations in the rate of an activity.”* [ITU-T X.814 | ISO 10181-5]

According to that definition, dummy events change the overall number of data items, i.e. dummy data items are used together with the payload data items.

2.3.1.3 Examples

The security service *traffic flow confidentiality* offers confidentiality for both the *value* of the data and for the *context* of a data item, i.e. it does protect the data attributes, the fact that a communication takes place.
“*Traffic flow analysis*” is a passive attack against communication networks. Such a communication network could be an electronic network like the internet or a public switched telephone network (PSTN), but even a group of couriers carrying messages through a war battlefield.
This kind of attack consists of observing as much as possible of the traffic in a given network and analyzing the given communication patterns according to different criteria.

In contrast to intercepting complete messages (transport protocol information and payload data), the traffic analysis primary focuses on transport and routing information. This is based on the fact that the payload data itself is encrypted in many situations, e.g. encrypted E-mail attachments, encrypted TLS connections or encrypted IP packets using IPSec.
The extracted communication patterns can be used to draw conclusions on the involved parties of a communication. If also external events are taken into account (like "What actions have been taken by the recipient after receiving this information?"), the contents of a message can possibly deduced.

❑ The existence or the absence of a communication yields to information about potential communication partners.

❑ The amount, direction and frequency of transmitted information can yield to knowledge about the contents or the type of the information, if the attacker knows additional context about the communication or the involved entities (like external events performed by the entities).

To perform a traffic flow analysis, the attacker must be able to split the intercepted traffic into single chunks of data; the bit sequence must be structured. The manner of this structuring process is defined by the underlying network protocol (e.g. ATM, Ethernet or PPP, TCP/IP). Based on the used protocol stack, information of different protocol layers becomes available. Without such a structuring process, the traffic analysis would not be possible.
One countermeasure to traffic analysis is the "*link encryption*", which completely encrypts a communication link between two network elements. A link encryption mechanism encrypts all traffic (including synchronization information, message header information and data payload) prior sending it onto the communication channel. This single, continuous application of an encryption algorithm hides the visible boundaries (the structure) between single protocol messages. Link encryption mechanisms are used on synchronized communication links, where each clock cycle is encrypted.
In case that no messages are queued for transfer, randomized messages (or octets) are inserted (stuffed) into the plaintext traffic (traffic padding). This measure makes prevents an attacker to determining whether messages are transmitted or not.
Link encryption transforms a sequence of protocol messages which are transmitted on a single link into an encrypted octet sequence without visible structure.

2.3.2 Analogies between network traffic and structured data

Plaintext messages have an inner structure which is defined by the processing application. There are exact rules how to interpret e.g. particular octets of a PostScript file or an MPEG movie. Given today's landscape of different applications, a numberless amount of different file formats exists.
In cryptography, the plaintext of a message is treated as a sequence of bits, whose inner structure is unknown, dispensable and opaque to the encryption algorithm. This assumption makes is possible to design generic cryptographic

algorithms which do not have to consider the type of the plaintext, but rather work on arbitrary bit or octet sequences.
The widespread use of XML in many applications results in many new file and data formats whose inner structure is given by the XML syntax. This leads to the requirement to specific encryption systems which do not simply treat the XML structure as a generic octet sequence but as structured information. A specific encryption system for structured data like XML brings up new benefits for the application domain like "*selective field confidentiality*":

selective field confidentiality: "This service provides for the confidentiality of selected fields within the (N)-user-data on an (N)-connection or in a single connectionless (N)-SDU (service-data-unit)." [ITU-T X.800 | ISO 7498-2]

Structuring information with a generic format like XML leads to data items which have similar protection requirements like a communication link, because an attacker has access to the structure information.
The encryption of a complete XML instance with a general purpose encryption algorithm without taking the inner XML structure into account is similar to link encryption: the inner structure is completely invisible. W3C XML Encryption [ER02] enables the encryption of parts of an XML instance, which would correspond to encrypting selected PDUs on a link. What W3C XML Encryption misses is the concept of data padding and dummy events in order to hide the number, size and structure of encrypted portions in an XML instance.

2.4 Data integrity

The security service data integrity aims to protect data items against accidental and intentional changes and modifications while the data is being stored, processed or transmitted. For achieving the goal that data cannot be modified, physical protection of the data is required, e.g. by write access control to mediums or by using protected transfer mediums. In many cases, attackers have access to the data so that the goal that "data cannot be modified" cannot be achieved. This access can happen in various ways: by having physical access to storage mediums like a local hard drive, when the data is stored in the RAM of a computer, by gaining control over the software which processes the data or when the data is transferred over network media which the attacker has access to. If alterations of the data cannot be prevented, at least it should be possible to detect *whether* an alteration has happened.

data integrity: "The property that data has not been altered or destroyed in an unauthorized manner." [ITU-T X.800 | ISO 7498-2]

The creator of the data applies a *shielding mechanism* to transform data into *integrity protected data*. Using a *validating mechanism*, the receiving entity

(*validator*) can check whether the data is in its original form or whether an unauthorized or accidental alteration modified the data. Using an *unshielding mechanism*, the validator can regenerate the data from the integrity protected data.
As the shielding process usually concatenates the data with some cryptographic value, the unshielding operation is the removal of the attached value after validation of the integrity.

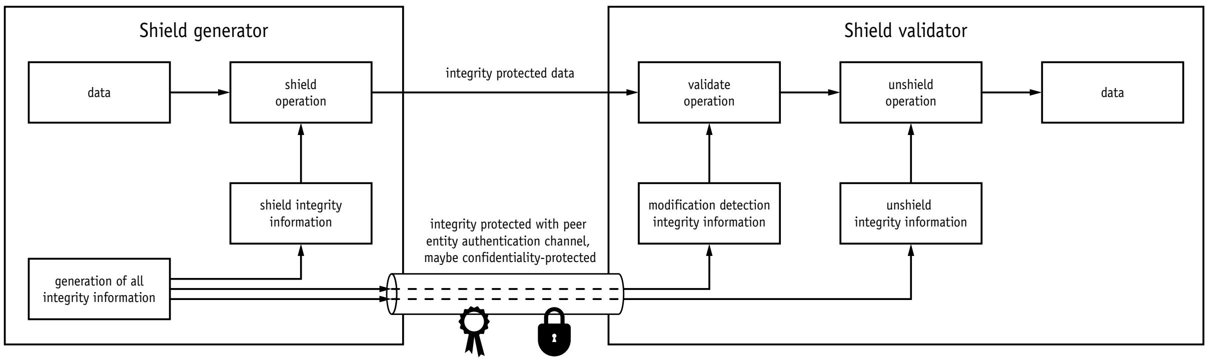

Shield generator
data
shield operation
shield integrity information
generation of all integrity information
integrity protected data
integrity protected with peer entity authentication channel, maybe confidentiality-protected
Shield validator
validate operation
unshield operation
data
modification detection integrity information
unshield integrity information

Figure 2-4: Data integrity mechanisms

2.4.1 Types of data integrity services

Different data integrity security services, including connection-oriented (CO) and connectionless (CL) integrity exist, which can also be applied on stored data. A connection-oriented data integrity service provides integrity for all user data and is able to detect modifications, insertion, deletion and replay of any data within the connection. A connectionless data integrity service is able to protect the integrity of single data packets (service data units - SDUs), but cannot detect insertion, deletion and replay of packets or changes in the sequence ordering.

2.4.2 Data integrity mechanisms

This document only refers to integrity mechanisms which are provided through cryptographic means:

❑ *Integrity provision through sealing* is the application of a symmetric algorithm to the data which involves a shared secret (symmetric) key. The result of the sealing is attached to the data. For both shielding and validating, both sides need the same secret key. The computation of the value can be done in different ways:

  ❍ by symmetric encryption of a *hash value* of the data (see definition of hash functions for hash values),

  ❍ by creating a *message authentication code* (MAC) using an n bit block cipher or

- ❍ by creating a HMAC (i.e. a MAC based on a keyed cryptographic hash function) [KrBeCa97].

- ❑ *Integrity provision through digital signatures* (see “digital signature” on page 22) is the application of a digital signature algorithm to the data. The shielding operation requires the private key, while the validation operation requires the public key of the shield generator.

2.4.3 Data integrity algorithms

hash function: “A (mathematical) function that maps values from a (possibly very) large set of values into a smaller range of values.” [ITU-T X.810 | ISO 10181-1]
The values in the ‘smaller range of values’ are called ‘hash values’.

one way function: “A (mathematical) function that is easy to compute but, when knowing a result, it is computationally infeasible to find any of the values that may have been supplied to obtain it.” [ITU-T X.810 | ISO 10181-1]

one way hash function: “A (mathematical) function that is both a one way function and a hash function.” [ITU-T X.810 | ISO 10181-1]
A one way hash function is also called c*ryptographic hash function* or *message digest function*.

A cryptographic hash function h calculates hash values $h(m)$ from a message m. The message m is a bit string with a defined length.
Given that h is a collision resistant hash function (see [MOV96, pp. 325]), a new hash function g can be defined with $g(m) = h(h(m))$ which is also a collision resistant hash function (chaining by the double application of h).
Additionally, a collision resistant hash function H can be defined which can calculate a hash value of a finite set of messages $m_1 \dots m_i$:

$$H(m_1, m_2, \dots, m_i) = h[h(m_1) \| h(m_2) \| \dots \| h(m_i)]$$

(In the above formula, $a \| b$ denotes the concatenation of the value a with the value b.) The hash function H calculates a hash value which depends on both content and ordering of the input messages. Each message m_i can have one or more additional attributes, represented by a_i. Such an additional attribute could be the address of the message or the message’s creation date. The additional attributes can be included in the hash value calculation as follows:

$$H(m_1, a_1, m_2, a_2, \dots, m_i, a_i) = h[h(m_1) \| a_1 \| h(m_2) \| a_2 \| \dots \| h(m_i) \| a_i]$$

A change or reordering in any input will change the overall hash value.
Current cryptographic hash functions include the "Secure Hash Algorithm" SHA-1 defined in the Secure Hash Standard [FIPS180-1] and RIPEMD-160 [ISO/IEC 10118-3].

2.5 Authentication

The security service *authentication* provides assurance about the claimed identity of an entity. In authentication, a *principal* aims to prove its identity to a *verifier*. After a successful authentication process, the principal's identity becomes an *authenticated identity*.
Two important classes of authentication are known:

peer entity authentication: *"The corroboration that a peer entity in an association is the one claimed."* [ITU-T X.800 | ISO 7498-2]

data origin authentication: "The corroboration that the entity responsible for the creation of a set of data is the one claimed." [ITU-T X.800 | ISO 7498-2]

message authentication: *"The property, given an authentication code/protected checksum, that tampering with both the data and checksum, so as to introduce changes while seemingly preserving integrity, are still detected."* [ERS02]

For using "*peer entity authentication*", the principal (represented by a *claimant*, e.g. a computer process) has a communications relationship to the verifier, i.e. it is intended for connection-oriented communication. At some instant of time during the communication between principal and verifier, the verifier has to prove its identity to the verifier. So at this point in time, the verifier has assurance that his communication partner is the claimed principal; further measures must be taken to ensure the continuity of the authentication during the whole following communication. Peer entity authentication can be performed unilateral by only one of the communicating entities, or by both so that mutual peer entity authentication is performed.
By using the security service "*data origin authentication*", the verifier has assurance that the principal is the source of a data item in question. Data origin authentication mechanisms implicitly provide that the authenticated data item is integrity protected.
Besides using the data origin authentication service for securing transmitted data (in both the connection-oriented and the connection-less case), this service can also be used to protect documents which are not transmitted but stored.

2.5.1 Authentication mechanisms

message authentication code: "A cryptographic checkvalue that is used to provide data origin authentication and data integrity." [ITU-T X.813 | ISO 10181-4]
Note: Both data integrity and data origin authentication can only be provided for the receiving entity. A third party cannot verify these properties, as both sender and receiver are capable to create the MAC (or HMAC).

digital signature: "Data appended to, or a cryptographic transformation of a data unit that allows a recipient of the data unit to prove the source and integrity of the data unit and protect against forgery, e.g. by the recipient." [ITU-T X.800 | ISO 7498-2]
The standard mechanisms for digital signatures are RSA [RSA78] and DSA [FIPS186-2].

Authentication mechanisms like digital signatures can be used to provide data origin authentication. The OSI standards define these mechanisms to be used for data items transmitted over networks. These mechanisms can by their very nature also be used to protect data items stored on a local storage medium (e.g. a hard drive) or in a database. Both 'data origin authentication' and 'data integrity' in the context of data storage means that the entity which created or stored the data item can apply these mechanisms to protect the stored data item.

2.5.2 Authentication protocols

Authentication protocols are necessary to provide peer entity authentication:

authentication protocol: to provide to one party some degree of assurance regarding the identity of another with which it is purportedly communicating. [MOV96]

An authentication protocol is a sequence of message exchanges between two or more entities, to corroborate the identity of one or more of these entities. Examples for authentication protocols are

- one-pass unilateral authentication protocols, e.g. password or digital signature based,
- two-pass mutual authentication protocols, e.g. challenge-response identification [Ramos98] using a shared secret key or digital signature based
- three-pass authentication protocols like zero-knowledge proofs, e.g. the Fiat-Shamir or Schnorr identification protocols.

A zero-knowledge proof is a technique by which possession of information can be verified without any part of that information being revealed.

2.6 Access Control

access control: The prevention of unauthorized use of a resource, including the prevention of the use of a resource in an unauthorized manner. [ITU-T X.800 | ISO 7498-2]

In access control, an initiator aims to access a target (the resource). An *'initiator'* (also called 'subject') is an entity (e.g. human user or computer based entity) that attempts to access other entities [ITU-T X.812 | ISO 10181-3]. The '*target'* (also called object) is an entity to which access may be attempted. The *'access request'* can be one in a set of different operations, for example a read access to the contents of a file, a write or change access on that file or the execution of a program. The access request is handled by the *'access control enforcement function'* (AEF, sometimes called '*reference monitor'*). The *'access control decision function'* (ADF) decides whether the initiator is authorized and access to the target is granted or not.

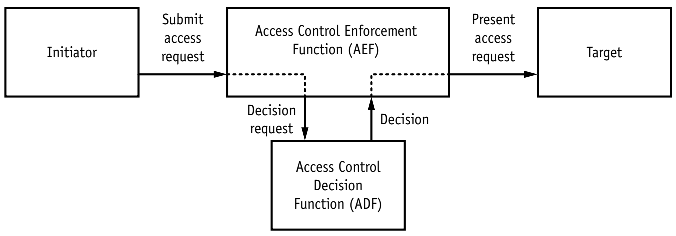


Figure 2-5: Fundamental access control functions

The operations can be sorted into two major groups:

- ❑ *observing access*, i.e. read only access where the object is not changed and
- ❑ *altering access*, i.e. writing or modifying access where the object or attributes of the object are changed.

Imposing restrictions on read only access is usually done in order to maintain the confidentiality of the object while altering access restrictions protect the integrity and authenticity of the stored information. The interesting point on access control is that the 'access control enforcement function' (AEF) has full (read) access to the complete set of data items and associated attributes which are stored on the target server. The AEF is a trusted process which only passes information to the initiator which the initiator is allowed to see.
Access control systems are divided into two main classes: '*discretionary access control*' (DAC) and '*mandatory access control*' (MAC). Discretionary access control, also called *identity-based access control* (IBAC), bases the access rights on the identity of subjects and objects involved. The owner of an object constrains the access by granting access only to specific set of subjects. In mandatory access control systems, also known as *rule-based access con-*

trol, the access control decision is taken by the operating system based on a given set of rules. In mandatory access control, the individual users do not set permissions for their objects.
As this document deals with confidentiality, the term 'access control' refers to observing access (read only) throughout this document.

2.7 Plausible Deniability

Security services describe often only a single side of a double sided coin. Security services provided for one entity are undesirable for other entities. Defining who the user of a system is and who an adversary depends on the point of view. For example, encrypting a database of members of the underground movement in a suppressive regime is indispensable for members of this group but is a serious threat for the military in the country.

plausible deniability: "Prevent that irrefutable evidence concerning the occurrence or non-occurrence of an event or action exists." [Roe97]

Many applications require *non repudiation* as a mandatory security service, as it provides confidence for business transactions by generating evidence that a particular event (like signing an order) has taken place. 'Non repudiation' can be seen as the opposite to 'plausible deniability'.
In some situations, such evidence is not desired and may even be a serious security threat: when a user can become under coercion by an adversary:

❑ In a suppressive police state, finding traces of a subversive document on an opponent's hard drive can cause great problems for that person.

❑ Robbers can coerce a victim to disclose the PIN of a banking card.

❑ Custom authorities could require travelers to decrypt any encrypted material found on the traveler's hard drive, in order to find subversive material or trade secrets [And01, pp. 442].

In situations like above, the security service "*plausible deniability*" could help the user: by having a way to plausibly deny the existence of the material in question. 'Plausibly' means that the adversary has no irrefutable evidence about the event in question, so that it cannot be proved that the event occurred respectively did not occur. A good thesis on plausible deniability by Michael Roe can be found in [Roe97]. Plausible deniability can be achieved by multiple mechanisms:

❑ Different steganographic mechanisms exist for embedding material in multimedia files like bitmap images or sound files.

❑ The use file systems which support plausible deniability like described in [ANS98].

❑ The use of encryption systems:

- By using an encryption mechanism which can decrypt the ciphertext to different plaintexts, depending whether the decryptor is under coercion by the adversary or not. If the decryptor is not under coercion, the correct key is used to decrypt the real message. If an adversary coerces the decryptor to decrypt the ciphertext or to reveal the key, an alternative key is used in order to reconstruct an innocuous message. A suitable encryption mechanism for this method is the one time pad.
- The creation of *dummy messages* by encrypting arbitrary random data and destroying the keys allows the decryptor to plausibly deny to decrypt a particular ciphertext, claiming that this ciphertext is not ciphertext but a random dummy message. In order to let this approach work, a 'regular' ciphertext must be indistinguishable from a dummy message.

3 Introduction to XML

The "*eXtensible Markup Language*" (XML) is a standard which describes a syntax for structuring data and documents. The XML recommendation was first published in February 1998 by the World Wide Web Consortium (W3C). XML is a subset of the "*Standard Generalized Markup Language*" (SGML), which is standardized in ISO 8879 [ISO8879]. The enormous complexity of SGML caused the development of XML; goal was to develop a meta language with only 10% complexity of SGML while keeping 90% of its potential. The W3C shortly describes XML in the following way [Bos99]:

- *"XML is for structuring data"*:
  XML is a text format for structuring arbitrary data. It's not a programming language. XML is a language for creating other languages. The new languages are formally defined using a schema language, e.g. W3C's XML Schema. XML supports Unicode.

- *"XML looks a bit like HTML"*:
  But it's not HTML, it only uses the same constructs like elements, attributes etc.

- *"XML is text, but isn't meant to be read"*:
  Unlike binary formats like ASN.1, it's pure text based syntax. This makes it easier for developers to understand which data is creating by their applications and debugging becomes easier. XML documents can be edited with a text editor.

- *"XML is verbose by design"*:
  Since XML is text, it has a high redundancy compared to binary encoded formats like ASN.1 or proprietary formats for a particular application. Compression can help here, if size is a problem.
  XML's verbosity has the advantage that developers and users can look into the data with standard tools like a text editor and get a good idea which information is stored in a given piece of XML.

- *"XML is a family of technologies"*:
  XML means both the XML 1.0 Recommendation and a complete family of standards (recommendations) which surround XML 1.0. This includes XLink and XPointer for extensible, fine-grained linking concepts, XHTML as successor for HTML, Scalable Vector Graphics (SVG) for vector based images, XML Signature for digital signatures and many more.

- *"XML is new, but not that new"*:
  SGML, the 'predecessor' of XML, is available since the early '80s and has been standardized by the ISO in 1986. XML is not completely new, but simply a subset of SGML.

- *"XML leads HTML to XHTML"*:
  XHTML (eXtensible HTML) is an XML application and will be the successor of the ubiquitous HTML.
- *"XML is modular"*:
  XML allows the definition of new document formats by reusing, combining and extending existing formats.
- *"XML is the basis for RDF and the Semantic Web"*:
  The Resource Description Framework (RDF), which is a metadata application proposed by the W3C, is based on XML.
- *"XML is license-free, platform-independent and well-supported"*:
  The use of XML-based data and the creation of new XML-based languages do not require the payment of license fees to the W3C. Due to the various XML parsers, XML processing tools are available on all common computing platforms. Interoperability between these platforms is given by XML's Unicode support.

The *"eXtensible Markup Language (XML) 1.0"* recommendation [BPMM+00] defines the basic syntax for XML. The "Namespaces in XML" recommendation [BHL99] extends XML 1.0 by defining a mechanism for binding elements and attributes to a specific namespace for allowing semantic separation. The "*XML Information Set*" recommendation [CoTo01] subsequently adds a 'philosophy' to XML through providing a set of definitions that are used for describing the information available in an XML document. The "*Document Object Model*" (DOM) defines a platform- and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure and style of documents [LLN+02]. The *"XML Path Language"* (XPath) [ClDe99] describes a language for addressing and selecting parts of an XML document.

3.1 XML v1.0

The "Extensible Markup Language (XML) 1.0" recommendation [BPMM+00] defines the basic syntax for XML. XML is a language to markup text based documents. Markup refers to the process of 'tagging' data and text in a way so that a description of the particular text becomes available to automated processing.
In general, XML documents are trees, called XML trees. These trees have a root node which is called the '*document node*'. The concept of a document node is a little bit abstract as this particular node has no textual representation in the XML instance, because it is the instance itself.
The basic building blocks of an XML tree are the '*element nodes*'. Element nodes (elements) can contain other elements or '*text nodes*'. Text nodes consist of one or more Unicode characters. Text nodes are always leafs of the tree. Each XML tree has exactly one element as top level element node, the so

called document element. This document element can contain an arbitrary arrangement of child elements and text node children.
Additionally to these basic constructs, there exist '*attribute nodes*' which are name/value pairs for adding metadata to elements, '*comment nodes*' which carry comments and '*processing instruction nodes*' (PI) which can be used to control the XML processing application. XML also defines entity reference nodes which are used for escaping characters which are usually used as markup.
An XML instance consists of a single document node which contains exactly one document element node and may also contain comments and processing instructions. The document element may contain arbitrary constructs.

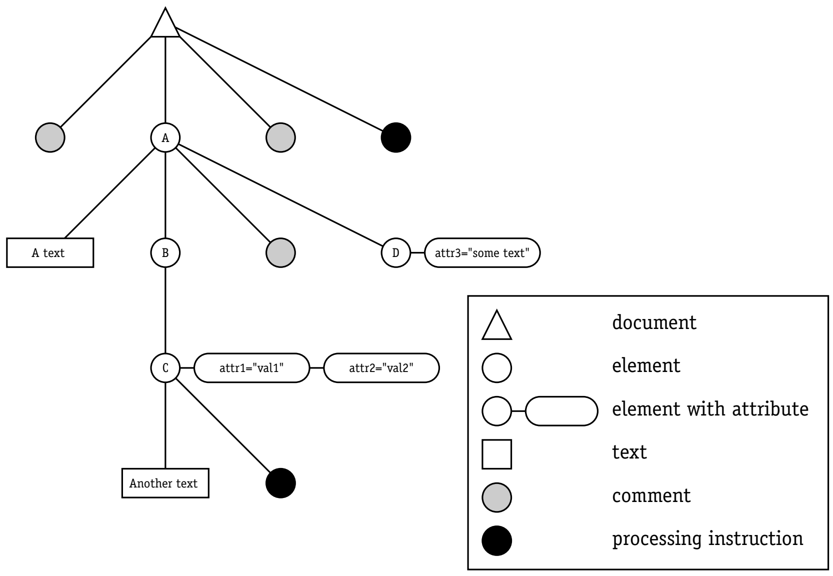


Figure 3-1: Sample XML document structure

The following XML source code gives the serialized (text) version of the previous XML tree (this example does not directly map to the above tree; in the below XML source code, there are linebreaks and other whitespace added to show the depth of an element via indentation; e.g., the `<B>` element in the tree only contains one child: the `<C>` element. In the XML source, it contains three children: a Text node, the `<C>` element and a second Text node).

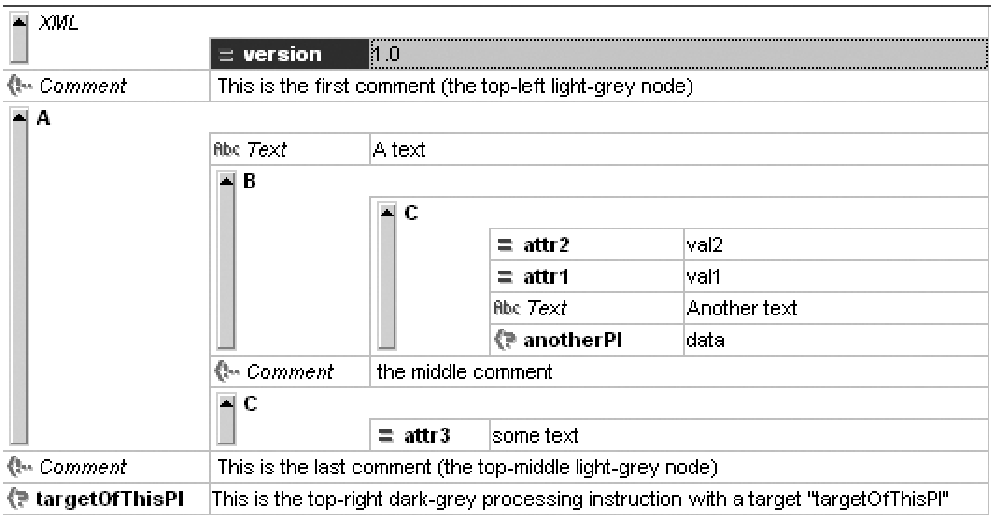
```
<?xml version="1.0"?>
<!-- This is the first comment (the top-left light-grey node) -->
<A>
   A text
   <B>
      <C attr2="val2" attr1="val1">
         Another text <?anotherPI data?>
      </C>
```

Example 3-1: XML Markup of the tree structure from figure 3-1 on page 29

```
   </B>
   <!-- the middle comment -->
   <C attr3="some text"/>
</A>
<!-- This is the last comment (the top-middle light-grey node) -->
<?targetOfThisPI This is the top-right dark-grey
                 processing instruction with a
                 target "targetOfThisPI"?>
```

Example 3-1: XML Markup of the tree structure from figure 3-1 on page 29

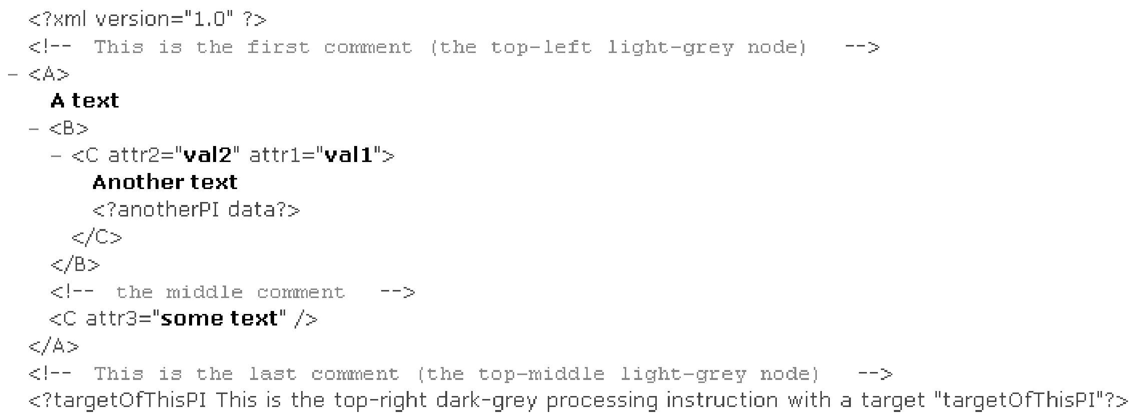
To demonstrate how a generic purpose XML aware software handles and presents this data, figure 3-2 shows how an XML Editor (Altova XML Spy) displays the XML code from example 3-1.



XML			
	version	1.0	
Comment	This is the first comment (the top-left light-grey node)		
A			
	Text	A text	
	B		
		C	
		attr2	val2
		attr1	val1
		Text	Another text
		anotherPI	data
	Comment	the middle comment	
	C		
		attr3	some text
Comment	This is the last comment (the top-middle light-grey node)		
targetOfThisPI	This is the top-right dark-grey processing instruction with a target "targetOfThisPI"		

Figure 3-2: Screenshot from the tool “XML Spy Editor”

Figure 3-3 shows how the Microsoft Internet Explorer renders a generic tree view of the example XML document.



```
  <?xml version="1.0" ?>
  <!-- This is the first comment (the top-left light-grey node) -->
- <A>
    A text
  - <B>
    - <C attr2="val2" attr1="val1">
        Another text
        <?anotherPI data?>
      </C>
    </B>
    <!-- the middle comment -->
    <C attr3="some text" />
  </A>
  <!-- This is the last comment (the top-middle light-grey node) -->
  <?targetOfThisPI This is the top-right dark-grey processing instruction with a target "targetOfThisPI"?>
```

Figure 3-3: Screenshot from the Microsoft Internet Explorer Browser

3.2 XML Namespaces

The “Namespaces in XML” recommendation [BHL99] states:

> *“XML namespaces provide a simple method for qualifying element and attribute names used in Extensible Markup Language documents by associating them with namespaces identified by URI references.”*

XML instances can combine markup from various applications in a single document. Without namespaces, this would lead to confusion if multiple applications use the same name for elements which have different semantics. For instance, the “XML Signature” and the “XML Encryption” specifications both define a `<Transforms>` element (both have the local name `Transforms`). These elements have different semantics. Namespaces help to solve this situation by binding the elements to different namespaces.
Associating a local name to a namespace is done by binding a URI [BFM98] to a prefix and using that prefix together with the local name. The binding is done with special attributes which have the form `xmlns:thePrefix="theURI"`. This binds the prefix `"thePrefix"` to the URI `"theURI"`.

3.2.1 An example with namespaces

After that declaration, the prefix can be used in conjunction with element names or attributes:

```
<rootElement xmlns:prefix1="http://www.company.com/#application1">
 <prefix1:name value="My Name" />
 <prefix2:name xmlns:prefix2="http://www.someothercompany.com/" />
 <name xmlns="http://www.education.edu">Some name from education.edu</name>
 <prefix:name xmlns:prefix="http://www.education.edu">
     A name from education.edu
 </prefix:name>
 <name>A name element without any namespace attached</name>
</rootElement>
```

Example 3-2: An example with namespaces

In example 3-2, there are 5 different name elements:

- The first name element is from the `"http://www.company.com/#application1"` namespace defined in the `<root>` element (the namespace definition is inherited therefrom)
- The 2nd name element defines its own namespace (with the `"prefix2"` prefix) which is bound to `"http://www.someothercompany.com/"`
- The 3rd name element is bound to the `"http://www.education.edu"` namespace but it does not use a prefix; the `xmlns="http://www.education.edu"` declaration defines a ‘default namespace’ which is defined for all elements which do not have a prefix.

- ❑ The 4th name element is bound to the `"http://www.education.edu"` namespace and is semantically equivalent to the 3rd sample, only the syntax (default namespace vs. prefix) differs.
- ❑ The last name has no namespace attached (no prefix and no default namespace inherited from an ancestor).
  Note: The default namespace defined in the 3rd name element is not inherited because they are siblings, not ancestors or descendants.

3.2.2 Namespaces for Attributes

A default namespace only applies to elements. Attributes which are not prefixed do not belong to any namespace. Binding an attribute to a namespace can only be done by prefixing it like in example 3-3.

```
<element xmlns:pref="http://www.foo.com" pref:attr="This is in foo.com namespace" />.
```

Example 3-3: Binding an attribute to a namespace

3.2.3 Redeclaring namespaces and undeclaring default namespaces

A namespace declaration propagates into the complete subtree, i.e. a namespace declaration is visible in all children and children of the children etc. of the element in which the namespace was declared. This means that namespaces do not have to be declared in all elements which utilize them, but the declaration is sufficient in an ancestor.

```
<a:root xmlns:a="http://www.a.com/">
   <b>
      <a:c />
   </b>
</a:root>
```

Example 3-4: Namespace ‘bleeding’

In example 3-4, the namespace prefix a in the `<a:root>` element is bound to `"http://www.a.com/"`. The `<a:c>` element can use this already declared prefix without refreshing the binding. So both the elements with the local name `root` and local name `c` are bound to `"http://www.a.com/"`.
A namespace prefix can be reused and be bound to another namespace by reassigning it to the new namespace.

```
<a:root xmlns:a="http://www.a.com/">
   <b>
      <a:c xmlns:a="http://www.b.com/" />
   </b>
</a:root>
```

Example 3-5: Namespace ‘bleeding’ 2

In example 3-5, the <c> element is in the `"http://www.b.com/"` namespace because the prefix a is bound to that URI.

Once a prefix is assigned to a particular namespace, there is no way to remove this binding; it can be reassigned to a new namespace, but in the given subtree, the prefix is used and cannot be free'ed. This rule does not apply to default namespaces.
In example 3-6, the `<root>` element declares a default namespace which means that the unprefixed `<root>` element is in the given namespace. The <b> element uses the `xmlns=""` statement which undeclares the default namespace—the <b> element is in no namespace, the default namespace is deleted. The <c> element declares the unused default namespace again.

```
<root xmlns="http://www.a.com/">
  <b xmlns="">
     <c xmlns="http://www.a.com/" />
  </b>
</root>
```

Example 3-6: Namespace 'bleeding' 3

3.2.4 Special namespaces

By definition, the prefixes xmlns and xml are bound to specific namespace without that they have to be declared explicitly:

- ❑ The prefix `xml` is bound to "`http://www.w3.org/XML/1998/namespace`"
- ❑ The prefix `xmlns` is bound to "`http://www.w3.org/2000/xmlns/`"
- ❑ Even the attribute `xmlns="http://foo"` is in the above namespace, although it does not use a prefix.

3.2.5 Relative URLs in namespaces

The W3C explicitly defined that namespaces must not have relative URLs as namespace URI. For instance, the namespace `xmlns="../1.dtd"` is not allowed as it is a relative URL. A document that contains namespace nodes with relative URLs as value cannot be canonicalized (see "Canonical XML" on page 47) and therefore not digitally signed.

3.2.6 Namespaces 1.1

The bleeding of namespaces into the subtree leads to problems during canonicalization (defined later): complicated processing models must be created to allow easy cut-and-paste movement of subtrees into different contexts. For this reason, the W3C started writing a new namespaces recommendation for XML 1.1 [BHLT02], which enables XML to undeclare prefixed namespaces. After XML 1.0 only allowed undeclaring the default namespace, XML 1.1 extends this concept to all namespaces.

3.3 XML InfoSet

In October 2001, the “XML Information Set” [CoTo01] was published to unify the way XML is described in further specifications:

> *“The XML Information Set provides a set of definitions for use in other specifications that need to refer to the information in an XML document.” [CoTo01]*

In the following paragraphs, the “XML Information Set” will shortly be called “infoset”. The infoset describes the “philosophy” behind the combination of XML 1.0 and namespaces. All well-formed XML documents that satisfy the namespace constraints defined by the namespaces recommendation [BHL99] have an infoset.
An infoset consists of a number of “information items”. Each document contains at least a “document information item” and an “element information item”. An information item is an abstract description of a specific part of the XML document. Each information item has a set of associated, named properties. The names of these properties are given in square brackets, `[propertyName]`. According to the infoset recommendation, the terms “information set” relates to the “tree” and the “information items” are the individual “nodes”. Table 3-1 on page 34 gives an overview to the existing information items and which properties exist in which information item.

Properties	**Infoset items**										
	Document	**Element**	**Attribute**	**Processing Instruction**	**Unexpanded Entity Reference**	**Character**	**Comment**	**Document Type Decl**	**Unparsed Entity**	**Notation**	**Namespace**
[all declarations processed]	X										
[attribute type]			X								
[attributes]		X									
[base URI]	X	X		X							
[character code]						X					
[character encoding scheme]	X										
[children]	X	X						X			
[content]				X			X				
[declaration base URI]					X				X	X	
[document element]	X										
[element content whitespace]						X					
[in-scope namespaces]		X									
[local name]		X	X								
[namespace attributes]		X									
[namespace name]		X	X								X
[name]					X				X	X	

Table 3-1: XML Information Set Properties

Properties	Infoset items										
	Document	Element	Attribute	Processing Instruction	Unexpanded Entity Reference	Character	Comment	Document Type Decl	Unparsed Entity	Notation	Namespace
[normalized value]			X								
[notation name]									X		
[notations]	X										
[notation]				X					X		
[owner element]			X								
[parent]		X		X	X	X	X	X			
[prefix]		X	X								X
[public identifier]					X			X	X	X	
[references]			X								
[specified]			X								
[standalone]	X										
[system identifier]					X			X	X	X	
[target]				X							
[unparsed entities]	X										
[version]	X										

Table 3-1: XML Information Set Properties

One problem of the information set is that the data model does not directly map to the XPath data model (“XPath” on page 36) and the DOM data model.

3.4 Document Object Model (DOM)

The “Document Object Model” (DOM) recommendation [LLN+02] describes the access to the tree of an XML document in a programming language neutral fashion. This access includes navigation through and retrieval of information (read access) and modification of the tree (write access). The DOM recommendation has different versions, called “levels”. DOM Level 1 is not any more important because of its lack for namespace support. DOM Level 2 extends Level 1 and is namespace aware.
In the JAVA programming language, all DOM access classes are grouped inside the `org.w3c.dom.*` package. The most common interfaces are:

- ❑ `org.w3c.dom.Document` represents the document information item
- ❑ `org.w3c.dom.Node` is the super class for all other information items
- ❑ `org.w3c.dom.Element` represents an element information item
- ❑ `org.w3c.dom.Attr` represents an attribute information item
- ❑ `org.w3c.dom.Text` represents a sequence of character information items (here, a divergence between the information set data model and the

DOM data model becomes visible: in the infoset, each character is a single information item while the DOM groups a contiguous charactes into a single text node.

- ❏ `org.w3c.dom.Comment` and `org.w3c.dom.ProcessingInstruction` represent the comment and the procession instruction information items.

3.5 XPath

The “XML Path Language” (XPath) [ClDe99] is used to identify and select parts of XML documents. XPath has a syntax which is not XML based (no elements etc.), so that the simple usage of XPath expressions inside attribute values or text nodes is possible.
For instance, XPath is used by the XSL Transformations language (XSLT) [Clark99] for selecting the nodes which are to be transformed by an XSLT style sheet. XSLT defines a language for describing how an XML tree structure is transformed into a result XML tree.
XPath allows the selection of nodes based on their local name, namespace names, node values, position relative to other nodes and various other selection mechanisms like simple arithmetic operations. Additionally, text and arithmetic operations can be performed. XPath Node types
The XPath data model knows only seven different types of nodes:

1. The root node (which maps to the document information item)
2. Element nodes
3. Attribute nodes
4. Namespace nodes
5. Comment nodes
6. Processing instruction nodes
7. Text nodes (a text node can consist of multiple character information items; if multiple Text nodes (e.g. an alternating sequence of ‘real’ text nodes and CDATA (character data) sections) are selected, only the first node in the sequence is in the selection)

3.5.1 XPath axes

The XPath recommendation defines 13 so called axes. For a given node (called context node), each axis identifies a node set relative to the context node. Before illustrating how these axes look like, the term *'document order'* must be introduced:

> *"There is an ordering, document order, defined on all the nodes in the document corresponding to the order in which the first character of the XML representation of each node occurs in the XML representation of the document after expansion of general entities. Thus, the root node will be the first node. Element nodes occur before their children. Thus, document order orders element nodes in order of the occurrence of their start-tag in the XML (after expansion of entities). [...]. Reverse document order is the reverse of document order." [LLN+02]*

The left tree in figure 3-4 is the structure on which the axes will be visualized in the next samples; the black node in the middle named 'I' is the context node which is used for selecting the axes.

3.5.1.1 SELF AXIS

The self axis (right tree in figure 3-4) contains just the context node itself [ClDe99].

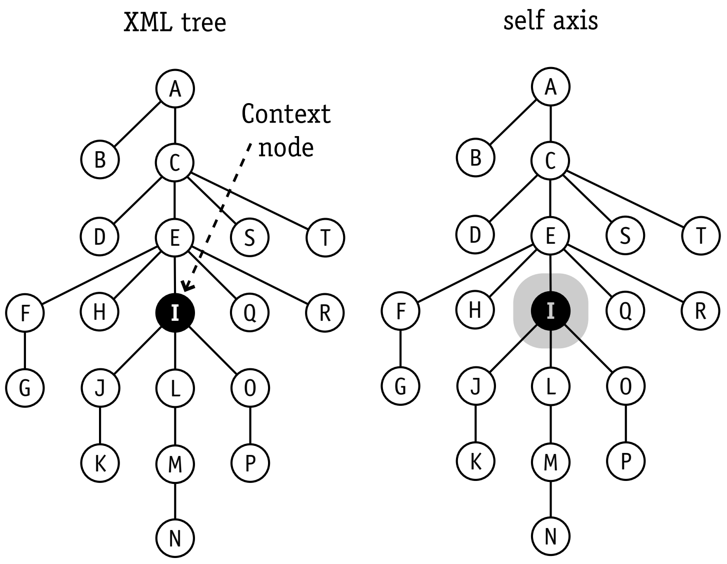


Figure 3-4: The sample document and the self axis

3.5.1.2 Parent axis

The parent axis (left tree in figure 3-5) contains the parent of the context node, if there is one [ClDe99].

3.5.1.3 Ancestor axis

The ancestor axis (middle tree in figure 3-5) contains the ancestors of the context node; the ancestors of the context node consist of the parent of context node and the parent's parent and so on; thus, the ancestor axis will always include the root node, unless the context node is the root node [ClDe99]. Seen relative to the document order, the start tag of an ancestor opens before the start tag of the context node and the end tag of the ancestor node closes after the end tag of the context node.

3.5.1.4 Ancestor-or-self axis

The ancestor-or-self axis (right tree in figure 3-5) contains the context node and the ancestors of the context node; thus, the ancestor axis will always include the root node, i.e. the document node [ClDe99]

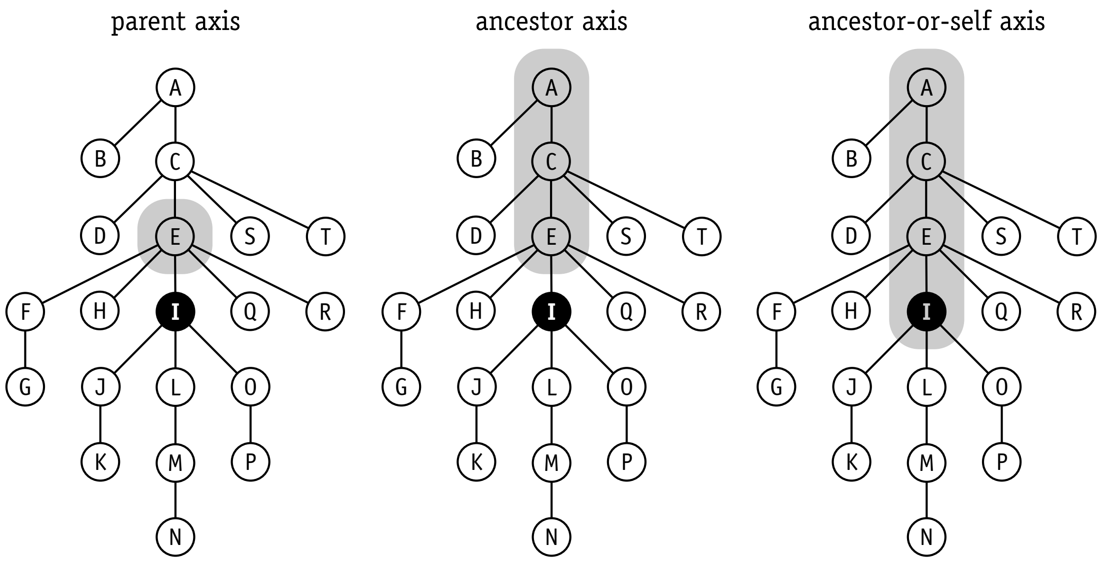


Figure 3-5: parent, ancestor and ancestor-or-self axes

3.5.1.5 Child axis

The child axis (left tree in figure 3-6) contains the children of the context node [ClDe99].
The child axis never contains attribute or namespace nodes (see “Attribute axis” on page 41 and “Namespace axis” on page 41).

3.5.1.6 Descendant axis

The descendant axis (mid tree in figure 3-6) contains the descendants of the context node; a descendant is a child or a child of a child and so on; thus, the descendant axis never contains attribute or namespace nodes [ClDe99].
Relative to the document order, the start tag of a descendant follows the start tag of the context node and the end tag of the descendant precedes the end tag of the context node.

3.5.1.7 Descendant-or-self axis

The descendant-or-self axis (right tree in figure 3-6) contains the context node and the descendants of the context node [ClDe99].

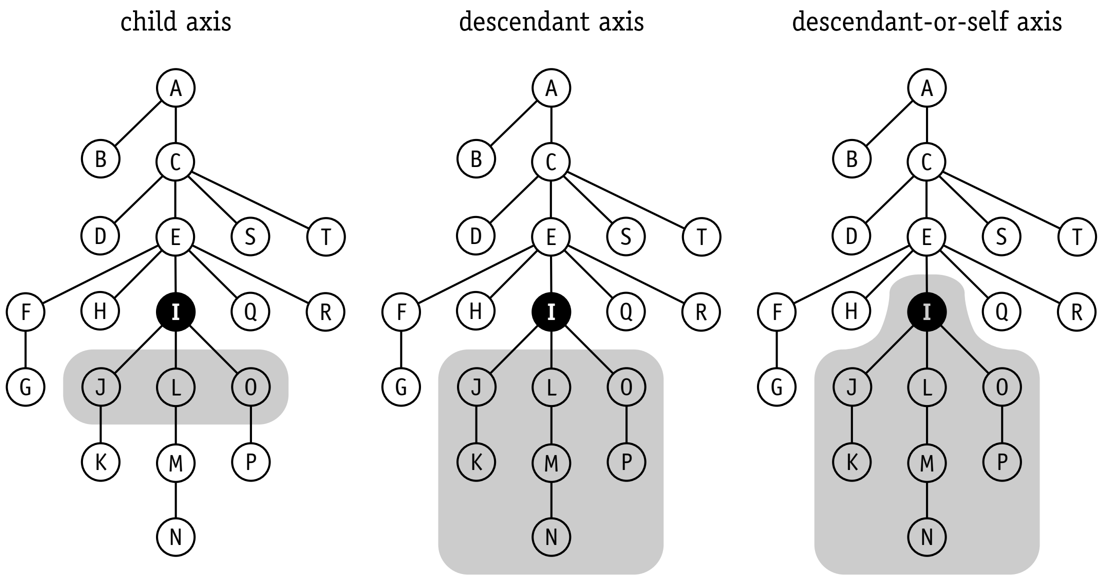


Figure 3-6: child, descendant and descendant-or-self axes

3.5.1.8 Preceding-sibling axis

The preceding-sibling axis (left tree in figure 3-7) contains all the preceding siblings of the context node; if the context node is an attribute node or namespace node, the preceding-sibling axis is empty [ClDe99].

3.5.1.9 Following-sibling axis

The following-sibling axis (right tree in figure 3-7) contains all the following siblings of the context node; if the context node is an attribute node or namespace node, the following-sibling axis is empty [ClDe99].

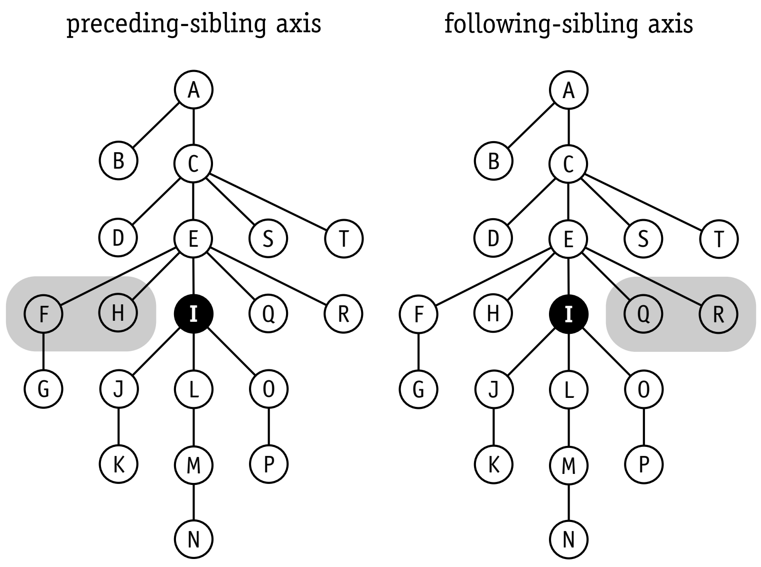


Figure 3-7: preceding-sibling and following-sibling axes

3.5.1.10 PRECEDING AXIS

The preceding axis (left tree in figure 3-8) contains all nodes in the same document as the context node that are before the context node in document order, excluding any ancestors and excluding attribute nodes and namespace nodes [ClDe99]. Relative to the document order, the end tag of a preceding node precedes the start tag of the context node; there is no parent/child or ancestor/descendant relationship between preceding and context node.

3.5.1.11 FOLLOWING AXIS

The following axis (right tree in figure 3-8) contains all nodes in the same document as the context node that are after the context node in document order, excluding any descendants and excluding attribute nodes and namespace nodes [ClDe99]. Relative to the document order, the start tag of a node in the following axis follows the end tag of the context node; there is no parent/child or ancestor/descendant relationship between preceding and context node.

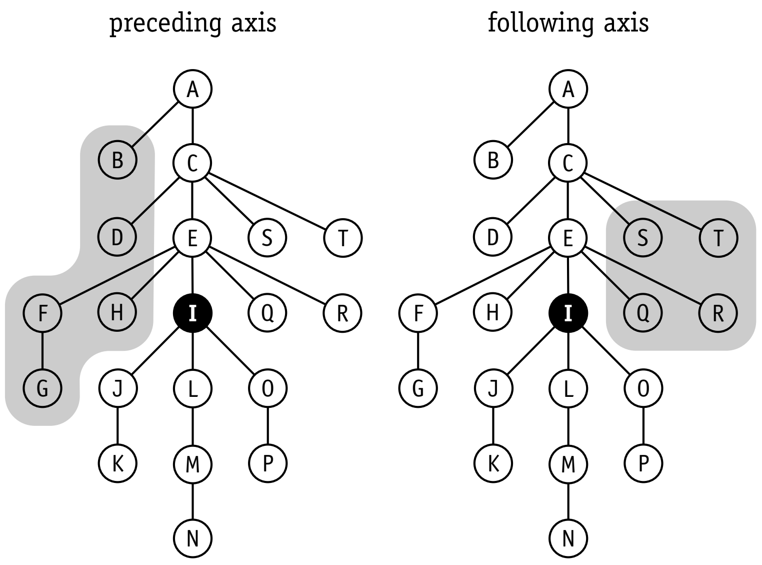


Figure 3-8: preceding and following axes

3.5.1.12 ATTRIBUTE AXIS

The ‘attribute axis’ cannot not be visualized by the example, because the attributes (as well as the namespaces) are directly bound to the element, i.e. their position in the tree is inside their owner element:
The attribute axis contains the attributes of the context node; the axis will be empty unless the context node is an element [ClDe99].

3.5.1.13 NAMESPACE AXIS

The ‘namespace axis’ contains the namespace nodes of the context node; the axis will be empty unless the context node is an element [ClDe99].

3.5.1.14 Partitioning of the document using axes

The five axes 'ancestor', 'preceeding', 'self', 'descendant' and 'following' partition the complete document (see figure 3-9).

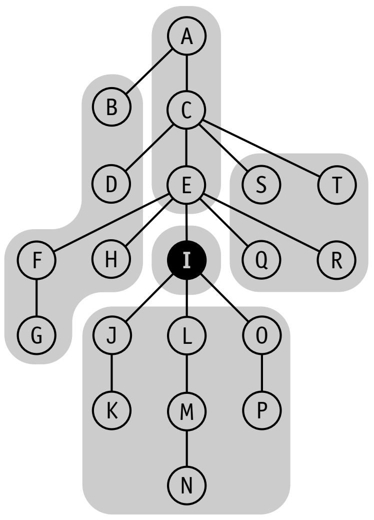


Figure 3-9: Partitioning using XPath axes

For an arbitrary context node, each Element-, Text-, Comment- or PI-node in the document is on one of the context node's axes. So the axes can be used to describe their relative position to each other. The details of a classification scheme can be found in table 7-3 on page 97.

3.5.2 XPath examples

The use of XPath for selecting nodes is shown using examples.

3.5.2.1 Example 1

Given an HTML document, the text value of the third chapter heading element can be selected using the XPath expression in example 3-7.

```
/html/body/h1[3]/text()
```

Example 3-7: Abbr. XPath expression to select a heading in an HTML doc.

The syntax in the example is the abbreviated syntax which does not explicitly specifies the axes. The '/' at the beginning of the expression selects the document node, i.e. the parent node of the document element node. The 'html' assumes that the document element is an <html> element. Otherwise, the node set would be empty. The next location step selects all <body> elements which are children of the <html> element. From all selected <body> elements, all <h1> elements are selected. From the resulting node set, the third node is selected. On this node, the 'text()' function is evaluated. This evaluation returns the concatenated string values of all Text descendants.

Using the full syntax (not abbreviated), the XPath would read like in example 3-8.

```
/child::html/child::body/child::h1[position()=3]/child::text()
```

Example 3-8: XPath expression to select a heading in an HTML doc

3.5.2.2 EXAMPLE 2

Given the HTML document in example 3-9. The paragraph <p> contains two <a> hyperlinks with a `href` attribute and corresponding link text.

```
<html><body><p>
<a href="http://www.xml-conference.org/" > The Markup conference   </a>
<a href="http://www.security-conf.org/"  > The Security conference </a>
</p></body></html>
```

Example 3-9: HTML document with two links

The XPath expression

`//a/@href[contains(../text(), "Markup")]/..`

selects the first of the two <a> elements, because the hyperlink text contains the character sequence 'Markup'.

- ❑ The '`//a`' part of the expression selects all <a> elements in the complete document and is the short form of '`/descendant-or-self::a`'.
- ❑ The '`/@href`' selects all `href` attributes in the <a> elements. The long form of this expression is '`/attribute::href`'.
- ❑ The predicate test '`[contains(../text(), “Markup”)]`' is evaluated against all `href` attributes.
    - ❍ The `contains()` function returns true if the string in the first argument contains the string from the second argument.
    - ❍ The first string is the result of the evaluation of '`../text()`'.
        - ✩ The expression '`..`' selects the parent of the context node. In this case, the context node is the `href` attribute. The parent of the `href` attribute is the <a> element.
        - ✩ The result of the `text()` function is the value of all text nodes.
    - ❍ For the first <a> element, the result of the `text()` function is "` The Markup conference   `". In that case, the `contains()` function returns true.
    - ❍ For the second <a> element, the evaluation of the `contains()` function returns false.

- ❑ After the predicate test, the result contains the `href` attribute of the first <a> element. In the concluding '`..`' step, the first <a> element is selected.

The abbreviated XPath expression

```
//a/@href[contains(../text(), "Markup")]/..
```

has the following long form:

```
/descendant-or-self::node()/child::a/attribute::href[ contains(
  parent::node()/child::text(), "Markup")]/parent::node
```

3.6 Differences between the DOM2 and XPath data model

The W3C works hard on making all recommendations consistent to each other, but sometimes, this goal is not fully reached. One example for this is the handling of namespaces:
The *Document Object Model Level 2* (DOM2) is namespace aware, i.e. all elements and attributes have a namespace property. Given the XML code in example 3-10.

```
<a:root xmlns:a="http://www.a.com/" attr="foo">
  <b>
     <a:c />
  </b>
</a:root>
```

Example 3-10: XML snippet on DOM2 namespace handling

If XML is parsed into a DOM2 structure, the <`a:root`> element has two attributes, an "`xmlns:a`" and an "`attr`" attribute. This means that both the namespace node and the attribute node can be found in the attributes of the element. The XPath model is slightly different: The attribute axis contains only one attribute, namely the "`attr`" attribute. The namespace axis contains a node for the "`xmlns:a`" namespace declaration.
Given the DOM2 representation of the <b> element in example 3-10, the element does not have any attributes. Again, the XPath model is different: The namespace axis contains a node for the "`xmlns:a`" namespace declaration which was made in the <`a:root`> ancestor element. So the XPath view of example 3-10 would be like example 3-11.

```
<a:root xmlns:a="http://www.a.com/" attr="foo">
  <b xmlns:a="http://www.a.com/">
     <a:c xmlns:a="http://www.a.com/"/>
  </b>
```

Example 3-11: XPath view from example 3-10

```
</a:root>
```

Example 3-11: XPath view from example 3-10

This leads to some problems when it comes to evaluating very strange XPath expressions on documents: *XML Signature* (discussed in the next chapter) is defined in terms of the XPath data model, and depending on what XPath processor is used for performing operations on a document, the results can differ. For instance, the Apache Xalan-J processor does not return correct results if selections on the namespace axis are made.
Another namespace related problem is the existence of `xmlns=""` attributes in DOM space versus the absence of the default namespace in the namespace axis in XPath space. Given the XML in example 3-12.

```
<foo xmlns="">
</foo>
```

Example 3-12: Default namespace undeclaration

In a DOM2 tree, the foo element has a single attribute in the "XML Namespaces" namespace with localname `xmlns` and no prefix and with an empty value. In the XPath data model, the foo element has an empty attribute axis and an empty namespace axis. So for the XPath data model, it is equal to example 3-13.

```
<foo>
</foo>
```

Example 3-13: No default namespace undeclaration

Using the snippet from example 3-13 in another surrounding context like in example 3-14, in the DOM2 representation, both bar and foo have a single attribute. In the XPath model, the namespace axis of `<bar>` has one namespace while the namespace axis of `<foo>` is empty.

```
<bar xmlns="http://bar.com/">
   <foo xmlns="">
   </foo>
</bar>
```

Example 3-14: foo element with empty namespace axis

4 Canonical XML and XML Signature

The "*Canonical XML Version 1.0 Recommendation*" [Boy01] describes a method for creating a unique physical representation of an XML instance which accounts for permissible changes. If two XML documents have the same canonical form, then these documents are logically equivalent in a given context. Canonicalization discards irrelevant details from an XML document and supplies a non-ambiguous octet representation. (Canonicalization is often simply called "c14n" because this long word consists of the character 'c', then 14 other characters and the trailing 'n' character.)
The *"XML Signature Syntax and Processing Recommendation"* [ERS02] document specifies a syntax and basic processing rules for XML Signatures:

> *"XML Signatures provide integrity, message authentication, and/or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere." [ERS02]*

The term 'message authentication' is defined as follows:

message authentication: *"The property, given an authentication code/protected checksum, that tampering with both the data and checksum, so as to introduce changes while seemingly preserving integrity, are still detected."* [ERS02]

Therefore, XML Signatures provide integrity and data origin authentication for arbitrary parts of the XML document in which the signature resides, arbitrary parts of external XML documents and arbitrary non-XML resources.

4.1 Canonical XML

XML defines a serialization format for a tree structure that has specific properties. XML documents which have the same infoset can differ in their physical representation. The term "physical representation" refers to an XML document which is serialized into a octet sequence. Here are some examples to illustrate these ambiguities:

- ❏ Character encoding: XML is able to be serialized using different [character encoding schemes] like Unicode UTF-8, Unicode UTF-16, ASCII or ISO-8859-1. C14n always uses the [character encoding scheme] UTF-8.
- ❏ Line breaks: Different systems (UNIX, Microsoft Windows, Apple Mac OS) use different line endings. All line endings are normalized (UNIX convention).
- ❏ Attribute values: The values of the attributes are to be normalized according to the XML 1.0 specification. All attribute values are delimited by double quotes.

- Attribute ordering: Attribute information items are attached to a given element, but they are not children of that element. The attributes are an unordered set of name/value pairs. Additionally, the element information item has the [namespace attributes] property. C14n defines the order in which serialization must be performed on [namespace attributes] and [attributes].
- Whitespace: attributes in the 'start tag' are separated using a single space. Multiple spaces are reduced to a single space. Whitespace outside the document element (direct whitespace 'children' of the document information item are not text information items, so this 'indentation whitespace' is normalized).
  Whitespace in 'real' text information items is not changed (but the line endings).
- CDATA sections: Text can be stored in Text nodes or in CDATA sections. CDATA sections allow to directly use characters which are normally reserved for markup. CDATA sections are converted into text nodes.
- Character and parsed entities are expanded: XML 1.0 allows defining abbreviations for user-defined information. Character entities are some sort of abbreviations like the HTML "`&Uuml;`" entity for the German umlaut 'Ü'. Parsed entities are like macros for complete sequences of XML code.
- Empty elements: An empty element which has no child nodes can be represented like `<e></e>` or `<e/>`. C14n selects the former representation (start-and-closing tag) for all empty elements.
- XML declaration and DTD removed: XML is often prefixed by the `<?xml version="1.0"?>` declaration. Additionally, a DTD or DTD subset can be included. Both are omitted from canonical XML.
- Default attributes are added: The DTD subset can define default attributes for particular elements. These attributes must be added to the respective elements.
- Special characters: Special characters in text information items and attribute information item values are replaced by their character references.
- Namespace declarations: Superfluous namespace declarations are removed. If a namespace is already visible by the [in-scope namespaces] property, but is redefined using the same value, this declaration is redundant and will be removed.

The "*Canonical XML Version 1.0 Recommendation*" defines two different algorithms:

- ❏ *Canonical XML with comments* (including comments)
- ❏ *Canonical XML* without comments, also called "Canonical XML omitting comments)

Canonical XML is defined using the XPath data model and allows XPath node-sets as input. The most complicated part of the canonicalization process is to decide which namespaces are in scope and in which start tag they are to be declared during canonicalization (superfluous namespace declarations have to be removed).

4.1.1 Document subsets

Canonical XML allows the canonicalization of XML documents, XML document subsets or octet sequences which form an XML document. To illustrate the canonicalization of a document subset, the left illustration in figure 4-1 shows an input XML tree. The middle illustration shows which nodes are selected for inclusion in the document subset. The right illustration shows an XML document which corresponds to the canonicalized document subset. .

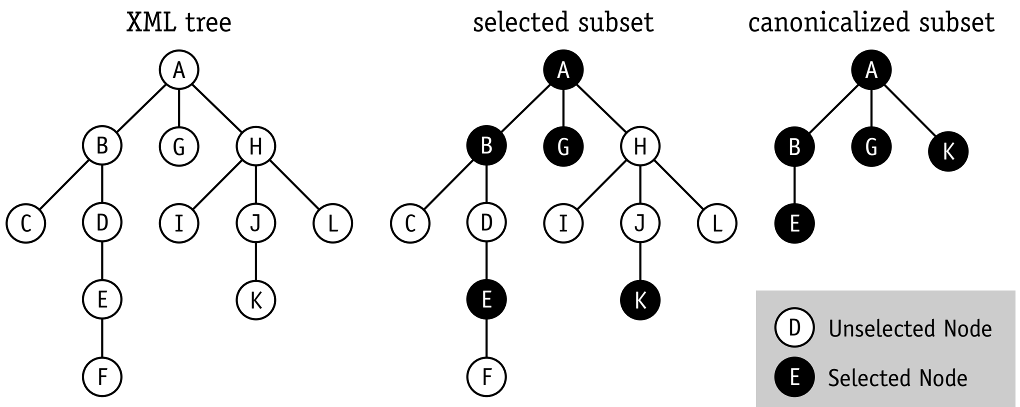


Figure 4-1: Canonicalizing a document subset

One possible XPath expression to select this subset could be

```
(/A | //B | //E | /A/G | /A/H/J/K)
```

Nodes become direct children of their first visible ancestor when their parent node has not been canonicalized; this is shown in the right tree in figure 4-1: The E node becomes a child of B and the K node becomes a child of A.
Figure 4-2 on page 50 illustrates how namespace declarations are inherited from ancestors, if the ancestors are not part of the selected document subset:

1. Element A binds the default namespace and the prefix `w3c`. The [in-scope namespaces] of the infoset is as follows:
   1. The default namespace is bound to `"http://www.ietf.org"`
   2. Prefix `"w3c"` is bound to `"http://www.w3.org"`
2. Element `B` does not change the [in-scope namespaces]. There is no ancestor in the document subset who defines the both namespaces, so they must be declared by element B. The [in-scope namespaces] are as follows:
   1. The default namespace is bound to `"http://www.ietf.org"`
   2. Prefix `"w3c"` is bound to `"http://www.w3.org"`
3. Element `C` removes the binding of the default namespace.
   1. The default namespace is not assigned.
   2. Prefix `"w3c"` is bound to `"http://www.w3.org"`
4. Element `D` binds the prefix blah.
   1. The default namespace is not assigned.
   2. Prefix `"w3c"` is bound to `"http://www.w3.org"`
   3. Prefix `"blah"` is bound to `"http://www.blah.org"`

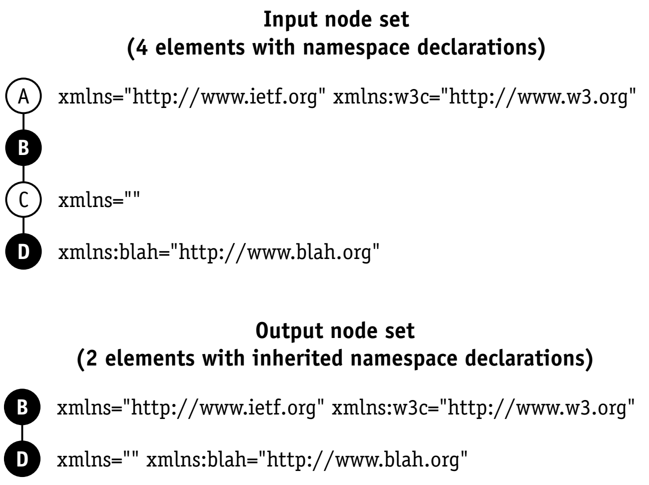

Input node set
(4 elements with namespace declarations)
A
xmlns="http://www.ietf.org" xmlns:w3c="http://www.w3.org"
B
C
xmlns=""
D
xmlns:blah="http://www.blah.org"
Output node set
(2 elements with inherited namespace declarations)
B
xmlns="http://www.ietf.org" xmlns:w3c="http://www.w3.org"
D
xmlns="" xmlns:blah="http://www.blah.org"

Figure 4-2: Namespace inheritance in document subsets

Note: The node-set is treated as a set of nodes, not a list of subtrees. If the input to be canonicalized is a node-set, the node-set must contain all nodes/information items which are to be serialized. In the DOM, selecting an element is analogous to select the element, all attributes and namespace declarations and all its descendants, or in short, the complete sub tree. In Canonical XML, the selection of this element means that only the element node without any attributes, namespace declarations or child nodes is canonicalized.

4.1.2 Applications of Canonical XML

4.1.2.1 XML Signature

Canonical XML is used by XML Signature to create a unique representation of an XML document or a subset hereof. This unique representation is necessary to compute a cryptographic digest value which is to be signed.
Canonical XML is very important if only a part of an XML document is to be signed. Many applications which process XML depend on the [in-scope namespaces] property of an element. This means that it is important which namespaces have been declared in the parent and ancestor element of the current node. The [prefix] of an element or attribute name does not carry semantic information in itself, it only binds the [local name] of the element/attribute to the [namespace name]. Depending on which [namespace name] is currently bound to the [prefix] in question, applications handle the information item differently. So, if an attacker could change that binding without breaking a signature, there would be a way to circumvent the trust model of the system.

4.1.2.2 XML Encryption

Canonical XML is used as a mechanism for the consistent serialization of XML into an octet stream prior XML encryption.
XML Encryption as currently defined by the W3C is a mechanism where an element (a complete subtree; similar to the XPath expression `descendant-or-self()`) or the element content (all children of an element; similar to the XPath expression `descendant()`). Plaintext (unencrypted) nodes are located in an XML document and 'live' in a specific context. This context is formed of their position, eventually the XML fragment uses [unexpanded entity references] and additionally, there possibly exist [in-scope namespaces]. During the encryption process, the XML becomes opaque ciphertext (an `<xenc:EncryptedData>` element). If the ciphertext replaces the plaintext, remains in place, is at a later time decrypted and if the resulting plaintext replaces the ciphertext, the plaintext after decryption is in the same context as before encryption. If the ciphertext is placed at a new location or the decrypted plaintext is placed somewhere else, the original context is not preserved. For this reason, the plaintext should include its full context prior encryption to allow cut-and-paste applications without loss of relevant information.

4.1.2.3 Comparison of XML documents or fragments

Canonical XML can be used to compare two XML documents and to decide whether they are logically equivalent. This would make sense in environments where UNIX-like 'diff' commands shall be applied to XML.

4.2 XML Signature

The "*XML Signature Syntax and Processing*" recommendation [ERS02] is a relatively young internet standard by the W3C which defines a syntax and processing model for a special format for digital signatures. These signatures are represented in an XML format and can sign arbitrary resources, including XML and parts thereof.

4.2.1 Introduction

The structure and processing of XML Signatures introduces some interesting concepts which will be explained briefly: First, a single XML Signature can cover (sign) *multiple* resources/messages. It is possible to sign an XML document, a web resource and a binary image on the hard drive using a single XML signature.
Conventional signature systems, which can only sign a single resource, sign the hash value of the signed resource. To enable XML Signature to sign multiple resources, the signing process consists of two distinct steps: The hash values of the signed resources and additional attributes (their URI) are collected in an octet string. This octet string is digested again, and the resulting hash value is then signed using a signature method like e.g. "RSA/SHA-1". (There can exist multiple independent digest functions; the first one for resources and the second one implicitly in the signature algorithm. This often causes confusion.)

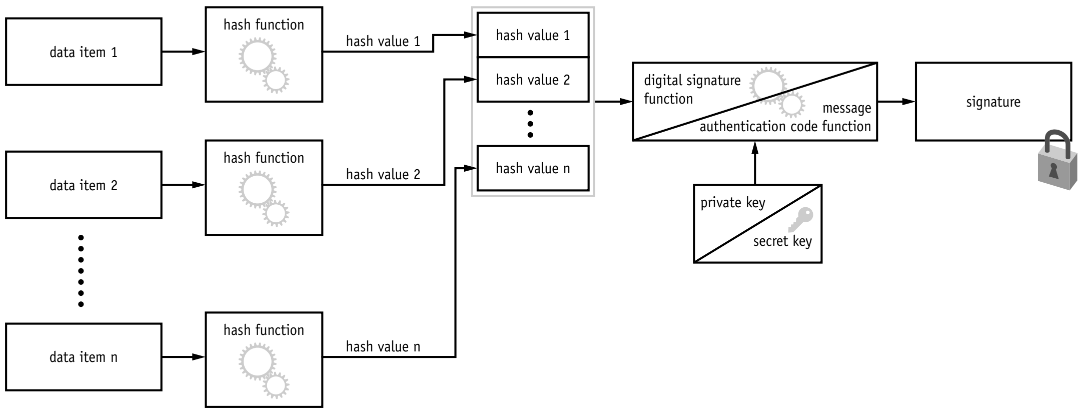


Figure 4-3: Hash value generation for XML Signature (simplified)

The hash value/URI collection and the signature value form the basic signature structure. Additionally to these basic building blocks, there exist structures to embed key management information like key names or certificates and arbitrary objects (XML itself or base64 encoded binary data) into the signature.
Each entry in the hash value/URI collection can contain an additional sequence of transforms which are to be applied prior digest value calculation. This enables XML Signature to apply some pre-processing operations to a referenced resource; the output of the transforms is the content to be signed. Transforms can include transforms like selecting specific parts of an XML doc-

ument or Base64 decoding or even user-defined transforms for a specific application domain.
The XML Signature recommendation supports digital signatures using algorithms like RSA or DSA which support data origin authentication and it supports symmetric MAC algorithms, which do *not* provide data origin authentication, so the term 'signature' and 'signing' is not always correctly used in the sense of 'data origin authentication'. Nevertheless, in the following chapter, "signing" and "verifying" refer to both creating/validating a signature using a private key or a MAC using a secret key.

4.2.2 Enveloping, enveloped and detached signatures

Old-fashioned signature systems like PGP or S/MIME produce two different forms of signatures:

1. *Enveloping signatures* wrap the signed contents; they form an envelope around the signed contents. The signed contents become part of the signature.
2. *Detached signatures* are objects which are separated from the signed contents.

These forms have different properties and limitations:

4.2.2.1 Enveloping signatures

Enveloping signatures have the advantage that only one data object exists: The signature and the signed content form a single entity which can be handled easily during transport—there is no problem to miss the signature or the contents, it's always together. But this strength is also the problem: Before an application can handle and process the signed contents, the signature application must strip away the signature-envelope; e.g. if the signer signs a Microsoft Word document using PGP and does not create a detached signature (which implicitly means that an enveloping signature is created), the verifier must first validate the signature before he get's access to the signed Word document. Only the signature application knows how to "unpack" the signed contents. The packaging problem is not specifically related to a particular signature application, but it's inherent if binary data is wrapped by an enveloping signature.
If the enveloped data is an XML instance, it's very easy for an XML processing application to access this data because it's accessible inside the XML document without the necessity to use an XML Signature tool.

4.2.2.2 Detached signatures

Detached signatures have the advantage that the signed contents are not merged into the signature so that they stay where they are without being modified in any way. The association between signature and signed contents is usually done using a simple mechanism like the file name (PGP) or by other logical bindings like the ordering of a e-mail-attachments (MIME expresses which message parts are signed by a S/MIME signature). These bindings are a

little bit weak: The signed contents can be unintentionally separated from the signature by copying or forwarding only the contents while forgetting the signature. The vice-versa situation is also possible: Forward only the signature and forgetting the signed contents. If the file containing the signature is renamed, the link is also broken. This makes the handling of detached signatures very prone to errors and user mistakes. The weak binding between signature and signed contents makes it even more complicated because if a recipient requests the missing signed contents from the signer, the filename itself is often insufficient information to find the corresponding contents.
XML Signature allows the creation of both enveloping and detached signatures. XML Signatures use URIs to identify the signed contents. This makes it possible to select files in a directory, on a web server and in any entity which is addressable via URI mechanisms. The creator of an XML Signature (the signer) is free on how he uses URIs, whether he uses absolute or relative URLs or even URNs for naming non-network resources. URIs add a strong binding mechanism to the detached signature mechanism. If the signer sends only the signature without signed contents, the verifier implicitly 'knows' how to retrieve the signed contents. Enveloping signatures benefit from XML Signature through the possibility to include multiple signed objects.

4.2.2.3 Enveloped signatures

Additionally to enveloping and detached signatures, XML Signature introduces a new type of signature:
Enveloped signatures are signatures which are placed *inside* the signed contents, they are enveloped by the signed contents. Enveloped signatures can only sign XML documents because they must become part of that document. Of course, this signed XML document can contain base64 encoded binary data.
Placing the signature into the document brings digital signatures very close to the way on how people use handwritten signatures today: The signature becomes part of the signed document, like the ink on the paper of a contract. The incorporation of a signature into an XML document changes the data structure of the document, so that the signed contents are changed by the signature itself. For that reason, XML Signature has a mechanism (Transforms) to select which portions of a document have been signed.

4.2.2.4 Comparison

In contrast to conventional signature systems, XML Signature enables the user to sign multiple resources within a single signature. Additionally, an XML document can contain multiple signatures.
An XML Signature can be enveloped in a signed resource, while enveloping a second resource and signing a third resource outside to the document. The terms 'enveloping', 'enveloped' and 'detached' for XML Signatures refer to the relationship between signed contents and signature:

- An enveloping signature is an ancestor of the signed contents in the XML tree.
- An enveloped signature is a descendant relative to (parts of) the signed contents in the XML tree. (Note: In most cases, the signature is not a descendant relative to *all* signed contents but to some of them; otherwise, an enveloped signature could sign only nodes which are on the XPath `ancestor()` axis. For illustrating this, see figure 4-4 on page 55.)
- A detached signature has no parent/child relationship to the signed contents. This is the case for two situations:
  - The signature and the signed contents reside in separate entities, e.g. in two different files or
  - the signature and the signed contents reside in the same XML document but have no parent/child relationship, e.g. both are siblings. The signed contents are outside of the signature element.

Figure 4-4 illustrates the different types of XML Signatures.

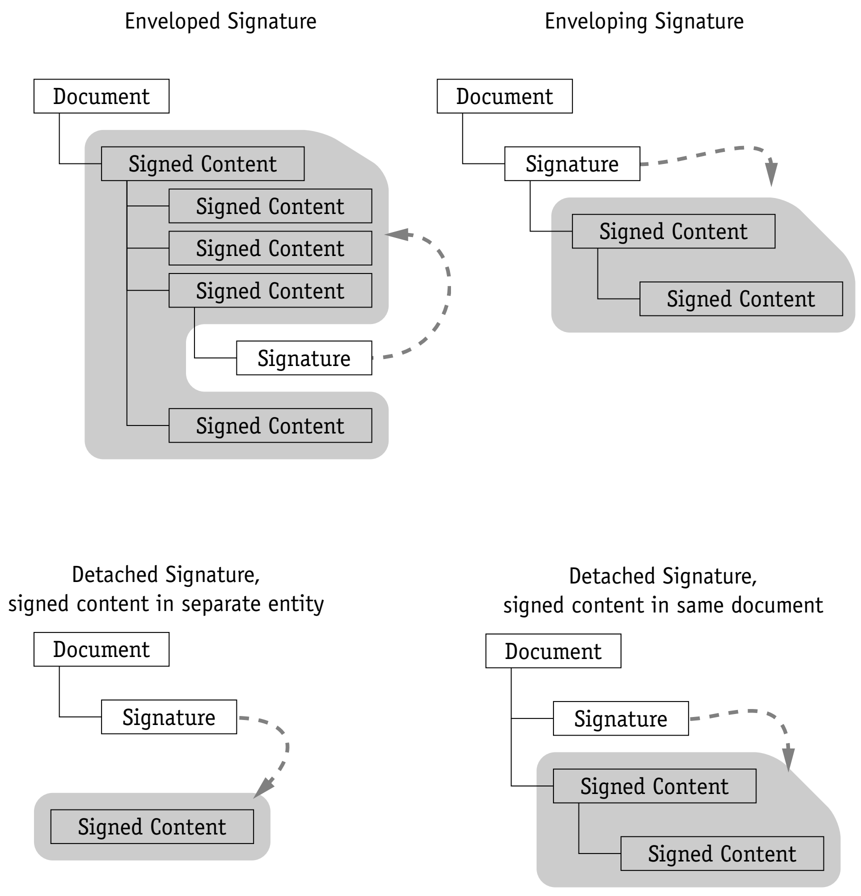


Figure 4-4: Enveloped, enveloping and detached signatures

Table 4-1 summarizes the differences between the two existing signature forms and the three XML based signature types:

Signature Type	**Enveloping signature**	**Detached signature**	**Enveloping XML Signature**	**Detached XML Signature**	**Enveloped XML Signature**
Nr. of existing objects	1	2	1	2	1
Transport handling	easy and robust	complicated and fragile	easy and robust	depends on usage	easy and robust
Signed contents handling	complicated; Signing application required for unwrapping	easy; signed content is directly accessible	depends on usage; binary contents are base64 encoded; signed XML contents are accessible	easy; signed content is directly accessible	easy; signed content is directly accessible
Signed contents constraints	Everything can be signed	Everything can be signed	Everything can be signed	Everything can be signed	Only XML can be signed
Binding to signed contents	fixed	simple mechanism like filename	URI plus Transforms	URI plus Transforms	URI plus Transforms
Number of signed entities	only one object can be signed	only one object can be signed	an arbitrary number of objects can be signed	an arbitrary number of objects can be signed	an arbitrary number of objects can be signed
Signature types can be combined	no	no	yes	yes	yes

Table 4-1: Comparison of signature types

4.2.3 References

4.2.3.1 Basics

Like mentioned earlier, XML Signature allows signing multiple resources within a single signature. For example, a single XML Signature can sign both an XML document and a file of a binary PNG image. To enable such functionality, the XML Signature recommendation introduces a construct called "Reference". A Reference is created using the `<Reference>` element. A Reference contains a pointer (`URI` attribute) to the signed contents and the hash value of these signed contents. The hash value is base64-encoded and stored in the `<DigestValue>` element. To indicate which hash function has been used, each Reference must also contain the `<DigestMethod>` element which contains the algorithm identifier of the hash function as an `Algorithm` attribute. A simple example of a `<Reference>` is in example 4-1.

```
<ds:Reference URI="http://www.w3.org/TR/xml-stylesheet">
   <ds:DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1" />
   <ds:DigestValue>60NvZvtdTB+7UnlLp/H24p7h4bs=</ds:DigestValue>
</ds:Reference>
```

Example 4-1: <Reference> element

The processing chain for a single `<Reference>` are as follows:

- ❏ De-referencing the `URI`,
- ❏ optionally transforming the contents (if `<Transforms>` are given),
- ❏ optional conversion to octets (only if the result of the previous step is a node set, canonical XML is applied) and
- ❏ applying the given `<DigestMethod>` to the octets to get the hash.

4.2.3.2 De-referencing URI attributes

The signed resource has the URI `"http://www.w3.org/TR/xml-stylesheet"`, the used hash function is SHA-1 and the base64-encoded hash value is in the `<DigestValue>` element. To verify this reference, the resource is fetched from the web (or from another location like a proxy or local cache) and the de-referenced octets are directly used as input to the hash function.
The de-referenced contents which are identified by the `URI` attribute are either octet streams or XPath node-sets. The data type of the de-referenced contents depends on the type of the `URI`. URIs which point to resources outside of the current document *always* return octet streams. This includes network resources or files in the local filesystem. An octet stream is even returned for resources which are external XML instances. URIs which point to parts of the same document in which the signature resides are called "*same-document*" references; de-referencing a same-document URI results in an XPath node-set.

This implies a duality of de-referenced contents: it is determined at runtime whether the de-referenced contents are represented as an octet stream or an XPath node-set, so the signature software must handle this duality.
The XML Signature recommendation defines the behavior for some very common types of URI:

❑ `URI="http://example.com/1.gif"`
The resource is an image file in GIF format. The file is fetched from the site `example.com` using the HTTP protocol. This reference URI returns an octet stream.

❑ `URI="http://example.com/bar.xml"`
The resource (file) `bar.xml` is fetched from the site `example.com` using the HTTP protocol. This reference URI returns an octet stream, regardless that the resource itself is (probably) an XML instance.

❑ `URI="http://example.com/bar.xml#chapter1"`
Identifies the element with ID attribute value '`chapter1`' of the external XML resource '`http://example.com/bar.xml`', again provided as an octet stream. Again, for the sake of interoperability, the element identified as ‘`chapter1`’ should be obtained using an XPath transform rather than a URI fragment.

❑ `URI=""`
This same-document URI returns an XPath node-set which contains all nodes from the XML document in which the signature itself reside, *without* any comment nodes.
A very common mistake by XML Signature newbies is to assume that comment nodes are available after `URI=""` is de-referenced.

❑ `URI="#xpointer(/)"`
This same-document URI returns an XPath node-set which contains *all* nodes from the XML document in which the signature itself reside, *including all comment nodes*.

❑ `URI="#paragraph1"`
This same-document URI returns an XPath node-set which contains *all* nodes which have the element with an ID attribute of value `"paragraph1"` on the `ancestor-or-self` axis, *without any comment nodes*. Simply speaking, the subtree (minus the comments) rooted by the element with a `"paragraph1"` ID.

❑ `URI="#xpointer(id('paragraph1'))"`
This same-document URI returns an XPath node-set which contains *all* nodes which have the element with an ID attribute of value `"paragraph1"` on the `ancestor-or-self` axis, *including the comment nodes*. Simply speaking, the subtree rooted by the element with a `"paragraph1"` ID.

The above URI-References can be divided into two classes: *'regular'* and *'same-document'* URIs. A URI points to the signed resource. A reference pointing to some external file is simply a URI. A subclass of URIs are 'same-document' URIs: A same-document URI references to a node set in the same document in which the Signature resides. The URIs `""`, `"#foo"`, `"#xpointer(/)"` and `"#xpointer(id('foo')"` reference nodes in the same document.

4.2.3.3 TRANSFORMATION OF RESOURCES USING TRANSFORM ELEMENTS

The XML Signature recommendation introduces a new mechanism for transforming resources: the optional `<Transfoms>` element. Without transforms, the de-referenced contents are directly used as input for the hash function to calculate the cryptographic hash value. Transforms enable the signature to extract the relevant information out of the de-referenced data or to unify (e.g. canonicalize) the de-referenced data.

The `<Transfoms>` element contains one or more `<Transfom>` elements which describe a single transform. Like mentioned earlier, these transforms are applied subsequently to the de-referenced contents. The output of the last transform is converted to an octet sequence which is used as input to the hash function.

Conversion between octet streams and XPath node-sets: De-referencing the URI can result in either octet streams or in XPath node-sets; hash functions can only calculate the hash value of octet streams. Some transforms need octet streams as input, some need XPath node-sets, and others can handle both. The XML Signature recommendation defines how to convert octet streams into XPath node-sets and vice versa:

❏ Octet streams are converted into XPath node-sets by parsing the octet stream.

❏ XPath node-sets are converted into octet streams by applying Canonical XML (without comments).

The Canonical XML recommendation only defined "Canonical XML (omitting comments)" as mandatory, so the XML Signature recommendation follows that path.

The concept of transforms is a powerful mechanism which enables arbitrary transformations on the input data; all transforms have to goal to filter the input data so that the important, to be signed information, is extracted and to make the data unambiguous. The XML Signature recommendation introduces five different transforms:

❏ *Canonicalization with and without comments*: The input is treated as XML and the canonical form is computed.

❏ *Base64 decoding*: The input is treated as a text which consists of base64 encoded octets. The output is the decoded octet stream.

❏ *XPath filtering*: The XPath filter takes an XPath node-set as input and evaluates an XPath supplied in the transform to each node in the input node-set. The result of the XPath evaluation is converted to a Boolean value. If it evaluates to true, the particular node is included in the result node-set.

  - ❍ The XPath transform has a high computational overhead: The XPath must be executed on *every* node in the input node-set. If the input to the transform consists of all nodes from a large document, this is a very costly process.
  - ❍ The XPath transform uses the XPath which is to be used is stored inside a `<XPath>` element which is a child of the `<Transform>` element.

- ❑ *Enveloped signature*: *"The enveloped signature transform T removes the whole `<Signature>` element containing T from the digest calculation of the Reference element containing T."* [ERS02]
  This transform is the foundation for enveloped signatures: a signature can "remove" itself from hash value input.
- ❑ *XSLT (eXtensible Stylesheet Language Transforms)*: the XSLT [Clark99] transform allows transforming an XML tree into another tree, based on transform instructions which are stored inside a stylesheet. This stylesheet is included as a child of the `<Transform>` element.
  The XSLT transform enables XML Signature to perform sophisticated tree operations: the vision of WYSIWYS ("What you see is what you sign") becomes realistic.
  The XML recommendation does not define a user-representation for arbitrary XML structures. XML by itself is only structured data without any display and rendering capabilities. XSLT can be used to transform XML into a representation which is displayed to a user, e.g. XML can be rendered into XHTML for web browsers or into WML ("Wireless Markup Language") for mobile devices like mobile phones or PDAs. Given the XSLT transform, the user does not sign the initial XML structure or parts thereof but the result of the XSLT transform which is displayed to the user.
  Nevertheless, this displayed view is computed just-in-time during signature generation/validation. This means that the application can process the original tree structure. The rendering into a human representation is only required for generation and validation of signatures.

The transforms architecture was designed with extensibility in mind; every `<Transfom>` element can contain an arbitrary XML structure as its children. The transform itself is identified by its algorithm identifier (the `ds:Transform/@URI` attribute). These both properties (arbitrary transforms through the algorithm identifier and arbitrary execution content through the children of `<Transform>`) allow the users to deploy custom transform algorithms if the existing set does not fulfill their needs.

An example for the concept of Transforms could be a canonicalization algorithm for binary images. A user who looks at a displayed image in a web browser does not care about the images encoding. It makes no difference

whether the image is in GIF, PNG, TIFF or JPEG format (given that the compression is not too bad that he sees no artifacts and that the resolution (size) of the images is nearly the same). The only necessary requirement is that the visualization of the image remain the same, regardless which encoding was used. Different encodings of the same image result in completely different hash values.
A canonicalization transform for images would allow signing the *contents* of an image, and allowing to transcode the image into different representations (formats) without breaking already existing signatures. Such a transform must be designed carefully so that it does not discard security-relevant image information, but robust enough to allow small, invisible modifications and transcodings.

4.2.4 SignedInfo element

A Reference contains a pointer to the signed information (URI). Optionally processing steps (Transforms) in the Reference can be used to extract and unify the relevant information. Additionally, the Reference contains the hash function used for calculation of the hash value and the literal hash value of the relevant information itself. This section introduces a container for references.
The `<SignedInfo>` element is the hash value/URI collection mentioned earlier in "Data integrity algorithms" on page 20. A `<SignedInfo>` contains one or more `<Reference>` elements which bind the signed resources to the hash values. For each signed resource, a `<Reference>` is included in the `<SignedInfo>`. Each signature must have exactly one `<SignedInfo>` element to indicate what is signed by the signature (see figure 4-5 on page 62).
The `<SignedInfo>` itself is not the signature, but an intermediary list of hash values (more exactly `<Reference>`s).
Generating the `<SignedInfo>` does not require the usage of a private or secret key, as only hash values are generated. Nevertheless, great care must be taken that no sensitive material is exposed as part of the `<SignedInfo>` element: careless use of URLs in the `URI` attribute can expose confidential information, e.g. passwords for HTTP realms or other credentials.
The SignedInfo is the final object what is being signed by the cryptographic signature or MAC algorithm. For that purpose, the SignedInfo is canonicalized and the resulting octets are processed by the cryptographic signature or (H)MAC algorithm.
The algorithms used for canonicalization and signature/MAC computation are also included in the `<SignedInfo>` element using the `<CanonicalizationMethod>` element and the `<SignatureMethod>` element. Including this information inside the SignedInfo ensures that these values are covered by the signature value and prevents sustitution of these algorithms by weaker ones: For instance, a hostile canonicalization algorithm could rewrite `<Reference>` structures so that a signature is always true.

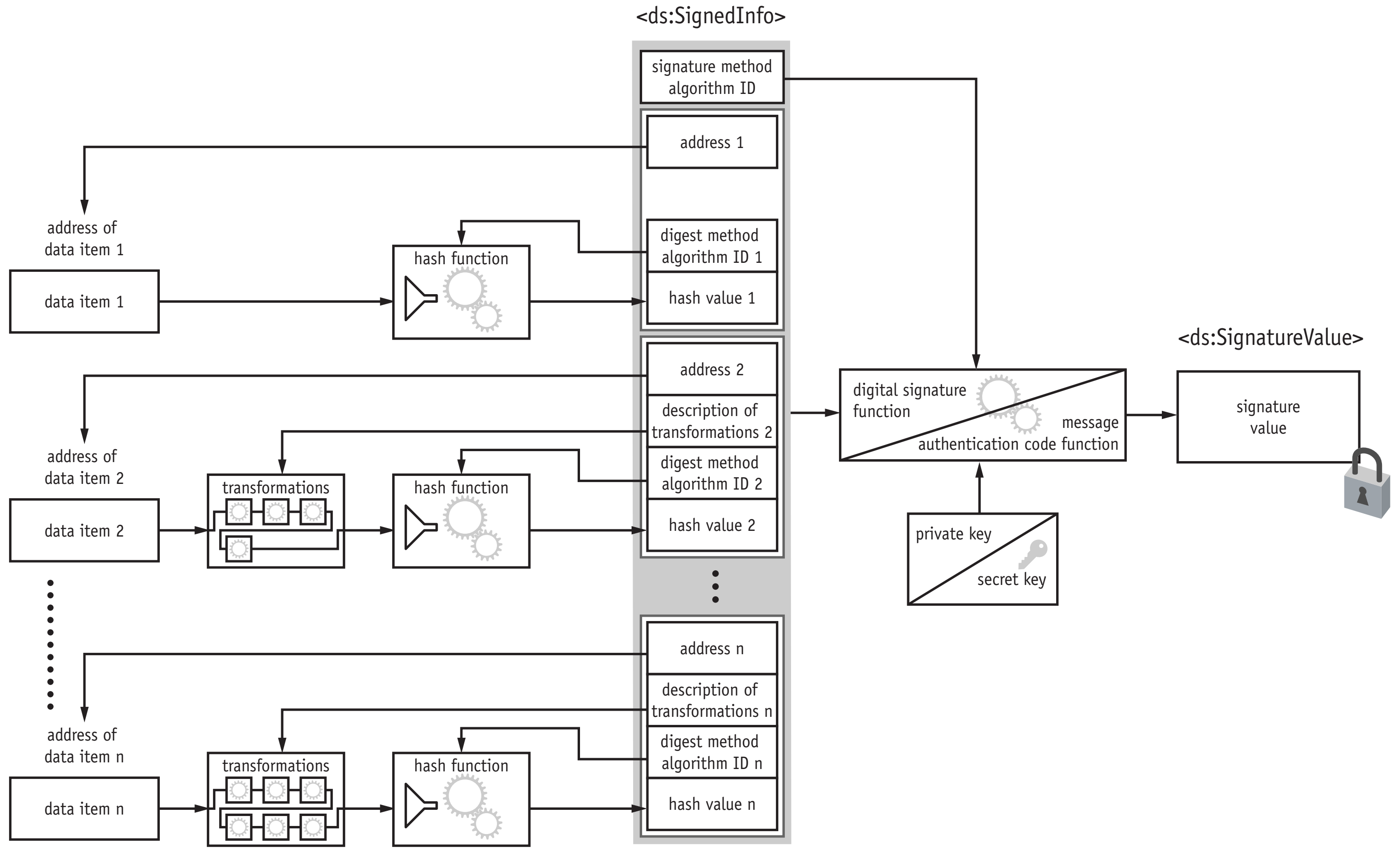


Figure 4-5: Signature value generation for XML Signature

4.2.4.1 SignatureValue element

After all signed contents are referenced using the SignedInfo, the SignedInfo is canonicalized using the c14n algorithm from the `<CanonicalizationMethod>` element and the resulting octets are signed using the signature or MAC algorithm from the `<SignatureMethod>` element. The resulting signature value is base64 encoded and stored inside the `<SignatureValue>` element which is a direct child of the `<Signature>` element.

4.2.4.2 Complex transforms vs. multiple references

It was mentioned earlier that for each signed resource, a `<Reference>` is included in the `<SignedInfo>`. This allows signing external resources by using URIs which are not same-document URIs ('same-document URIs' are explained on page 59). Multiple parts of the same XML document are to be signed using same-document URIs. It depends on semantics of the signed contents and on the application, whether all these parts are signed using a single `<Reference>` or multiple ones.
To illustrate this (figure 4-6 on page 64), consider a document with multiple enveloped signatures. The document is segmented into different content paragraphs which are signed in different constellations; e.g. `signature_1` signs `paragraph_2` and `paragraph_1` while `signature_2` signs `paragraph_3` and `paragraph_1`.
This gives the developer two options on how to construct the references:

❏ the first one is to sign both paragraphs using a single reference; this requires a complicated XPath transform to selects the right nodes out of the document; or

❏ to sign both paragraphs using two separate references without need for complicated selection transforms. In this second form, the both references which sign `paragraph_1` look completely the same if they use the same hash function. The second approach can make clearer what has been signed, because the individual `<Reference>`s are simpler.

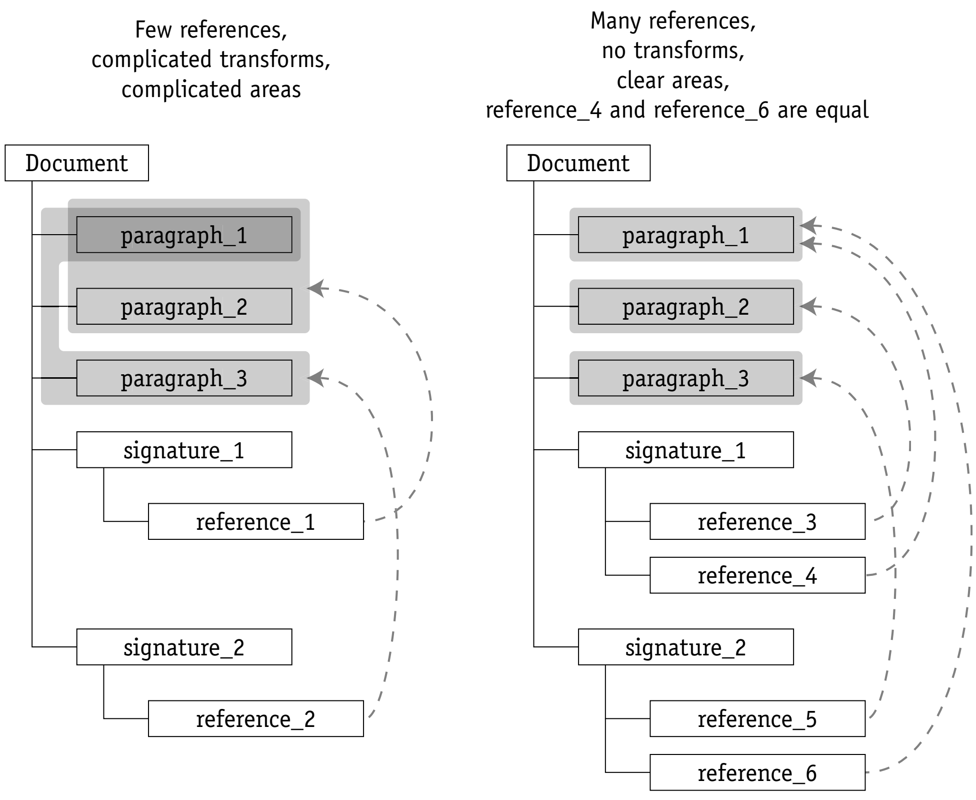


Figure 4-6: Reference design

The XPath transform defined in [ERS02] has a high computational overhead (as discussed on page 59). Therefore, the XML Signature working group created an additional transform called "*XML-Signature XPath Filter 2.0*" [BHR02]. XFilter2 specifies a second XPath transform which allows easier selection of portions of the XML document using union/intersect/subtract mechanisms.

4.2.5 Key Management using the KeyInfo element

XML Signature does not enforce usage of a particular signature scheme (although the implementation of some algorithms like DSA or HMAC-SHA1 is mandatory). The same freedom of choice is offered for the key management: XML Signature provides a generic container for key management information: the `<KeyInfo>` element. This container allows transportation of both identification information for keys and certificates and the values of the keys and certificates itself. This key material is needed to validate the signature. Different types of key information can be embedded into the signature. This is done by adding child elements to the `<KeyInfo>` element. Multiple items in the `<KeyInfo>` must all refer to the same key. The following list gives an overview of the specified mechanisms:

- The `<KeyName>` element contains a simple String which identifies a particular key. That can be an identifier for public keys (for use with signature algorithms) as well as an identifier for secret keys (for HMAC algorithms). The `<KeyName>` can only be used in a particular application context, as there is no commonly agreed standard on how to resolve a `<KeyName>` to a key or certificate.
- The `<KeyValue>` element is a container for carrying plain (literal) public keys. For obvious reasons, it does not make sense to include secret keys in the clear, because that would compromise the security of a HMAC. (The same `<KeyInfo>` element can contain an additional X.509 certificate. The `<KeyValue>` must contain the same public key as the ASN.1 encoded X.509 certificate.) The `<KeyValue>` element does carry the key in nested elements which indicate the type of the key.
  - The `<DSAKeyValue>` element contains the `<Y>` element for the *y* parameter of the DSA public key and optionally the elements `<P>`, `<Q>`, `<G>` and `<J>` as well as `<seed>` and `<pgenCounter>` for DSA prime generation as defined by [FIPS186-2].
    All these values are integer values encoded in base64 format.
  - The `<RSAKeyValue>` element contains the `<Modulus>` and the `<Exponent>` of the RSA public key, which are also base64 encoded integer values.
- The `<RetrievalMethod>` element is used for retrieving the key material from another location. This level of indirection allows that multiple signatures point to a key or certificate which is only included once in the document. The `RetrievalMethod` provides the same means for de-referencing contents like a `Reference`, i.e. it uses a `URI` attribute and allows optionally provided Transforms for retrieving and transforming a resource until the final key is selected. The difference is that the result of de-referencing and transforms is not digested but provided as key material.
  The `RetrievalMethod` additionally provides a `Type` attribute which can be used for stating which type of key is provided after de-referencing.
- X.509 certificates are very important in public key crypto systems. The `<X509Data>` element is a container for storing various X.509 related data items:
  - The `<X509Certificate>` element contains a base64 encoded X.509 certificate.
  - The `<X509IssuerSerial>` element contains the distinguished name of the certificates issuer and the certificate serial number.
  - The `<X509SubjectName>` element contains the distinguished name of the subject of the X.509 certificate.

  - ❍ The <X509SKI> element allows providing the base64 encoded SubjectKeyIdentifier X.509v3 extension of the certificate.
  - ❍ For transporting base64 encoded X.509 certificate revocation lists (CRLs), the <X509CRL> element is provided.
- ❑ For re-using OpenPGP key material in XML Signature, the standard provides the <PGPData> element for PGP public key identifiers using <PGPKeyID> children and/or <PGPKeyPacket>s for transporting the PGP key itself.
- ❑ Transporting SPKI public keys [EFLR+99], certificates and SPKI related material is enabled through the <SPKIData> element.
- ❑ The <KeyInfo> element is able to contain arbitrary elements. Even pure text can be stored in it. This is the maximum level of extensibilitym which allows custom solutions to be implemented in an easy way.
- ❑ A different way to include arbitrary contents in <SignedInfo> is by using the <MgmtData> element. This element can carry text data as a syntactic hook for in-band key distribution or key agreements. However, the usage of the MgmtData mechanism is *not* recommended by the XML Signature Recommendation, as no explicit semantics is defined for <MgmtData>. Instead, specific child elements should be defined, like done for W3C XML Encryption.

4.2.6 Embedded objects for enveloping signatures - the Object element

The <Object> element provides a mechanism for embedding arbitrary information into a signature. A signature can contain an arbitrary number of objects. Depending on the application requirements, these objects can be included in the signing process, so <Object>s can be signed or unsigned containers. An <Object> container enables the creation of enveloping signatures, i.e. signatures which contain the signed contents.
<Object>s can either carry base64 encoded binary data or XML structures. XML structures inside an <Object> are directly addressable using standard XML mechanisms, so that even applications which cannot process XML Signatures can access the object's content. The process of unwrapping the signed contents becomes much easier as for PGP or S/MIME because the signed XML content *is* already unwrapped.

5 Confidentiality Systems – State of the Art

On the internet, various systems for data confidentiality are in wide use. In this chapter, well-known ones like SSL/TLS, IPSec, PGP, S/MIME and W3C XML Encryption are described. Additionally to these encryption based systems, XML Access Control is mentioned.

5.1 Encryption of Unstructured Data

Many encryption systems today are able to encrypt arbitrary types of data. These systems do not need knowledge about the internal structure of the plaintext data. This enables encryption of arbitrary plaintext formats. For instance, SSL/TLS encrypts TCP (transport layer) network connections regardless of the transmitted payload's structure. PGP and S/MIME encrypt arbitrary-files and email attachments, regardless of the file's or attachment's internal data structure.

5.1.1 Example: IP Security Protocol (IPSec)

The "*IP Security Protocol*" (IPSec) Working Group [FraTs02] created a set of specifications which define security services for the IP layer. These security services include confidentiality and integrity. IPSec key management allows usage of X.509 certificates and OpenPGP keys.

5.1.2 Example: Transport Layer Security (TLS)

The most common encryption mechanism on the internet is the "*Secure Socket Layer*" (SSL) protocol respective its successor, the "*Transport Layer Security*" (TLS) protocol [DiAl99]. TLS is used to provide confidentiality, data integrity and peer-entity authentication for TCP connections. After creating a TCP/IP connection between TLS client and TLS server, the SSL/TLS session is established to provide the above security services. During the handshake phase, symmetric keys are generated which are used for transparently encrypting the communication between both parties. After establishing the TLS session, application data is transparently transmitted through the encrypted tunnel.

5.1.3 Example: S/MIME

"*Secure MIME*" (S/MIME) is primary used for encrypting (and signing) E-mail and is a standard specified by the IETF in [Rams99]. Generally, it can be used in any protocol which utilizes MIME ("Multipurpose Internet Mail Extensions", defined in [FrBo96]) for segmenting messages.
Message format: S/MIME uses the "*Cryptographic Message Syntax*" (CMS) [Hous99] as syntax for protected messages. S/MIME provides data origin authentication and confidentiality for MIME messages. CMS is defined using

the ASN.1 language [ITU-T X.680 | ISO 8824] for structuring the message's envelope.
Key material: S/MIME relies on X.509 certificates. The family of S/MIME related standards defines how to use various symmetric and asymmetric encryption, digital signature, MAC and digest algorithms.
Binding between data and management information: The basic data item which can be signed or encrypted in S/MIME is a MIME body part. S/MIME can only refer to signed and encrypted data inside the same MIME message.

5.1.4 Example: OpenPGP

The "*OpenPGP*"-Format, defined by the IETF in [CDFT98], is the official standardization of the "*Pretty Good Privacy*" (PGP) message format, originally defined by Phil Zimmermann. OpenPGP compatible implementations provide the security services data origin authentication and confidentiality for E-mail messages and files.
Message format: OpenPGP is an independent message format that is not compatible with the S/MIME message format.
Key material: Additionally to secure messages, OpenPGP defines a format for representing private and public keys and for signed public keys. Signing public keys enables OpenPGP users to create a "web of trust" by cross-certifying each others public key. As a result, the PGP trust model does not require a centralized key management authority like a certificate authority which issues X.509 certificates. Nevertheless, the PGP web of trust can be built on a centralized key certification architecture.
Binding between data and management information: The binding between an OpenPGP signature and the signed data can be in two different ways: When used in E-mail, the signature surrounds the signed text in the same MIME body part. When encrypting E-mail, the envelope replaces the encrypted body. When signing files in a file system, the binding is done using the filename: The signature has the same filename as the signed file, but a `.sig` extension is added to the filename. When signing the file `1.txt`, the signature is in `1.txt.sig`.
Additionally to encrypting individual files or attachments, commercial versions of PGP offer a tool called PGPDISK which can be used for emulating harddrive volumes under Microsoft Windows.

5.2 Selective Field Confidentiality

[ITU-T X.800 | ISO 7498-2] introduces "*selective field confidentiality*" as a service for encrypting only selected parts (fields) of a data item:

> *"This service provides for the confidentiality of selected fields within the (N)-user-data on an (N)-connection or in a single connectionless (N)-SDU."*

This is a general-purpose description for creating application oriented encryption systems which encrypt only parts of the user data. This requires that the data to be secured has an internal structure. As internal structure is application dependent, each application must define its own mechanism on how selective field confidentiality is implemented. The specific implementation i.e. which fields have to be kept confidential depends on the application scenario and on the data structure. This service can be applied to (parts of) protocol messages, selected fields in databases or (parts of) documents.

5.3 W3C XML Encryption

5.3.1 Introduction

The W3C XML Encryption Recommendation [ER02] specifies a confidentiality security mechanism for XML. When talking about "XML Encryption" in this document, the W3C document "XML Encryption Syntax and Processing Recommendation" [ER02] is meant. As already seen in the XML Signature recommendation, customizing a security mechanism for XML offers various benefits:

- An XML element containing XML Encryption information can act as a container for encrypted data (payload) or as container for encrypted key material (key management) or both.
- XML Encryption 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 usually all child nodes (and all their descendants)) inside an XML document or
  - arbitrary binary contents outside of an XML document.
- XML Encryption allows direct inclusion of the encrypted contents into the container or to de-reference the encrypted contents via the URI/ Transforms mechanism already known from XML Signature.
- XML Encryption offers key management facilities for
  - symmetric wrapping of secret keys (secret key needed to retrieve secret key)
  - key transport of secret keys (private key needed to retrieve secret key)
  - key agreement using Diffie-Hellman

Related to encrypting XML, the W3C XML Encryption Recommendation allows two different granularity levels: the encryption of full subtrees (a single element and all its 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). Encrypting an element *always* implies encryption of all descendants of that element. Figure 5-1 on page 70 illustrates these possibilities:

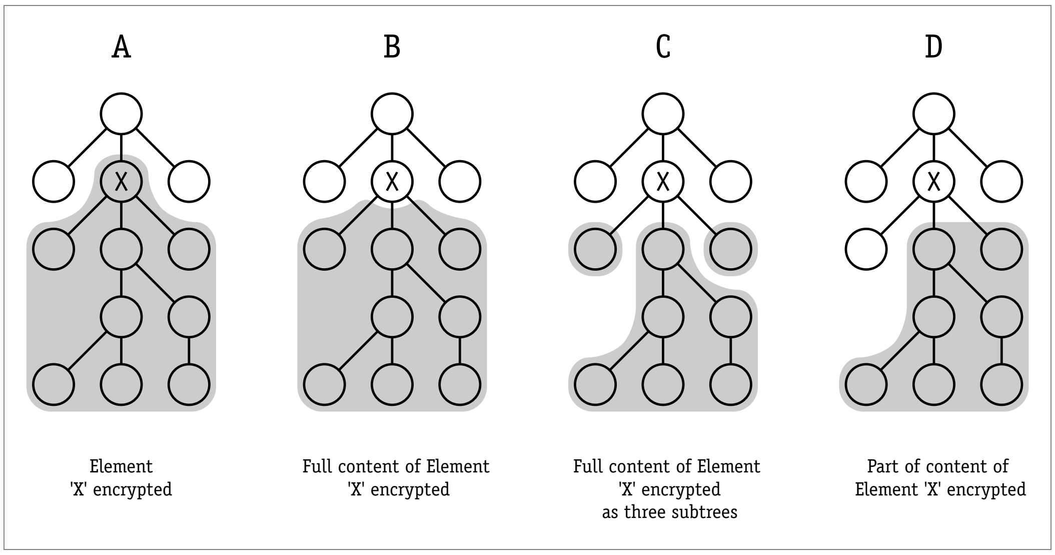

A
B
C
D
X
Element 'X' encrypted
Full content of Element 'X' encrypted
Full content of Element 'X' encrypted as three subtrees
Part of content of Element 'X' encrypted

Figure 5-1: W3C XML Encryption modes

- Example A in figure 5-1 shows the encryption of the subtree rooted by the element 'X'. The element and all its descendants are encrypted into a single `<xenc:EncryptedData>` element. The `Type` attribute has a value of `"&xenc;#Element"`. The decryption result *must* be a single element (which itself may have arbitrary children).
- Example B in figure 5-1 shows the encryption of the content of element 'X'. All children of the element and their respective descendants are encrypted into a single `<xenc:EncryptedData>` element. The `Type` attribute has a value of `"&xenc;#Content"`.
- Example C in figure 5-1 shows subtree encryption applied three times to each child of element 'X'. Each subtree rooted by a child node of element 'X' is encrypted into a separate `<xenc:EncryptedData>` element. The difference to example B is that three independent encryptions are performed. Each encryption can have different encryption properties. Encrypting multiple child nodes of an element usually refers to encrypting *all* children of the given element (like in example B). XML Encryption implementations must be aware to handle such a use case appropriately.
- Example D in figure 5-1 shows another way to use content encryption: two subsequent subtrees are grouped together and are encrypted together. This is a special case of example B which shows that even parts of the children can be encrypted, as long as they are direct siblings. This could even be used to split a single Text node (a sequence of multiple character information items) into encrypted and unencrypted parts.

Like example C, this use case is also an obfuscated example on what is possible with XML Encryption but is not explicitly defined in the spec.

The decryption of the examples A and C leads to single elements, i.e. the decrypted octets can be parsed directly by the XML parser. The octets resulting after decryption of examples B and D are not directly parseable, but must be wrapped into a start-tag/end-tag combination. The "*XML Fragment Interchange*" [GroVei01] introduces the term "*well-balanced*" to describe (part of) the content of an element:

> *"A region (consecutive sequence of characters) of an XML document is said to be (well-)balanced if it matches production "43 content" of XML 1.0. Informally this means that, if the region includes any part of the markup of any construct, it contains all of the markup of that construct (e.g., in the case of elements, all of both the start and end tag)."* [GroVei01]

This means that every octet sequence which is allowed to occur between the start- and end-tags of an element is well-balanced, e.g. `"foo <a/>q<b/>"`. Well-balanced content is data that is allowed as element content.

5.3.2 Encryption for multiple recipients

5.3.2.1 Encrypting the same content

Encrypting a given resource for multiple recipients can be done in different ways. The trivial case is that all recipients are allowed to see the same portion(s) of the 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 for each recipient an `<xenc:EncryptedKey>` element which contains the content encryption key encrypted under the recipient's key.

5.3.2.2 Super-Encryption

Another way to encrypt contents for multiple recipients applies when the recipients are allowed to see different portions of a given document, i.e. when encrypted contents are encrypted multiple times (see figure 5-2).

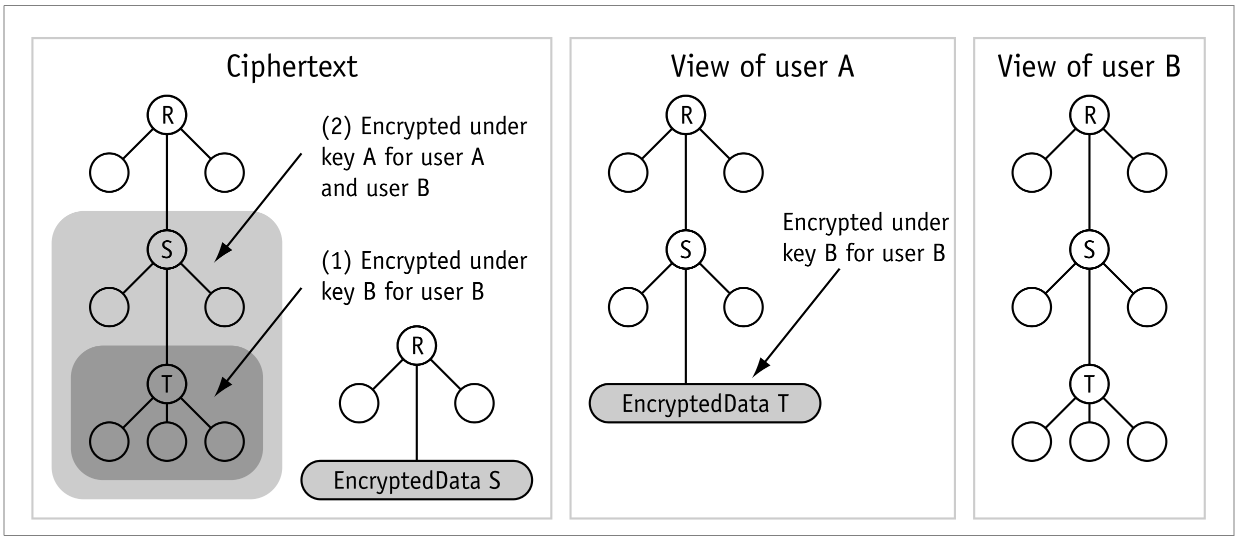

Ciphertext
R
S
T
(2) Encrypted under key A for user A and user B
(1) Encrypted under key B for user B
R
EncryptedData S
View of user A
R
S
Encrypted under key B for user B
EncryptedData T
View of user B
R
S
T

Figure 5-2: W3C Super-Encryption: Encrypting <EncryptedData>

Encrypting parts of an XML tree leads to substitution of the plaintext structure with an `<xenc:EncryptedData>` element. Super-encryption applies when this `<xenc:EncryptedData>` element and some of its siblings or ancestors are by itself encrypted.

In figure 5-2, the subtree rooted by element T is encrypted under a key B for a recipient B. After that first encryption step, the subtree rooted by the element S is encrypted under key A for the recipients A *and* B. Recipient B possesses the keys A and B. Recipient A only possesses key `A`.

After the two subsequent encryption steps, the document contains the R element and two unencrypted nodes as well as an `<xenc:EncryptedData>` element which contains the encrypted element S and its descendants.

Both recipient A and recipient B can decrypt the outer `<xenc:EncryptedData>` element as both recipients possess key `A`. The decrypted element S contains the inner `<xenc:EncryptedData>` element. Only recipient B can decrypt the inner `<xenc:EncryptedData>` element as only recipient B possesses key `B`.

Recipient A is aware of the *existence* of a part in the document which he is not able to decrypt; there is information leakage to recipient A that more powerful users of the system exist. The term 'powerful' refers to the ability to decrypt content. 'More powerful' means more content can be decrypted as the powerful user possesses more keys than a less powerful user.

Based on the number of octets of the undecryptable ciphertext, recipient A can make good estimation on how large the undecrypted plaintext is.

Recipient B possesses both content decryption keys A and B and performs decryption in two steps: After decrypting the `<EncryptedData>` element

which contains the S plaintext, the `<EncryptedData>` element containing T is decrypted. After both decryption steps, the document is completely decrypted and fully available to recipient B. Recipient B is aware that the super-encryption of the innermost `<EncryptedData>` 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 capabilities of other users.

5.3.3 Serialization of XML for XML Encryption

For encrypting large amounts of data, usually symmetric encryption algorithms are used. Symmetric encryption algorithms transform a plaintext octet string into a ciphertext octet string and vice versa. XML itself is a tree-structure which must be converted into an octet string prior encryption and converted back from an octet string to a tree-like structure after decryption.
For encrypting parts of an XML document, the encrypting application selects a well-balanced piece of XML and serializes it into a UTF-8 encoded octet sequence.
Special care must be taken on namespace nodes and attributes in the XML namespace: Moving encrypted data into a different context can lead to an inconsistent result after decryption. This happens if the decrypted plaintext uses namespace prefixes without defining them.
To illustrate the problem, the octets of the <b> element in the example 5-1 are moved to another context.

```
<a xmlns:foo="http://foo.com/">
   <b foo:attr="some attr which utilizes the foo namespace" />
</a>
```

Example 5-1: Octet sequence in original context

In the XML code in example 5-1, the <a> element contains the declaration of the `foo` namespace. If the <b> element is improperly serialized like in example 5-2, it looses information which namespace was bound to the `foo` prefix:

```
<b foo:attr="some attr which utilizes the foo namespace" />
```

Example 5-2: Insufficiently serialized element

After inserting the octets from example 5-2 into another context like in example 5-3, the result is useless.

```
<anotherA xmlns="http://bar.com/#movesBToWrongDefaultNamespace">
   <b foo:attr="some attr which utilizes the foo namespace" />
</anotherA>
```

Example 5-3: Wrong insertion into different context

There does not exist any information which namespace was bound to the `foo` prefix. This leads to the problem that the moved octets cannot parsed with a namespace enabled parser. Additionally, the <b> element which has not been in a namespace in the example 5-1 context is in a wrong default namespace in example 5-3. The missing `foo` declaration can be avoided by canonicalizing the octets. The problem with the wrong default namespace can only be avoided by explicitly stating that the default namespace is empty.
In example 5-4, a representation of the <b> element which contains all relevant information is.

```
<b xmlns="" xmlns:foo="http://foo.com/"
   foo:attr="some attr which utilizes the foo namespace" />
```

Example 5-4: Correct serialization of <b> element

All infoset information is preserved, if the snippet from 5-4 is moved to another (arbitrary) context like in example 5-5:

```
<anotherA xmlns="http://bar.com/#movesBToWrongDefaultNamespace">
   <b xmlns="" xmlns:foo="http://foo.com/"
      foo:attr="some attr which utilizes the foo namespace" />
</anotherA>
```

Example 5-5: Correct insertion into different context

Canonical XML can solve a part of the problem. Problems occur if a document subset is to be canonicalized which has some namespace nodes omitted from the document subset. In that case, it's possible that the result of canonicalization is not parseable. Additionally, Canonical XML does not emit an xmlns="" declaration in apex nodes; but this is necessary (see above). The term 'apex node' refers to the "Exclusive Canonical XML" recommendation: *"An apex node is an element node in a document subset having no element node ancestor in the document subset."* [BER02]
Special care must also be taken for `xml:*` attributes. If e.g. *Exclusive XML Canonicalization* is used for serialization of the plaintext and if the decrypted plaintext is placed into a different context, the new context may have e.g. a different value for the `xml:base` attribute which would result in errors while retrieving relative URIs (see the XML Encryption spec [ER02] for a detailed discussion of that topic.).

5.3.4 An Example of XML Encryption

To show how XML Encryption can look like in practice, consider the plaintext in example 5-6 (taken from [ER02]), which represents a payment transaction containing public information (like the name of the account owner) and confidential information (the credit card data).

```
<?xml version='1.0'?>
<bank:PaymentInfo xmlns:bank="http://example.org/paymentv2">
  <bank:Name>John Smith</bank:Name>
```

Example 5-6: Sample plaintext prior W3C encryption

```
  <bank:CreditCard Limit="5,000" Currency="USD">
    <bank:Number>4019 2445 0277 5567</bank:Number>
    <bank:Issuer>Example Bank</bank:Issuer>
    <bank:Expiration>04/02</bank:Expiration>
  </bank:CreditCard>
</bank:PaymentInfo>
```

Example 5-6: Sample plaintext prior W3C encryption

The result could like in example 5-7, if the `<bank:CreditCard>` element is selected for encryption (encrypting the complete subtree).

```
<?xml version='1.0'?>
<bank:PaymentInfo xmlns:bank="http://example.org/paymentv2">
  <bank:Name>John Smith</bank:Name>
  <xenc:EncryptedData
       xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
       Type="http://www.w3.org/2001/04/xmlenc#Element">
    <xenc:CipherData>
      <xenc:CipherValue>A23B45C56</xenc:CipherValue>
    </xenc:CipherData>
  </xenc:EncryptedData>
</bank:PaymentInfo>
```

Example 5-7: Plaintext after partly W3C XML encryption

The `<bank:CreditCard>` element was substituted by an `<xenc:EncryptedData>` element of type `Element`. The `<xenc:EncryptedData>` element contains the `<xenc:CipherData>` element which uses a `<xenc:CipherValue>` element for storing the serialized and encrypted `<bank:CreditCard>` element and all its descendants.

5.3.5 Ciphertext Locations

The above example uses the `<xenc:CipherValue>` to include the ciphertext directly in the `<xenc:EncryptedData>`. W3C XML Encryption also allows creating a *reference* to the ciphertext using the `<xenc:CipherReference>` element, which is similar to the `<Reference>` element from the XML Signature recommendation: The `@URI` attribute is used to identify the location of the ciphertext. An optional `<xenc:Transforms>` element can be used to 'mangle' the de-referenced contents until the ciphertext is extracted. This works the same way the `<Transforms>` element works for signed contents.
This mechanism allows storing ciphertext in detached locations:

- ❑ elements in the same document or
- ❑ an XML file which contains base64 encoded ciphertext, which is extracted using XPath transforms.
- ❑ a non-XML resource somewhere on the network, e.g. like a streaming audio or video file or
- ❑ a non-XML resource on another place which can be identified by URI, e.g. a file on the local hard disk.

5.3.6 XML Encryption Key Management

In the above example, there is no provision to describe which key is necessary for decryption. XML Encryption offers three ways how encrypted data and its associated key can 'find' each other:

- An `<xenc:EncryptedData>` element (as well as the `<xenc:EncryptedKey>` element) can contain an optional `<KeyInfo>` element for carrying key management information (encrypted data points to key).
  - The `<KeyInfo>` of a `<xenc:EncryptedData>` refers to a content encryption key.
  - The `<KeyInfo>` of a `<xenc:EncryptedKey>` refers to a key encryption key.

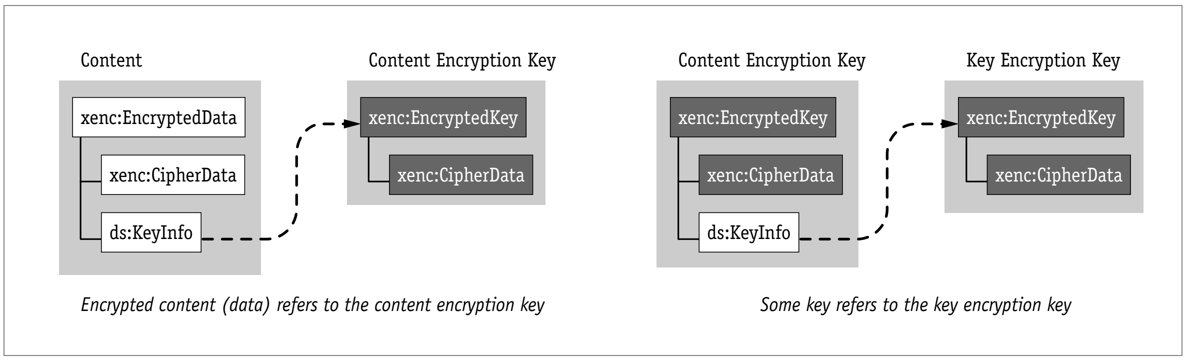


Figure 5-3: Reference from data to key

- An `<xenc:EncryptedKey>` element can contain a reference to the data or key material encrypted under this key (key points to encrypted data).
  - The `<xenc:DataReference>` of a content encryption key points to an `<xenc:EncryptedData>`.
  - The `<xenc:KeyReference>` of a key encryption key points to an `<xenc:EncryptedKey>`.

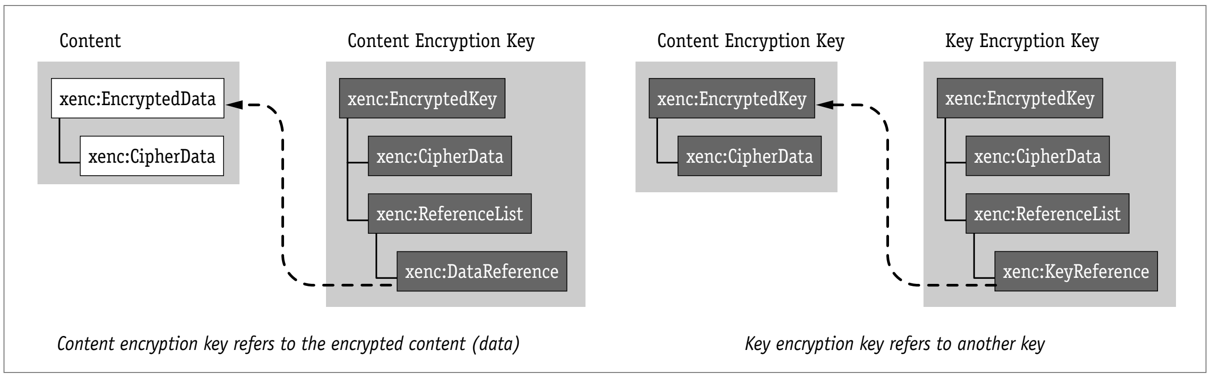


Figure 5-4: Reference from key to data

- ❑ The binding between key and encrypted data can be implicitly defined for the processing application, i.e. there is no reference inside the key or the encrypted data. For example, an application could use only one predefined key.

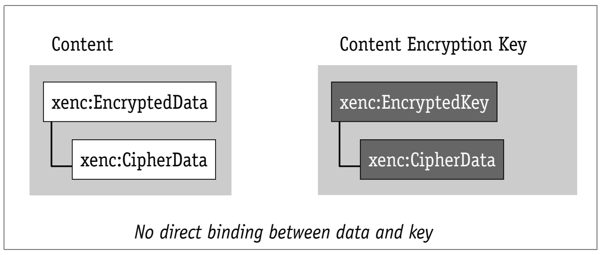


Figure 5-5: No reference between data and key

References to key material can be recursive. That chain of references must be followed until the decryptor determines the final key. For instance,

- ❑ the encrypted data payload was encrypted under a symmetric content encryption key K1.
- ❑ K1 itself is encrypted under the symmetric key encryption key K2.
- ❑ K2 is encrypted under the symmetric key K3.
- ❑ K3 is encrypted under a public key with a given KeyName.

Such a chain must be followed until all key material required to decrypt the contents is available. This powerful construct can lead to an infinite recursion, if an attacker creates a structure where two keys reference each other for decryption: Such a malicious structure could claim that key A was encrypted under key B and key B was encrypted under key A. Implementations of XML Encryption must be aware of this threat and defeat it by detecting loops and/ or by limiting the amount of computing and/or network resources which can be consumed by a key retrieval process.

5.3.7 XML Key Management

The W3C XML Key Management Recommendation [Hal02] specifies protocols for distributing and registering public keys. It is intended to be used by XML Signature and XML Encryption. The “XML Key Management Specification” (XKMS) comprises two parts — the “XML Key Information Service Specification” (X-KISS) and the “XML Key Registration Service Specification” (X-KRSS).

5.4 Information Disclosure in Encryption Systems

Based on the ciphertext size (number of octets), an attacker can deduce the approximate size of the plaintext. Generally, systems can be constructed which provide traffic flow confidentiality or to hide the size of the ciphertext.

Implementations like TLS, IPSec, S/MIME and PGP do not offer any provisions for providing traffic flow confidentiality or hiding the content size.
W3C XML Encryption offers two different ways to handle ciphertext (see 5.3.5 "Ciphertext Locations" on page 75):

- ❑ store the ciphertext in an `<xenc:CipherData>` element or
- ❑ use a link to the ciphertext with an `<xenc:CipherReference>`.

If the ciphertext is directly included using a `<xenc:CipherData>` element, an attacker can make an estimation about the size of the plaintext, like in encryption systems for unstructured data.
Using a `<xenc:CipherReference>` *can* hide the ciphertext from an attacker, if the ciphertext is stored in an access control restricted area. For instance, the ciphertext could be stored in a password protected area of a web server, so that the ciphertext is only disclosed to the decryptor if the decryptor has sufficient credentials to fetch the ciphertext.
An example for such a `<xenc:CipherReference>` element is shown in example 5-8.

```
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
               Type='http://www.w3.org/2001/04/xmlenc#Content'>
  <CipherData>
    <CipherReference URI="http://user@intranet.corp.com/cipherData?id=cipher3982">
      <Transforms>
        <ds:Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
            <ds:XPath xmlns:cipher="http://intranet.corp.com/#encryptedData">
              self::text()[parent::cipher:CipherValue[@Id="cipher3982"]]
            </ds:XPath>
        </ds:Transform>
        <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64"/>
      </Transforms>
    </CipherReference>
  </CipherData>
</EncryptedData>
```

Example 5-8: Retrieving ciphertext from an access control restricted HTTP realm

The ciphertext is stored in an XML instance which can be fetched at the URI `"http://user@intranet.corp.com/cipherData?id=cipher3982"`.
The `"user@"` prefix in front of the host name indicates that the HTTP server requires an authentication with the user name `"user"`. Additionally to the user name, a credential like a password or a cookie is needed which has to be supplied by the decryptor and which is not included in the given HTTP URL. Therefore, only authenticated users are able to see the size of the ciphertext. In this example, the encryption mechanism provides confidentiality for the content, whereas the access control mechanism provides confidentiality for both content and content size.

5.5 XML Access Control

5.5.1 Introduction

The term “XML Access Control” is used in two different ways:

- ❑ Expressing *access control policies* using an XML-based language, namely “*XML Access Control Markup Language*” (XACML) [GoMo02] and
- ❑ Using these XACML–based policies to enforce *fine-grained* access control for XML–based resources.

“*Fine-grained access control*” refers to the concept that an access control policy does not only define access to an XML document as a whole (full access to the complete document or nothing), but that access to individual nodes in the document can be defined. In various works on XML access control [BBC+, BCFM00, BCFM99, DVPS01, DVPS02], the authors propose systems in which the *access control processor* (both *access control enforcement function* and *access control decision function*) has multiple policies which are applied to a given request:

- ❑ Access control policies which restrict access for a class of XML documents. Such a class is a group of XML documents which all conform to the same DTD or XML Schema. For instance, such a class could be all invoice documents.
- ❑ Access control policies which restrict access to individual XML documents.

The general process of fine-grained access control on a given document is illustrated in figure 5-6 on page 80 and works as follows:

1. The initiator requests access to a given XML document target.
2. The requested document is parsed into a tree.
3. The access control processor retrieves all relevant access control policies:
   - ❍ policies which apply to the overall DTD/Schema of the requested document and
   - ❍ policies which have been defined specifically for the requested document).
4. According to the applying policies, the XML tree is labeled. The final result of the labeling process is a flag for each node, namely whether the node is to be included in the result (‘permit’ +) or excluded (‘deny’ –).
5. In the final transformation step, the document is pruned, i.e. all nodes which are labeled ‘deny’ are removed from the document.

6. The pruned document is serialized and returned to the initiator.

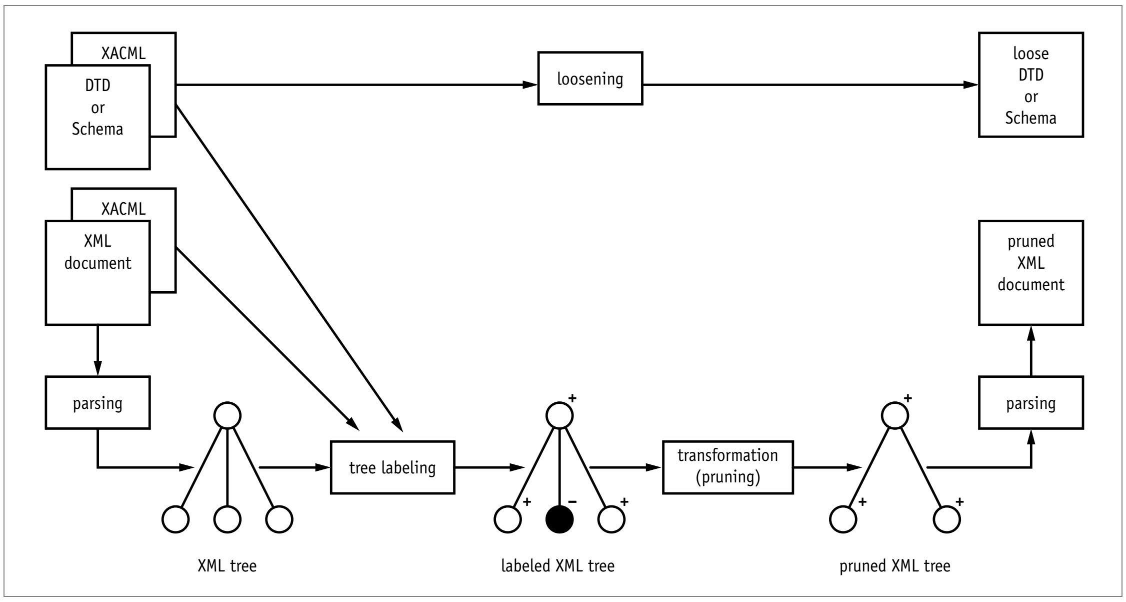


Figure 5-6: XML Access Control pruning process

to prune: Pruning is the process in which nodes are removed from a tree.

The removal of nodes during the pruning process can have the consequence that the resulting XML document is no longer valid according to a given DTD or Schema. For that reason, the DTD/Schema is also processed in a *loosening* step. A simple approach for such a loosening is given in [DVPS00]: all nodes (e.g. attributes) which are defined as required in the original DTD/Schema are indicated to be optional in the loosened DTD/Schema.

5.5.2 The invisible ancestors problem

One question must be discussed in more detail: "*What happens if an ancestor node (namely an element) is labeled 'deny (-)' but a descendant is labeled 'permit (+)'?*".
In this case, an ancestor is confidential which a descendant of the ancestor is not confidential. This is meant by the phrases '*invisible ancestor*' and '*deep visible descendant*'.
Different solutions exist how to deal with such a situation:

5.5.2.1 THE SCHEMA-FRIENDLY SOLUTION

The authors of [DVPS00] suggest the following rule:

> *"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." [DVPS00]*

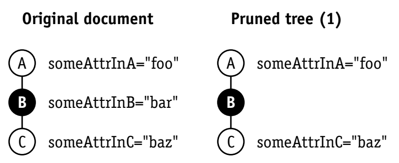
The advantage of this solution is that the damage to Schema-validity is limited: The element structure remains the same as in the original document, but all attributes in the element are removed. The disadvantage is that the existence of the ancestor elements remains visible for the initiator, even if the initiator is not allowed to see the element. 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 is labeled 'deny (–)', while all other elements are permitted to be seen by the initiator:

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

Example 5-9: Input document for 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 result looks like in example 5-10.

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

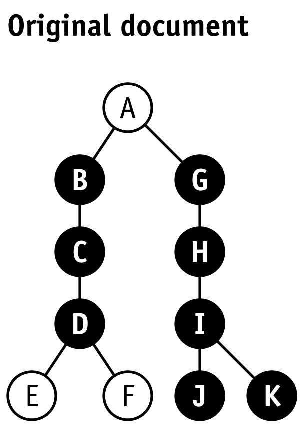


Example 5-10: Result document (element persists)

A complete subtree can also be omitted. A complete subtree has the property that all nodes in the subtree are labeled 'deny'. Example 5-11 illustrates this situation. The elements <A>, <E> and <F> are labeled 'permit', access to all others is denied. To emphasize the pruning result, each element in the original document has an `attr="foo"` attribute.

Original document

```
<A attr="foo">
 <B attr="foo">
  <C attr="foo">
   <D attr="foo">
    <E attr="foo"></E>
    <F attr="foo"></F>
   </D>
  </C>
 </B>
 <G attr="foo">
  <H attr="foo">
   <I attr="foo">
    <J attr="foo"></J>
    <K attr="foo"></K>
   </I>
  </H>
 </G>
</A>
```

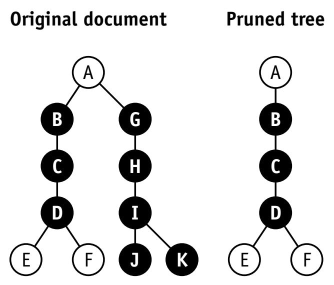


Example 5-11: Original document before subtree removal

Example 5-12 shows the result of the pruning process:

Pruned document

```
<A attr="foo">
 <B>
  <C>
   <D>
    <E attr="foo"></E>
    <F attr="foo"></F>
   </D>
  </C>
 </B>
</A>
```



Example 5-12: Original document after subtree removal

The subtree rooted by `<B>` is not a completely omitted subtree as the elements `<E>` and `<F>` are labeled 'permit'. Therefore, the start- and end-tags of `<B>`, `<C>` and `<D>` persist in the pruned document. The subtree rooted by `<G>` only contains nodes labeled 'deny'. Therefore, the pruned document does not contain any start- or end-tag for the elements `<G>`, `<H>`, `<I>`, `<J>` or `<K>`.

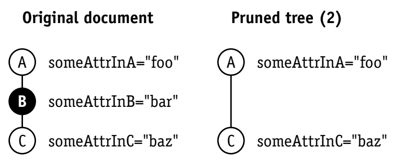
5.5.2.2 *Real* Invisible Ancestors

Another solution for this conflict is to prune out all ancestors like done in *Canonical XML*. No start or end tags are emitted if the element is labeled 'deny (–)'. This has an impact on Schema-validity, if required elements are pruned. On the other hand, completely pruning confidential nodes is a clean solution that is also consistent with the idea of serializing an XML Infoset and Canonical XML.
The result from the previous sections sample would look like in example 5-13 on page 83.

```
<A someAttrInA="foo">

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

</A>
```



Example 5-13: Result document (element removed)

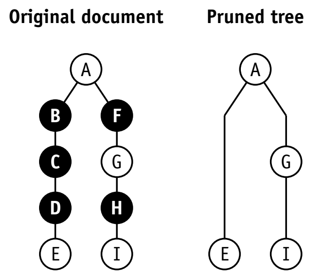
After access to the <B> element is denied and the node is removed, the requester has no knowledge that the element did exist in the original document. This model allows removing ancestors from the document while non-confidential descendants remain accessible and visible to the requester.
The same method applies for if multiple elements are pruned away (see example 5-14).

Original document

```
<A>
   <B><C><D><E></E></D></C></B>
   <F><G><H><I></I></H></G></F>
</A>
```

Pruned document

```
<A>
   <E></E>
   <G><I></I></G>
</A>
```



Example 5-14: Pruning multiple elements

5.5.3 Information disclosure

Using the Schema-friendly approach, complete subtrees can be hidden from an initiator (like shown in example 5-12 on page 82). Deeply nested nodes which are labeled 'permit' enforce that the complete ancestral element structure remains visible.
The 'real visible ancestors'-approach is able to *completely* prevent disclosure of element structure information to the initiator. A maximum of the document's structure can be hidden.

5.6 Summary

Currently available systems for encrypting data usually encrypt a full entity (like a file) or a complete bit stream on the wire. Due to the unknown structure of the plaintext, it is not possible to select only the confidential material for encryption. Inserting the partly encrypted data into unencrypted data is not possible as the decryptor does not know which portions have to be decrypted.

The selective-field-confidentiality service states that only sensitive data fields are to be encrypted. Both the encryptor as well as the decryptor must be able to distinguish between encrypted and unencrypted data, in order to decrypt the correct portion of the data fields.

W3C XML Encryption is one possible system implementing a selective-field-confidentiality security mechanism with relatively coarse granularity, especially designed for XML data.

XML Access Control defines a way in which the access to confidential material inside an XML document can be restricted by an access control processor.

By comparing *W3C XML Encryption* and *XML Access Control*, it is seen that *W3C XML Encryption* is restricted to encrypt full subtrees, while *XML Access Control* can remove sensitive material from arbitrary positions of the tree.

W3C 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 which enforces the policy by removing nodes before giving the results to the client.

W3C XML Encryption does not allow ‘invisible ancestors’ which certain *XML Access Control* systems do.

W3C XML Encryption discloses the position of the plaintext. The ciphers specified by the W3C XML Encryption recommendation allow derivation of the plaintext size by the size of the ciphertext. The explicit use of `<ds:KeyInfo>` elements by W3C XML Encryption also reveals which user is intended to decrypt particular `<xenc:EncryptedData>` elements. *XML Access Control* has the ability to hide most of this information.

The rest of this work is to achieve the advantages of both XML Encryption and XML Access Control with cryptographic mechanisms.

6 Requirements for the New Confidentiality System

The previous chapter introduced W3C XML Encryption and XML Access Control. This chapter describes the requirements for the new confidentiality system, which will be called *'XML Pool Encryption'* for the rest of this document. The new system will be capable to encrypt *arbitrary* parts of an XML document. Encrypting arbitrary parts of an XML document means that selected parts of the original document structure are hidden by the confidentiality mechanism.

Advantages and drawbacks of the existing mechanisms. Both, W3C XML Encryption and XML Access Control, provide confidentiality for XML documents by using different security mechanisms:

- ❏ W3C XML Encryption is a cryptographic system that is constrained to the encryption of subtree structures in an XML document.
- ❏ XML Access Control can remove arbitrary parts from an XML document, but requires an access control processor for this task.

Goal of this work. The goal of this work is to develop a system which does not require an access control processor *and* which can encrypt *arbitrary* parts of an XML document, not only subtree structures. Goal is to provide confidentiality for arbitrary nodes in an XML document and therefore hide the document's original structure.

Requirements. It should be possible to

- ❏ provide confidentiality for arbitrary nodes (not only subtrees),
- ❏ provide confidentiality for the *original structure* of the document by removing the nodes from their original position,
- ❏ provide confidentiality for the *amount* of confidential nodes

Trust policy. The system introduces two classes of entities:

DACP: A *DACP* (*'document access control provider'*) is an entity which provides access to a document. The DACP entity uses the DACP service. The DACP provides confidentiality for the plaintext document or parts thereof. The DACP entity decides which nodes are confidential and encrypts these nodes.

user: A *user* is an entity that is given an public document.

The DACP can give key material to the user. Based on the composition of the key material, the user can decrypt a selected subset of the encrypted nodes and insert the decrypted nodes back into the document. The DACP can grant different views to the document by giving different key material to different

users. A user without key material cannot decrypt any encrypted node. Such a user could be an attacker who gained unauthorized access to the public document.
The DACP has complete access to all public nodes, decides which users are allowed to see which portions of the document and is responsible for enforcing his decision by encrypting the respective contents.
The users get access to specific parts of the document by being given their key material. Each user is free to disclose the decrypted nodes or his key material to third parties.

No constraints about the storage location of encrypted nodes. The system should not impose particular constraints on the DACP how it organizes the storage of the encrypted nodes, i.e. how many 'pools of encrypted nodes' it creates (see page 88 for definition of this term) and which encrypted nodes are stored in which pool.

Algorithm independence. The system should be algorithm independent, i.e. no particular cryptographic algorithm is to be mandated. Nodes are encrypted using symmetric block or stream ciphers. The key material for the users has to be confidentiality protected and authentic.

7 XML Pool Encryption

This chapter describes the concepts and design principles of XML Pool Encryption.

7.1 Basic mechanism

The idea behind pool encryption is to *remove* confidential nodes from the document tree and to *encrypt* each confidential node individually. These encrypted nodes are stored in a container. After *decryption*, each node can 'find its way back' to its appropriate position in the document, so that it can be restored correctly.

7.2 Terms used in this chapter

This section defines various terms which are needed in the following chapter:

7.2.1 Document states

During the various stages of the XML Pool Encryption process, the same document has different names to indicate its state:

plaintext document: A plaintext document is an XML document, before the document or parts of it are encrypted. All nodes are present in the plaintext document, i.e. all public nodes as well as all confidential nodes.

public document: A public document is a plaintext document after application of the pool encryption procedure. The confidential nodes have been removed from the plaintext document and are encrypted. The pool of encrypted nodes (see page 88) can be either inside the public document or in a separate entity.

decrypted document: A decrypted document is a public document after (partial) decryption. The decrypted nodes have been restored into the appropriate position.

7.2.2 Node types

public node: A public node is a node in a document that is not confidential. A public node is not modified by encryption or decryption processes. Note: If a plaintext document contains public nodes, the public document also contains the same public nodes.

confidential node: A confidential node is a node in a plaintext document that is confidential, i.e. certain users are not allowed to see it. The attribute 'confidential' means that the node is selected for further confidentiality protection. In further processing steps, the confidential node is transformed into an encrypted node.

encrypted node: An encrypted node is a confidential node which (1) has been removed from its associated plaintext document during the pruning procedure and which (2) has been encrypted under the node key. Several encrypted nodes can be combined in a pool of encrypted nodes.

pool of encrypted nodes: A pool of encrypted nodes contains one or more encrypted nodes.

decrypted node: A decrypted node is an encrypted node which has been decrypted using its associated node key. This decryption is carried out as part of the node decryption procedure.

restored node: A restored node is a decrypted node which has been restored inside the public document to create the decrypted document. The re-integration into the document is part of the node restoration procedure.

node key: A node key is a unique symmetric key that is needed to encrypt one single confidential node's plaintext data and its associated label.

7.2.3 Components of the pool encryption procedure.

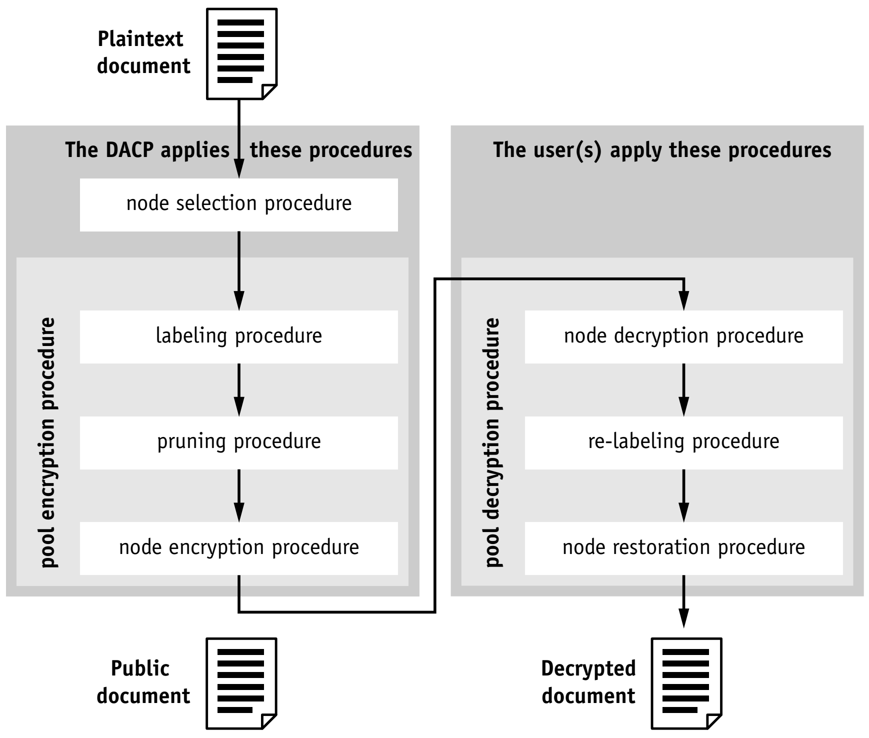

Plaintext document
The DACP applies these procedures
node selection procedure
pool encryption procedure
labeling procedure
pruning procedure
node encryption procedure
Public document
The user(s) apply these procedures
pool decryption procedure
node decryption procedure
re-labeling procedure
node restoration procedure
Decrypted document

Figure 7-1: Overview to the procedures

pool encryption procedure: The pool encryption procedure is the overall process of XML Pool Encryption. This includes the labeling procedure, the pruning procedure and the node encryption procedure. (see figure 7-1 on page 89 for an overview of all existing procedures)

node selection procedure: Before the pool encryption procedure can be applied, the DACP ('document access control provider') has to decide which of the plaintext document's nodes are confidential nodes and therefore must be encrypted during the subsequent steps.

labeling procedure: During the labeling procedure, each node in a plaintext document is assigned a label.
The labeling procedure is formally described in the section "Labelling procedure" on page 122.

pruning procedure: During the pruning procedure, the confidential nodes are removed from the plaintext document.

node encryption procedure: During the node encryption procedure, each confidential node and its respective label is symmetrically encrypted under its node key.

7.2.4 Components of the pool decryption procedure

pool decryption procedure: The pool decryption procedure is the counterpart to the pool encryption procedure: During the pool decryption procedure, both the pool decryption procedure and the node restoration procedure are executed.

node decryption procedure: During the node decryption procedure, each encrypted node is symmetrically decrypted using its associated node key. The result is a set of decrypted nodes and their labels.

node restoration procedure: During the node restoration procedure, the decrypted nodes are restored in the public document, which leads to the decrypted document.

7.2.5 Terms about the labeling procedure

label: During the labeling procedure, a label is assigned to each node. Each label consists of one left value and one right value.

left value: The left value is a label value that is assigned at first to a node.

right value: The right value is the second label value that is assigned to a node.

label value: A label value is a numeric value. Two label values (the left value and the right value) are assigned to a node during the labeling procedure. These both label values form the node's label.

7.3 Concepts and design principles

7.3.1 Removing nodes from the tree

The first novel concept of XML Pool Encryption is that confidential material is removed from its original position in the document. Therefore, it is not possible to deduce whether a specific region of the document contains confidential material or not.

The steps to perform the pool encryption procedure are

1. label all nodes in the plaintext document (labeling procedure)
2. remove (prune) confidential nodes from their original position in the plaintext document (pruning procedure),

3. encrypt each confidential node (the confidential node's plaintext data and its label) separately (each confidential node under its individual so-called 'node key') and
4. collect all encrypted nodes in the pool of encrypted nodes.
5. Each user is given the necessary key material as a set of node keys. For each encrypted node which the user is permitted to decrypt, the key set contains the respective node key (node decryption procedure).
6. The user decrypts the respective encrypted nodes and inserts the decrypted nodes back into the appropriate position in the document (node restoration procedure).

Given the plaintext document in figure 7-2, the confidential nodes are marked black (E, F, J, M, N, O, P, U and V). The white nodes are public nodes.

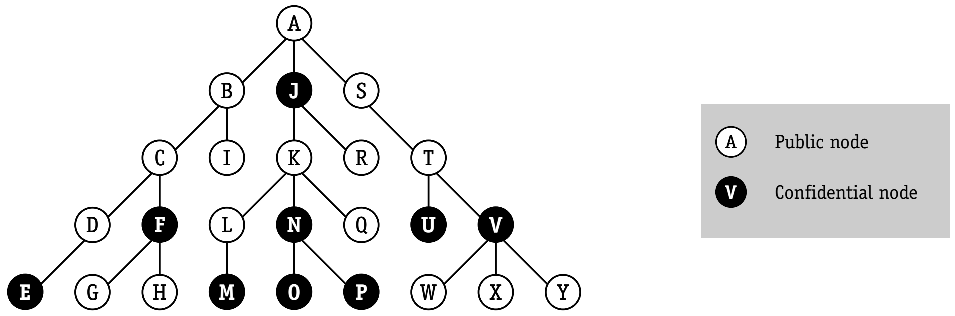


Figure 7-2: Plaintext document with confidential nodes

The first step performed by the DACP is the labeling of all nodes in the plaintext document. The labeling procedure itself will be described later. After labeling all nodes, the confidential nodes are removed from their original position in the plaintext document. Then each confidential node is encrypted individually. All encrypted nodes are collected in a pool of encrypted nodes (see figure 7-3 on page 92). Figure 7-3 does not specifically show the node encryption procedure itself, but only the result of the pruning procedure. During the pruning procedure, 'orphaned nodes' which have lost their parent node become direct children of their former 'grandparent' (or the next unencrypted ancestor if the grandparent is also encrypted).

orphaned node: An orphaned node is a public node which has a confidential node as parent node.
Note: It is called 'orphan' because it looses its parent node during the pruning procedure.

In figure 7-2, the node J is a confidential node, while the node K is a public node. During the pruning procedure, node J is removed from the document,

i.e. node K becomes an orphaned node. After the pruning procedure, node A (another public node) becomes the new parent of node K (see figure 7-3).
Using a human analogy: As long as the parents cannot care for their children by themselves, the grandparents (or the next living ancestor) are in charge to look for the children and give them a home, until the real parents are back to serve their role as a direct ancestor.
The pool of encrypted nodes (see figure 7-3) is then placed in the public document.

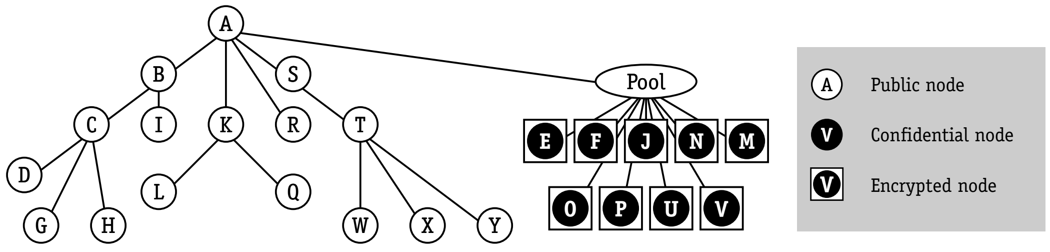


Figure 7-3: Public document (with pool of encrypted nodes)

Depending on which node keys are available to the user, a specific set of encrypted nodes can be decrypted. In the given example, the user is able to decrypt the nodes N, O, P, U and V. The node keys for the nodes E, F, J and M are not available to the user, so these nodes cannot be decrypted. After decrypting a node, both the node's plaintext data and its original position information become available to the user (node decryption procedure). These decrypted nodes are restored back into the document, to the appropriate position (node restoration procedure).
In the given example, the decrypted document differs from both the plaintext document (which contained all nodes) and the public document (in which all confidential nodes were encrypted). Only a user who possesses all node keys (so that he can decrypt all encrypted nodes) can reconstruct a decrypted document, so that the result equals to the plaintext document. The user in the example does not possess all node keys, therefore the document is only partially decrypted (see figure 7-4).

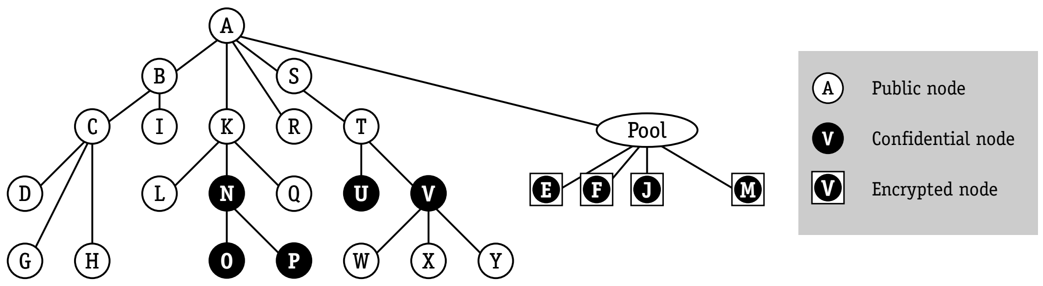


Figure 7-4: Decrypted document (after partial decryption)

The overall pool encryption procedure (see figure 7-5) works as follows:

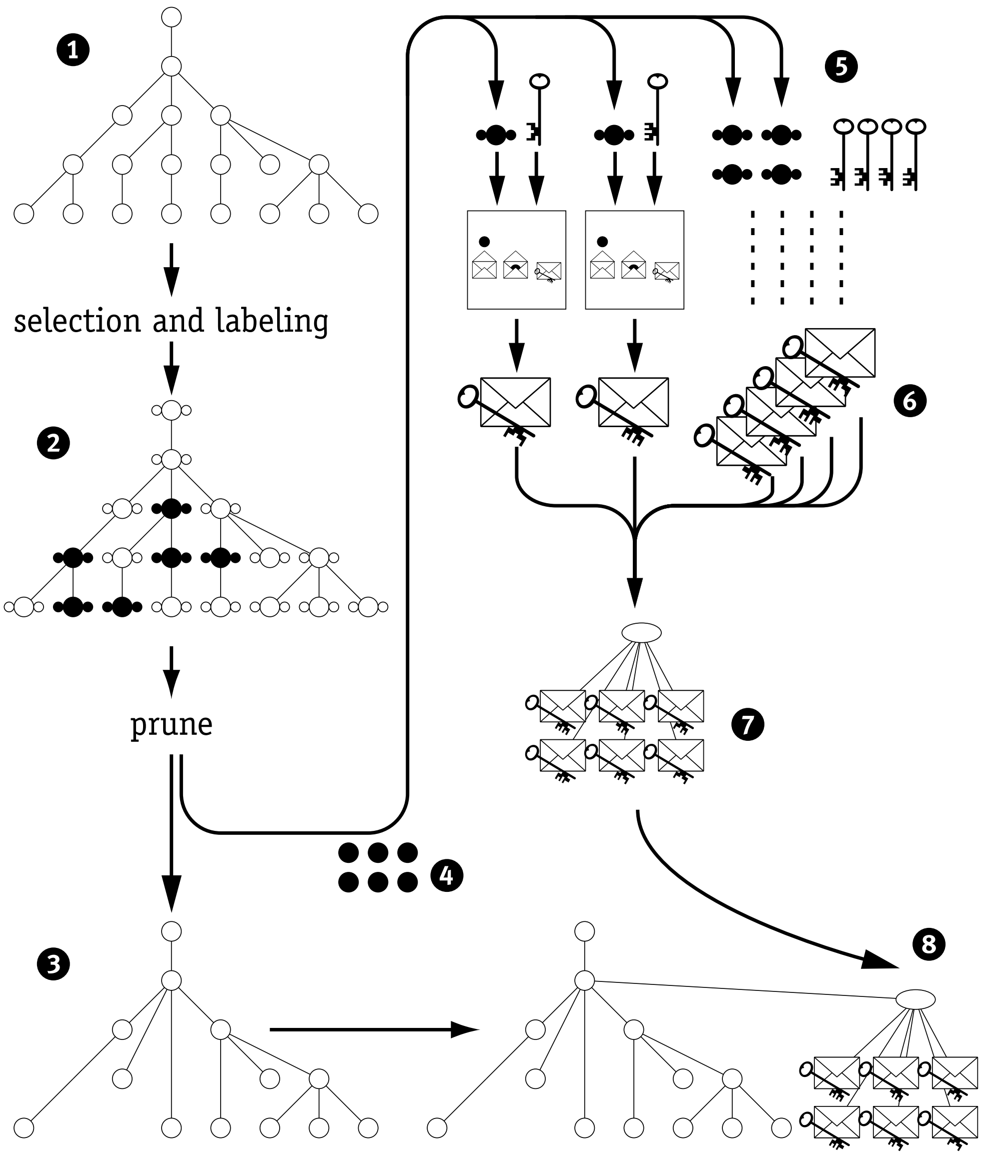


Figure 7-5: Pool encryption – general overview

The XML plaintext document is represented by the input tree ❶. The encrypting party selects the confidential nodes (marked black) and labels all nodes in the document ❷. If multiple users are granted different views to the document by allowing them to decrypt different nodes, the selected nodes from step ❷ represent the union set of all individual sets. All confidential nodes are extracted from the tree prior encryption, so that the pruned tree ❸ and the extracted confidential nodes ❹ are separated. The extracted confidential nodes contain both the node's plaintext data and the original position of the node in the tree (the label). Each extracted confidential node is encrypted with an individually generated node key ❺, resulting in the encrypted nodes ❻. The node encryption procedure is symbolized by putting the confidential node (symbolized as black circle) into the envelope and sealing the envelope using the node key (encryption). These envelopes are bundled in a pool of encrypted nodes ❼. In the example, the pool is added to the public document ❼. Figure 7-5 does not illustrate the key management aspects, i.e. how the node keys are transported to the users.

7.3.2 Pool Key Management

The second concept in XML Pool Encryption is that it can be controlled at a fine granular basis what parts of a document a specific user is permitted to decrypt. The key management for XML Pool encryption is described specifically in section 7.6 on page 113.

7.3.3 Dummy nodes

The third novel concept for XML Pool Encryption is the introduction of '*dummy nodes*'. During the pool encryption procedure, the encrypted nodes are moved altogether into the pool of encrypted nodes. By observing the total number of encrypted nodes in that pool, an attacker may make an estimation on the total number of encrypted material in a given document. By adding dummy nodes to the pool, this attack can be prevented. The concept of dummy nodes is described in section 7.8 on page 118.

7.4 Representing the position of a node in the tree

The biggest problem is to find a sufficient representation for the original position of each node. To make restoring the tree possible, the user needs the original position information of a decrypted node. This is not a trivial task as the user may have only access to a reduced subset of the original tree.

7.4.1 Simple approaches

The absolute position of a node could be described by its ancestor nodes (i.e. the depth in the tree) and the position relative to its siblings. Restoring a node becomes impossible, if the position information is expressed in terms of ancestors and the user does not see the direct ancestors (e.g. because they cannot be decrypted).
One (insufficient) way to represent the position information using an XPath based expression could be

```
/A/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 XPath cannot be evaluated.

A powerful and extensible scheme for representing the position of a node will be outlined in the following section.

7.4.2 "Adjacency List Mode" (ALM)

7.4.2.1 Overview

In the article "Trees in SQL" [Cel00], Joe Celko describes a scheme how tree structures can be stored in flat tables like SQL databases, i.e. how a tree can be converted into a flat table and restored back from the table information. This scheme is called *Adjacency List Mode* (ALM). Given the tree in figure 7-6. The tree contains six nodes (named `Albert`, `Bert`, `Chuck`, `Donna`, `Fred` and

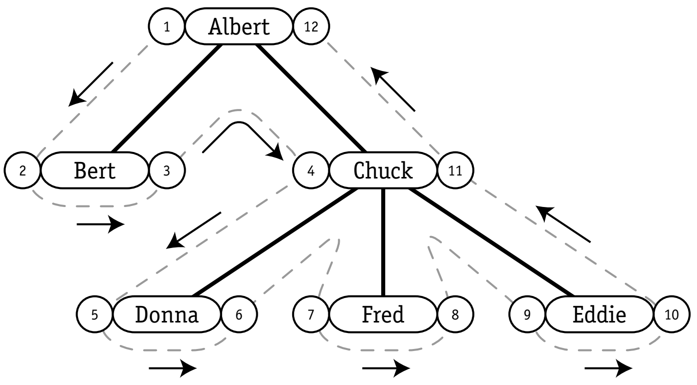


Figure 7-6: Sample tree for the "Adjacency List Mode"

`Eddie`). The algorithm for defining the position of each node has to traverse the full tree and assign labels to all nodes, using an inorder traversal. For each node, the label consists of two integer values, the left value and the right value. The dotted line enveloping the tree is the sequence in which nodes are visited and label values are assigned.

Each time a node is visited for the first time, a counter X is incremented by 1 and the value of X is assigned to the node's 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 node's right value.

Starting with an initial value of $X = 0$, `Albert` is the first visited node in the traversal. `Albert` is visited for the first time, so X is incremented by 1 and the new value is assigned to the `Albert`'s left value: $\text{Albert}_{\text{left}} = 1$. `Bert` is visited for the first time, X is incremented by 1 and the new value is assigned to the `Bert`'s left value: $\text{Bert}_{\text{left}} = 2$. `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 `Bert`'s right value: $\text{Bert}_{\text{right}} = 3$. This method is applied until the full tree has been traversed. After the traversal, each node in the tree has a left value and a right value, i.e. a full label. The dashed grey line outlines the order in which the values are assigned to the label values of the respective nodes, i.e. the path of the traversal walk.

The tabular representation of the tree from figure 7-6 is explicitly written out in table 7-1.

Node	**left value**	**right value**	**label**
Albert	1	12	(1/12)
Bert	2	3	(2/3)
Chuck	4	11	(4/11)
Donna	5	6	(5/6)
Fred	7	8	(7/8)
Eddie	9	10	(9/10)

Table 7-1: Sample table for the "Adjacency List Mode" (see figure 7-6)

Given table 7-1, it is simple to determine the position of two given nodes relative to each other. The range that is spanned by the left value and the right value determine the position in the tree. Figure 7-7 shows the ranges for all nodes.

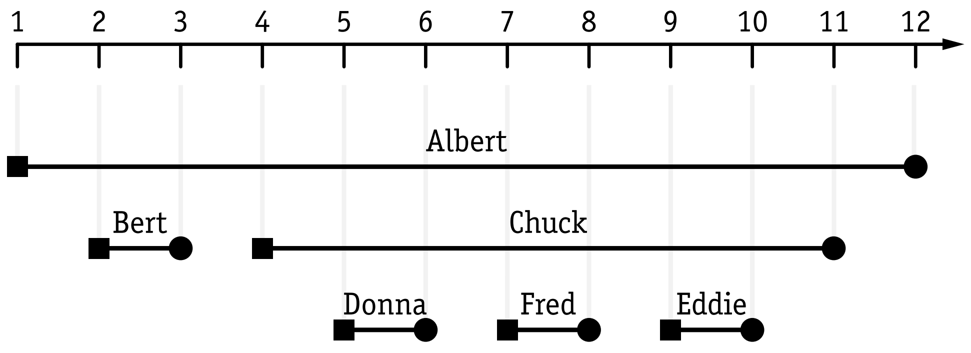


Figure 7-7: Sample ranges for the "Adjacency List Mode"

It can be seen that the range of `Albert` (1/12) overlaps the ranges of all other nodes, so `Albert` is an *ancestor* of *all* other nodes, i.e. `Albert` is the root node of the tree. The ranges of `Bert` (2/3) and `Donna` (5/6) have no common interval, so there is no ancestor/descendant relationship between them. `Bert` is a *preceding* node of `Donna` and `Donna` is a *following* of `Bert`. So the label 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 defined context node. This is exactly the same partitioning of the document like described earlier (in figure 3-9 on page 42).

Given a context node C and a 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 as follows: Table 7-2 introduces four parameters; table 7-3 describes the classification.

Symbol	**Type**
C_L	The left value of the context node
C_R	The right value of the context node
X_L	The left value of the node which is to be classified
X_R	The right value of the node which is to be classified

Table 7-2: Used symbols

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	$C_R < X_L$	
Note: The traversal algorithm guarantees that the left value of a label is smaller than its right value.		

Table 7-3: Classification by document order

With table 7-3, it is possible to determine the relationship between two nodes X and C, if the label values C_L, C_R, X_L and X_R are known. This is necessary when the tree is reconstructed from a list of nodes and their respective labels.

The Adjacency List Mode (ALM) converts a *complete* tree into a table structure and vice versa. 'Complete' tree means that all nodes from the tree are stored in the database structure. This property allows an easy labeling scheme: To assign the left values and the right values to the nodes, the counter is simply incremented by 1 in each step.

The ALM is not able to label a partial tree. For XML Pool Encryption, the plaintext document's tree is not *fully* converted into a table. Only the confidential nodes must be converted into a table structure. The public nodes remain in the public document. Section 7.5 on page 102 introduces the "Modified ALM" which overcomes this limitation.

When applying the ALM algorithm to a tree, always a full labeling traversal is performed. All label values can be calculated during that traversal. There is no need to attach the label values to the nodes in the tree using a metadata mechanism, because the tree is converted to the table on-the-fly. The tree is not altered by attaching the label values directly to the tree.

7.4.2.2 Analogy between the ALM and the event stream of an XML parser

The previous section described how the ALM algorithm labels a tree structure. An easy mapping analogy can be found between ALM label values and the serialized octets of an XML document. The intervals shown in table 7-1 on page 96 can also be found in the serialized XML instance in figure 7-8. It

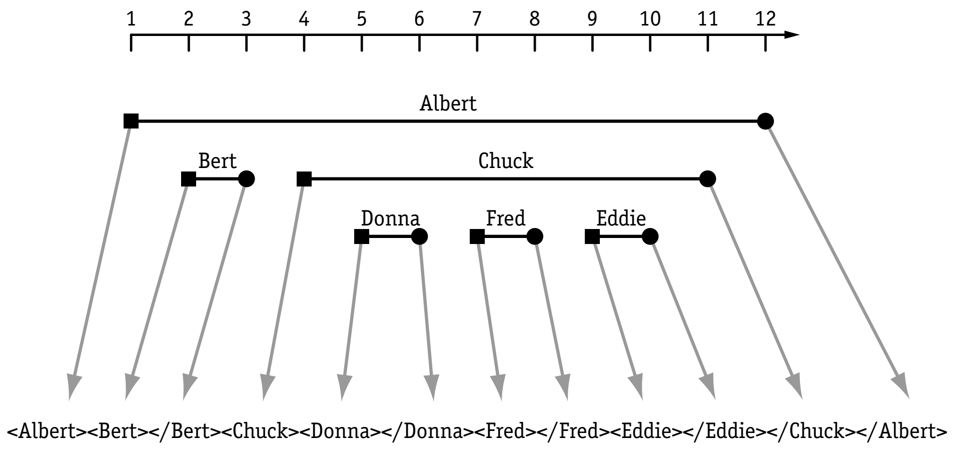

1 2 3 4 5 6 7 8 9 10 11 12
Albert
Bert
Chuck
Donna
Fred
Eddie
<Albert><Bert></Bert><Chuck><Donna></Donna><Fred></Fred><Eddie></Eddie></Chuck></Albert>

Figure 7-8: Mapping between ALM values and XML Markup

shows how the numeric label values directly correspond to the XML markup: the relative position of the start tag of an element corresponds to the left value of the node, the relative position of the end tag of an element corresponds to the right value of that node.
So instead of generating the ALM's label values with a traversal on a tree structure, the values could also be generated by a series of streaming events, e.g. using an XML parser with the SAX API [Bro02].

7.4.2.3 Storing ALM labels

For the encryption of confidential nodes of an XML document, the ALM labeling scheme has some important drawbacks: The pruning procedure removes the confidential nodes from the document, while the public nodes remain in the document.

Using the scheme to increase the counter by 1 and labeling the document, an example could look like in figure 7-9.

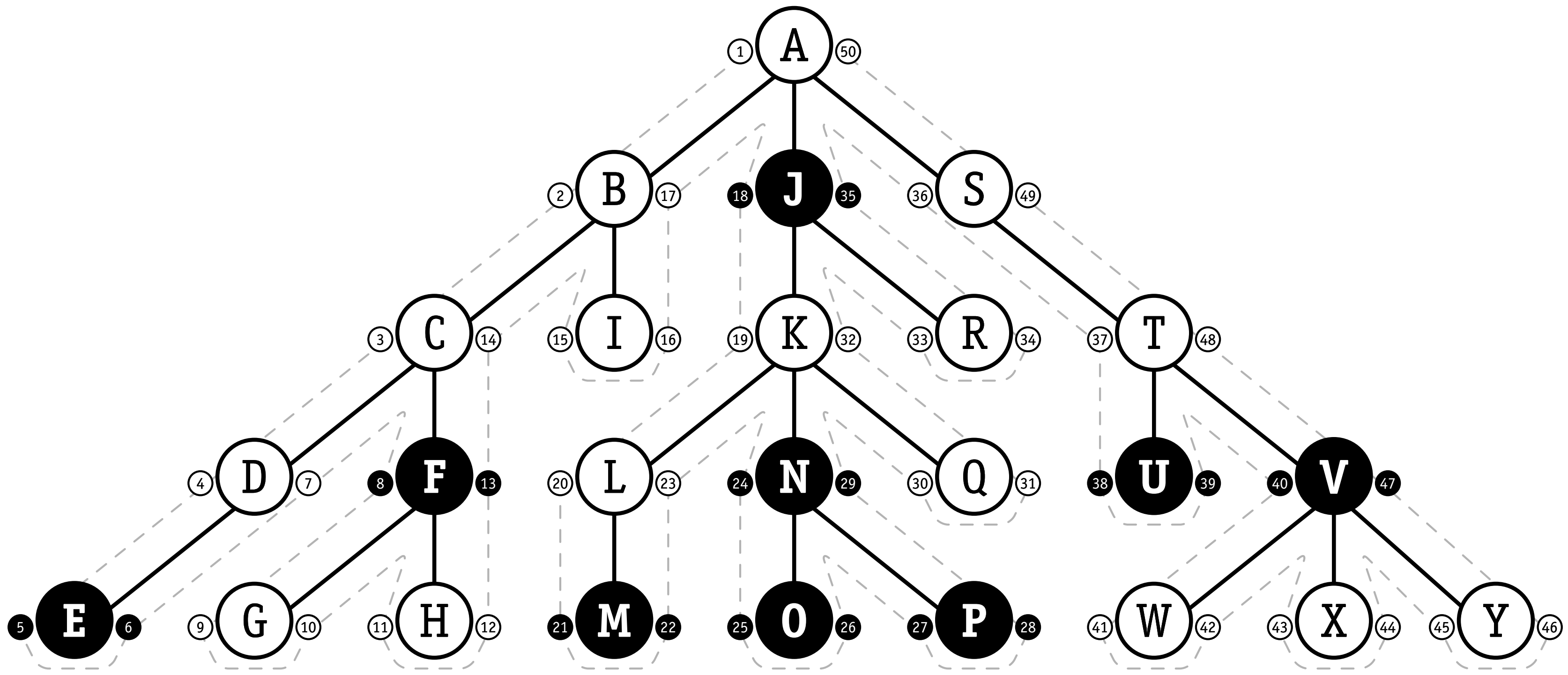


Figure 7-9: Labeled plaintext document (using ALM)

After the pruning procedure, all confidential nodes are removed document tree and the numbering scheme looks like in figure 7-10.

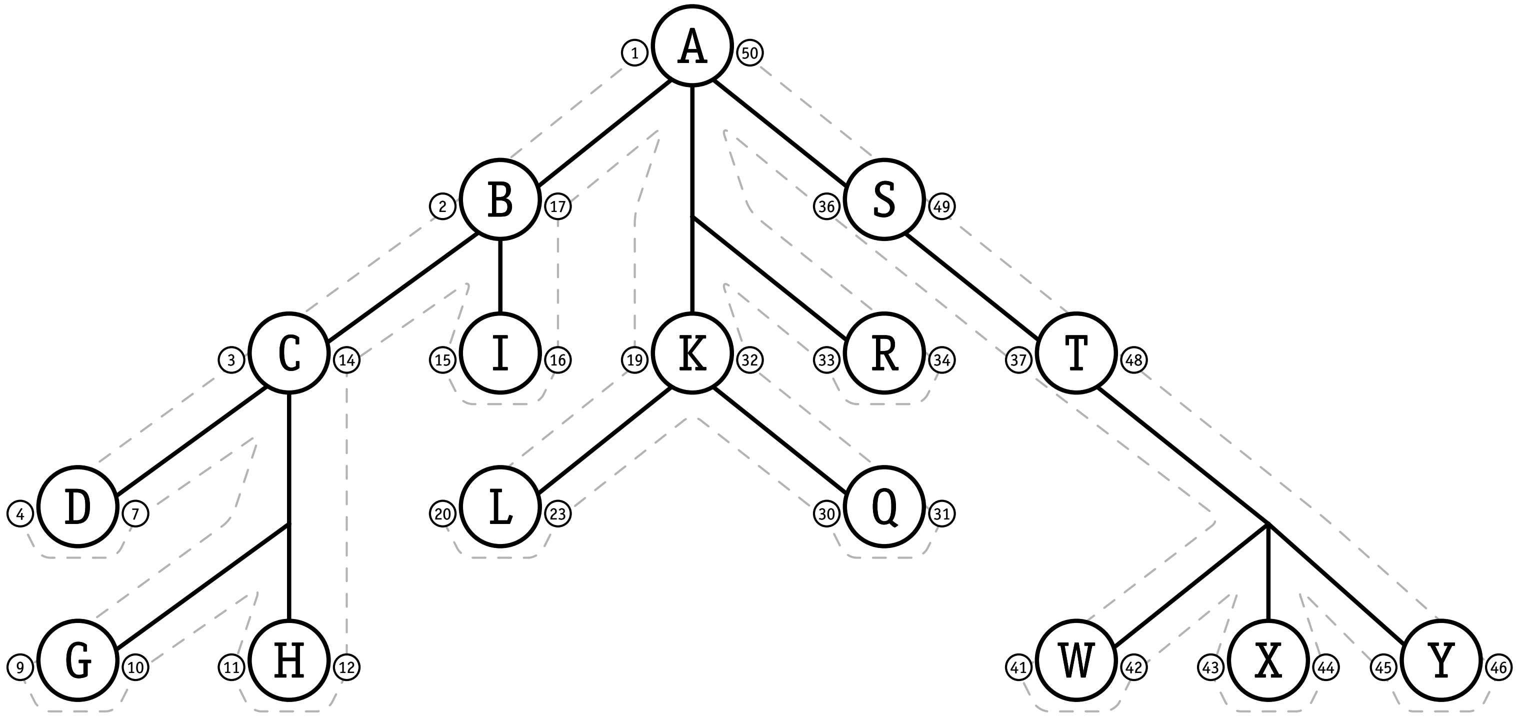


Figure 7-10: Public document with ALM label values after pruning procedure

To restore decrypted nodes in the public document, the user needs to know the label values of the public nodes. The labeling cannot be simply reconstructed by applying the labeling procedure, because the resulting labeling is not a sequence which can be reconstructed by the user. With the ALM, the labels of the public nodes would have to be transferred *into* the public document using some metadata mechanism. For an XML document, this metadata mechanism could be to add specific attributes for the left value and the right value to the document's elements (see example 7-1).

```
<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 7-1: Serialized public document with label values in attributes

Note: *The example tree in figure 7-10 consists only of elements. The XML source code adds whitespace for indentation to show the depth of a node in the tree. For simplicity, these whitespace text nodes do not show up in the figure, so strictly speaking, both figures do not match. The term 'indentation' refers to the addition of whitespace characters to source code like XML to make it more readable for a human reader.*

In the public document shown above, the left values and the right values are added to the document by using attributes in a particular namespace to store and transport these values, so that the decrypting user has access to them.

The most obvious drawback of this approach is the 'pollution' of the document's infoset with a large amount of attribute and namespace nodes; the size of the public document is increased significantly.

The second disadvantage is more subtle and relevant for the security of the system: The added attributes and the knowledge about the simple 'increase-by-one' scheme enable an attacker to make good assumptions about the plaintext document's structure. It can be spotted in which locations a confidential node has been removed (during the pruning procedure) or where no confi-

dential nodes have been as there is no space to do restore one. For instance, the Y element cannot have child nodes after decryption, because the difference between Y's right value and its left value is 1.
The last (and most obstructive) disadvantage is that only elements can be labeled by adding attributes, as only elements allow the addition of metadata using attributes. By their very nature, comments, processing instructions and character information items cannot be labeled in that way.
Therefore, the pure Adjacency List Mode is not sufficient for XML Pool Encryption.

7.5 "Modified Adjacency List Mode" (MALM)

This section presents a more advantageous way to label the tree. The novel idea is to increment the counter in larger steps using a modified label assignment algorithm, from now on called the "Modified Adjacency List Mode" (MALM):
In order to prevent the pollution of the public document with metadata, the labels cannot be attached to the public nodes. Therefore, the labels have to be attached in some way to the encrypted nodes.
The Modified ALM works similar to the original ALM. The sequence in which the nodes are visited is the same for both ALM and MALM, following the outer envelope of the tree.
In both ALM and MALM, the numerical sequence of label values (left values and right values) assigned to the nodes is a strictly monotonic increasing number sequence. That means while following the traversal at the outer envelope of a labeled tree, each value is greater than the previous visited one.
In the ALM, the difference between each label value and the previous label value is 1, because the counter for assigning label values is incremented by 1 in each step.
In the MALM, the counter X is increased in steps greater than 1. The MALM algorithm utilizes a parameter, called the *stepsize* S. S defines in which steps the counter X is incremented for public nodes.
The label values V of public nodes are always a multiple of S, i.e. $V = n \cdot S, n \in \aleph$. Between two subsequent public label values is a gap of size $S - 1$. This gap is used to hide label values which are assigned to confidential nodes.

7.5.1 A MALM example

Figure 7-11 illustrates, how the gap is used to hide label values.

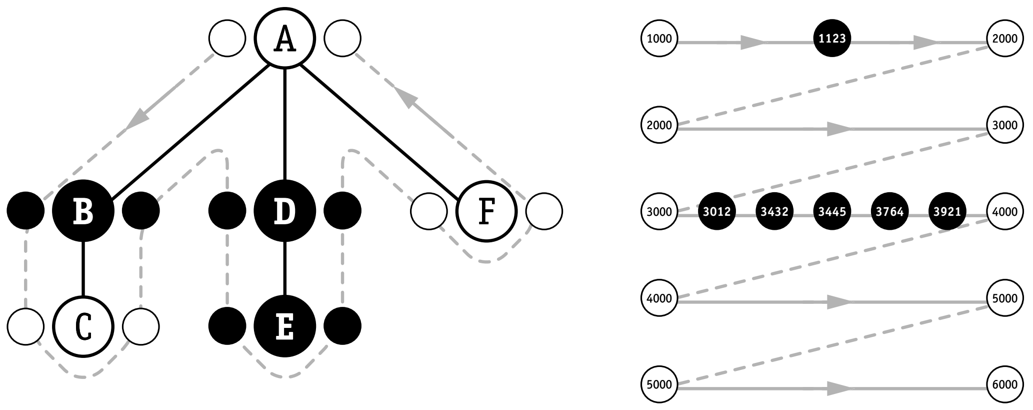


Figure 7-11: Hiding values in the gap

Note: In practice, the stepsize S will be a large integer value (Section 7.5.4 on page 109 describes how to choose S). To hide the tree’s original structure, the stepsize must be a cryptographically large value, e.g. 160 bit, i.e. $S = 2^{160}$. In this example, the stepsize is exemplarily set to $S = 1000$. The value 1000 has the advantage that the example is easier understandable, i.e. when assigning decimal values, it is obvious whether a particular value in the figures is a multiple of 1000 (assigned to a public node) or not.

Before the traversal, the counter is set to $X = 0$. The first node visited during the traversal is the public node A. The counter is increased by S, so that $X = 1000$. This value is assigned to the left value of A.
The next visited node is the confidential node B. The node B is pushed into a queue. *This queue holds all confidential nodes which have been encountered since the last label value has been assigned to a public node. This queue is flushed each time the traversal encounters a public node.*
The next visited node is the public node C. The counter is increased by S, so that $X = 2000$. The queue now contains a confidential node. The algorithm must find a label value v for that node with the property $1000 < v < 2000$. This is done by creating a random value r with the property $0 < r < S$. In the given example, the random value is chosen to be $r = 123$. Now r is added to the lower bound for v in order to lead to $v = 1000 + r = 1000 + 123 = 1123$. The value $v = 1123$ is assigned to the left value of the confidential node B. The node B is removed from the queue. After that, the queue is empty so that the tree traversal can continue.
The public node C has no child nodes, so that the traversal algorithm visits the public node C for a second time. The counter is increased by S, so that $X = 3000$.
The purpose of the queue is to store confidential nodes which have not yet been assigned a left or right label value during the traversal. Each time a pub-

lic node is visited, the queue is flushed by assigning values to the pending nodes.
The next five label values belong to confidential nodes, so that they are pushed into the queue. The sequence in which that happens is (1) B's right value, (2) D's left value, (3) E's left value, (4) E's right value and (5) D's right value. Note that the node B appears only once in the queue while the nodes D and E appear twice (they require both the left value and the right value). So the queue contains three different nodes (B, D and E), but for the time being, we denote that as "five nodes are in the queue", because D and E appear twice.
The next visited node is the public node F. The counter is increased by S, so that $X = 4000$. The queue contains the five confidential nodes mentioned earlier. The algorithm must assign five interstitial label values v_i for these nodes with the properties $3000 < v_i < 4000$ and $v_{i-1} < v_i$. This is done by creating five random values r_i with $0 < r_i < S$ and $r_i \neq r_j\big|_{i \neq j}$ and sorting these values in increasing order. In the given example, the random values are chosen to be $r_1 = 12$, $r_2 = 432$, $r_3 = 445$, $r_4 = 764$ and $r_5 = 921$. The r_i are added to the lower bound for v_i, which leads to $v_i = 3000 + r_i$.
The interstitial label values v_i are then assigned to the nodes in the queue: (1) B's right value is 3012, (2) D's left value is 3432, (3) E's left value is 3445, (4) E's right value is 3764 and (5) D's right value is assigned 3921. After that, the queue is empty so that the tree traversal can continue.
The result of the completed labeling procedure is shown in figure 7-12 .

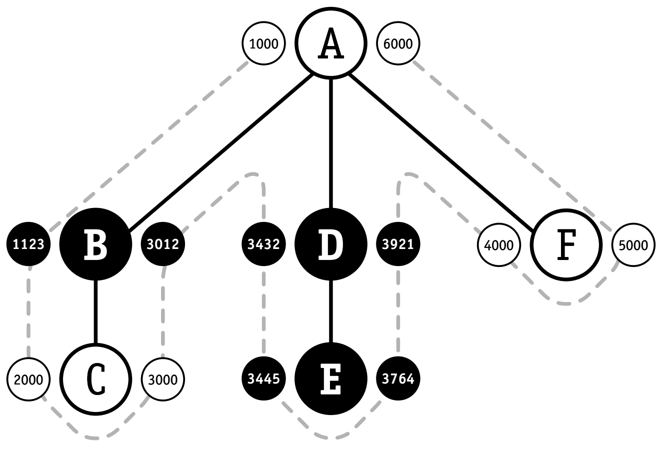


Figure 7-12: Plaintext document after labeling procedure (Modified ALM)

As describer earlier, the left and right label values which are assigned to public nodes are always a multiple of the stepsize S. Throughout this document, these label values will be called *'even label values'*. The label values assigned

to the public nodes A, C and F in the examples are (1000, 2000, 3000, 4000, 5000 and 6000).
The confidential nodes are assigned randomized label values, which lie between the even label values. These will be called *'interstitial label values'*. In the example, these interstitial label values are 1123, 3012, 3432, 3445, 3764 and 3921. The terms 'even' and 'interstitial' are defined below.
After labeling the plaintext document, the pruning procedure removes confidential nodes from the labeled plaintext document. The pruned document is called 'public document' (see figure 7-13).

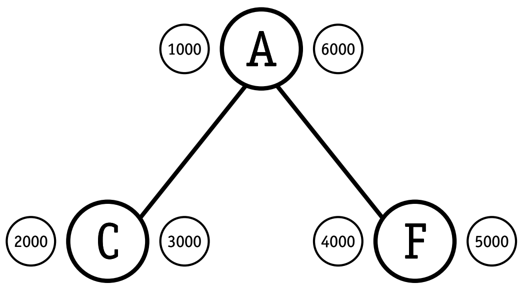


Figure 7-13: Public document after pruning procedure (Modified ALM)

7.5.2 Definitions

interstitial: The term 'interstitial' defines that the attributed item belongs to a confidential node, an encrypted node, a decrypted node or a restored node.

Note: The four terms ('confidential', 'encrypted', 'decrypted' and 'restored') all refer to the same node, while it is in different stages of the pool encryption procedure and the pool decryption procedure.
The word 'interstitial' (German „Zwischengitterplatz") is derived from crystallography: In crystallography, an interstitial is a space in the crystal's lattice where foreign atoms can be inserted, e.g. by doping a semiconductor.
The label of a confidential node is called 'interstitial label'. The interstitial label consists of two 'interstitial label values', the 'interstitial left value' and the 'interstitial right value'.

even: The term 'even' defines that the attributed item belongs to a public node.
(The term *even* is not meant as the opposite to *odd*, but refers to the regular structure of the sequence of even values, as opposite to an interstitial value.)

The label of a public node is called *'even label'*. An even label consists of two 'even label values', the 'even left value' and the 'even right value'.

The properties even and interstitial are orthogonal to left and right, i.e. each combination is possible.

interstitial sequence: A continuous sequence of interstitial label values between two subsequent even label values.

The dotted line in figure 7-14 is the path of the tree traversals. The black parts of the dotted line show the two '*interstitial sequences*'.

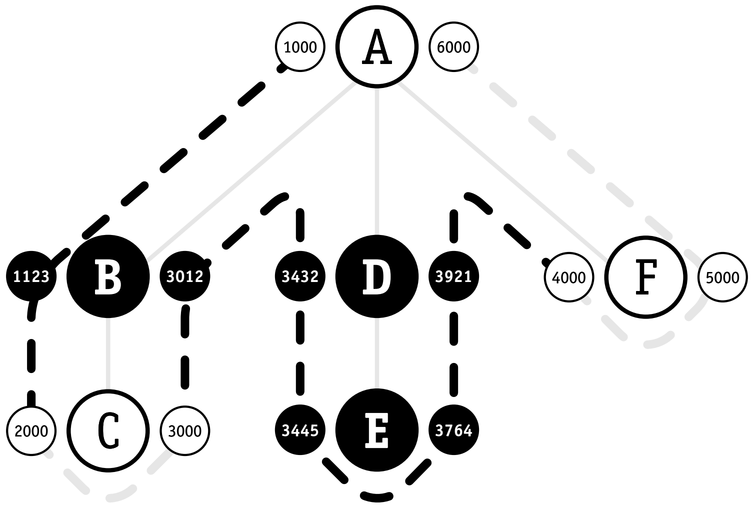


Figure 7-14: Two interstitial sequences in the tree

The first interstitial sequence contains one interstitial label value (1123) that is located in the range between the even label values 1000 and 2000. The second interstitial sequence is located in the range between the even label values 3000 and 4000 and comprises the interstitial label values (3012, 3432, 3445, 3764 and 3921). Figure 7-15 illustrates the concepts of public nodes, confidential nodes and the interstitial sequence.

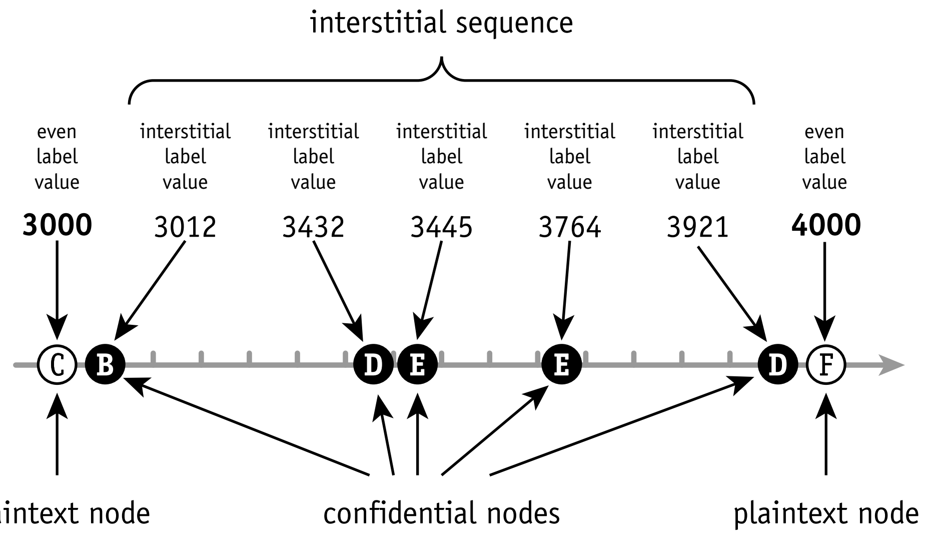


Figure 7-15: Interstitial sequence

Each interstitial label value is in a range that is delimited by two even label values. All interstitial label values of an interstitial sequence are in the same range between the same even label values. In the example in figure 7-15, it can be seen that all five interstitial label values are in the interval between 3000 and 4000. That interval is delimited by the even label values which are assigned to the public nodes C and F.

7.5.3 Interval generators

Like described earlier, each time an even label value is assigned, the queue for the interstitial label values is flushed and the interstitial label values are generated and assigned. The m interstitial label values v_i were calculated by generating m random values r_i with $0 < r_i < S$ and $r_i \neq r_j\big|_{i \neq j}$, sorting these values in increasing order so that $v_{i-1} < v_i$ and then generating the label values $v_i = S + r_i$.

An interval generator generates these m random values r_i. An interval generator receives two inputs which is (1) the stepsize S and (2) the number m of random values to be generated.

interval generator: An interval generator generates a numerical sequence in m random values. All values r_i are pairwise different from each other $r_i \neq r_j\big|_{i \neq j}$ and have the property that $0 < r_i < S$ with $0 < i \leq m$. The interval generator returns the sequence of random values

r_i sorted in increasing order.

To create m values with the above mentioned properties (pairwise different and all in the same interval), also $m < S$ is a constraint. For example, with a stepsize of $S = 10$, only 9 positive integer values r_i exist with $0 < r_i < 10$.

For XML Pool Encryption, the random number generator used to create the values r_i must be a cryptographically secure PRNG. The stepsize S is chosen to be $S = 2^n$, so that for each random value r_i to be created, the PRNG has to create n random bits.

The IntervalGenerator algorithm is described in figure 7-16. The algorithm creates m different random values with a length n-bit and returns them sorted.

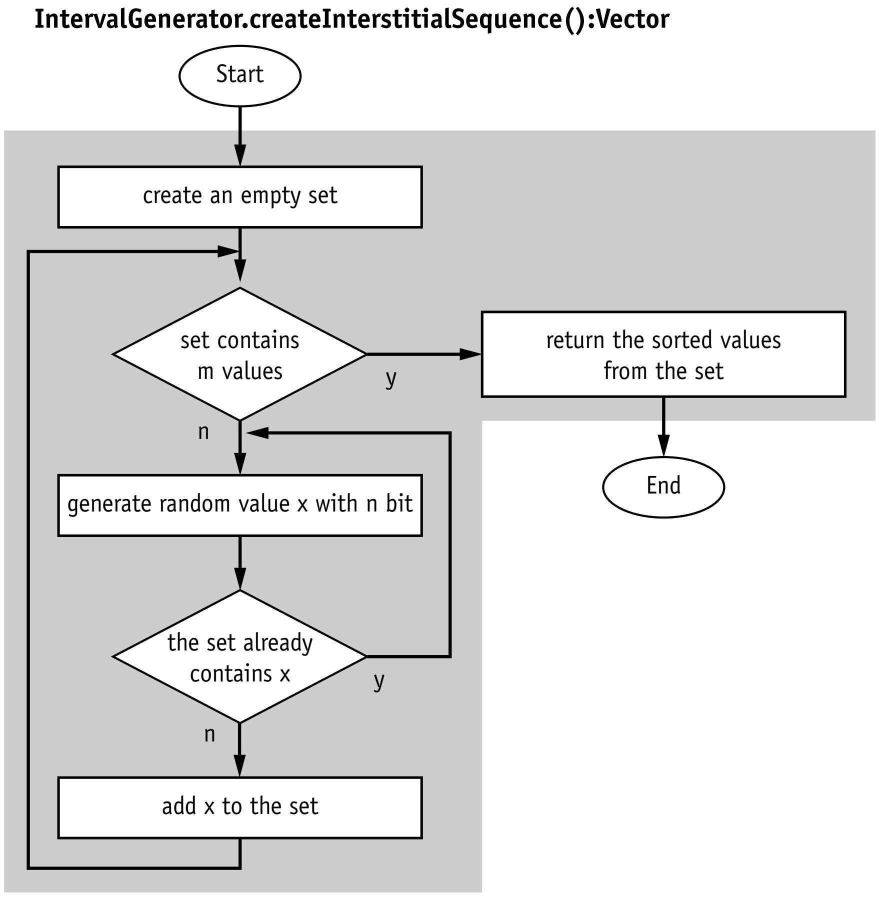

IntervalGenerator.createInterstitialSequence():Vector
Start
create an empty set
set contains m values
y
n
return the sorted values from the set
End
generate random value x with n bit
the set already contains x
y
n
add x to the set

Figure 7-16: IntervalGenerator algorithm

7.5.4 Stepsize S

The labels of the public document can be easily regenerated given the public document and the stepsize S. The stepsize S must be chosen in a way that

1. encryption and decryption are possible and
2. that it cannot be derived whether or where confidential nodes did exist between the public nodes.

The value of the stepsize S defines how many nodes 'have space' between two public nodes. With a stepsize of $S = 1$, the first even label value would be 1 and the second even label value would be 2. In order to find a interstitial label value which fits between these both, it would be necessary to find an integer x with the property $1 < x < 2$, which is not possible.

7.5.4.1 Enabling the tree labeling process

With a lower bound for the stepsize of $S = 2$, it would allow at least one single interstitial label value to fit into the interval. Generally speaking, to allow n encrypted nodes to be descendants of a public node, the stepsize must be chosen $S > 2n$.
An interval generator must generate m different random values. These values v_i must be in the given interval $0 < v_i < S$ and distinct from each other.
Each interval generator which fulfills this minimum requirement guarantees that the node restoration procedure will have *success*. The term 'success' only refers to the minimum requirement that reconstruction of the tree is possible, but not necessarily that the overall system is secure. To be secure, the dependencies between the nodes must be hidden.

7.5.4.2 Hiding dependencies between nodes

Besides the basic functionality of just being able to restore the document, the system should prevent information leakage to the user as good as possible. After decryption of the document, the user has access to the labels of all public nodes and decrypted nodes.
If the distance between two label values is too small, the user knows how many other label values could be between the two label values.

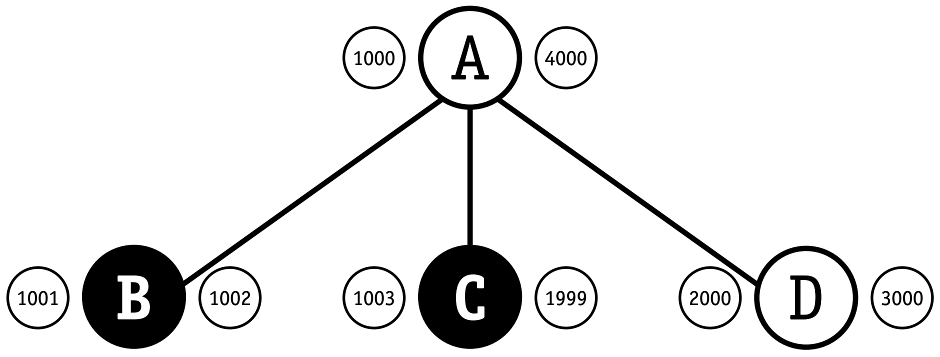


Figure 7-17: No space for interstitial nodes

Figure 7-17 shows a decrypted document, which contains two public nodes (A and D) and two decrypted nodes (B and C), along with their label values. The user knows the following about the given tree structure:

- A is the parent node of B. B is the first child node of A.
  There is no space to add an interstitial label value between the left value of A (1000) and the left value of B (1001).
- B cannot have any descendants, i.e. B is a leaf node.
  The difference between the left value of B (1001) and the right value of B (1002) does not allow insertion of any other child nodes to B.
- The direct following sibling of B is C. There are no siblings can between B and C.
  The difference between the right value of B (1002) and the left value of C (1003), does not allow insertion of any other sibling node.
- A is the parent node of C.
  All label values have been consumed by B, so there is no space to add an interstitial node.
- The next sibling of C is D.
- It is *possible* that A has ancestors.
- It is *possible* that interstitial nodes are between A and D, for instance as a child of A and parent of D.
- It is *possible* that the nodes C and D have descendants.

The interval generator must create a labeling sequence which makes it hard for users of the system to gain knowledge about the full structure of the original plaintext document.

Taken the example from figure 7-17 on page 109, the grey nodes in figure 7-18 illustrate the potential node locations. Each grey node can be a single node or a subtree.

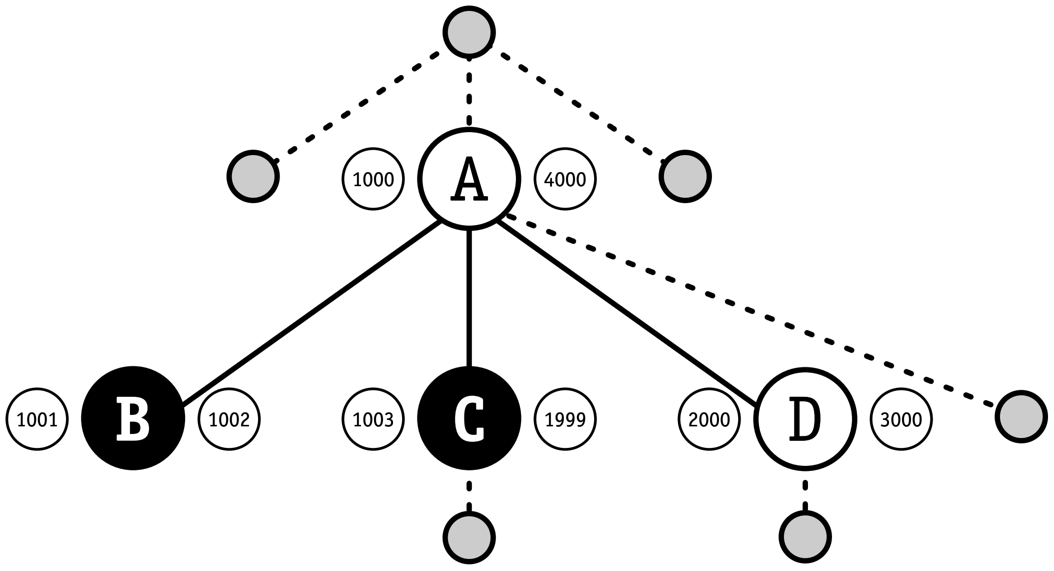


Figure 7-18: Insecurely labeled document with possible node positions

With a *secure* labeling scheme which does not leak information to the attacker, the potential node locations would be anywhere in the tree, as seen in figure 7-19.

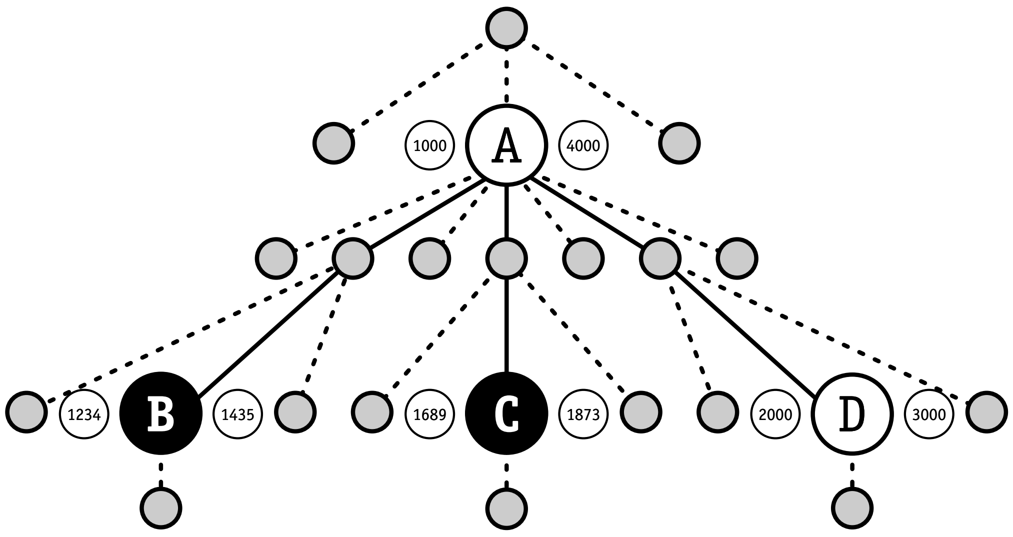


Figure 7-19: Securely labeled document with possible node positions

A user who possesses a decrypted document (which is labeled with even and interstitial label values) should not be able to use the given labeling to obtain further information about undecrypted nodes.
To achieve this unpredictability, the interstitial label values must be randomly spread across the available range of values. Additionally, the range of values must be large. Random spreading is achieved by using a cryptographically secure PRNG for generating the interstitial sequence, i.e. inside the interval generator. As discussed earlier, the range of values, i.e. the stepsize, is chosen to be a cryptographically large number.

7.5.4.3 Length of encoded labels

The labels of the confidential nodes are encrypted together with the confidential nodes' XML information set data. In order to prevent adversaries to guess information from the length of the ciphertext of an encrypted node, all labels are encoded with a fixed length.
For pool encryption, a stepsize S is chosen to be a power of 2, so that $S = 2^n$.
So a length of n bit is reserved for assigning interstitial label values in a given range. n must be sufficiently large enough to cover all possible interstitial sequences in a document.
An interstitial label value is encoded with k bits with $k = m + n$. m is the number of bits reserved for labeling the public nodes. n is the number of bits reserved for labeling the confidential nodes between the public nodes.
The bit length m must be chosen to allow the labeling of the public document, i.e. it corresponds to the total number of nodes in a given document. Typical XML documents have a size below 1 MB, so that a length of $m = 32$ is sufficient.
The bit length n (i.e. the stepsize) must be chosen to allow both the creation of an interstitial sequence and to prevent information disclosure to users. The parameter n should be chosen to be a cryptographically secure value, e.g. $n > 64$. Choosing a smaller value like $n = 12$ does *not* mean that a user could decrypt more nodes than he is permitted; it means that the user may derive parts of the original document structure information. Both values m and n are transmitted to the users in the pool of encrypted nodes. Figure 7-20 illustrates the structure of an interstitial label value.

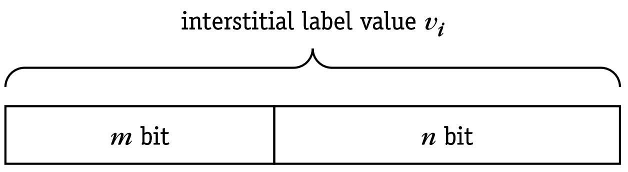


Figure 7-20: Interstitial value representation

7.6 Key Management

7.6.1 Overview

This section describes the key management process for XML Pool Encryption.

- Given a set U, containing the identities u_i of n different users, denoted by $U = \{u_1, u_2, \ldots, u_n\}$. An example is $U = \{\text{Alice, Bob}\}$.
- Given a set A of decisions for read access operations on a node, namely $A = \{\text{permit, deny}\}$
- Given an XML plaintext document's node set D, containing all document information items (both public nodes and confidential nodes). D is the result of evaluating the XPath expression "`(//. | //@* | //namespace::*)`".
- Given an access control decision function $m_{i,j} = M(u_i, d_j)$, which returns the access right for the user $u_i \in U$ on the given node $d_j \in D$, i.e. whether access is permitted or denied.
- The confidential node set C contains all confidential nodes $c_j \in D$ with $M(u_i, c_j) = \text{deny}$ for at least one user u_i.
- The public node set $P = D \backslash \mathrm{C}$ contains all public nodes, i.e. $M(u_i, p_j) = \text{permit}$ for all public node $p_j \in P$. Access to nodes in P is permitted to all users.
- For each confidential node $c_j \in C$, a unique node key k_j is created by a key generator. *Node keys* are secret keys for encryption and decryption of confidential node. Each confidential node c_j is encrypted under its node key k_j to create the ciphertext e_j:

$$e_j = \mathrm{E}_{k_j}(c_j)$$

- Each user u_i has an own confidentiality ***user key*** K_i. ***User keys*** can either be public or secret keys. Key management for user keys is outside scope of this document. It is assumed that authentic user keys are available to the DACP.
- The key collection KC_{u_i} contains all node keys k_j for which the user u_i is granted access to.

❑ The key collection KC_{u_i} is encrypted under the user's key K_i so that the user u_i is able to decrypt the encrypted nodes which he is given access to.

7.6.2 Relationship between encrypted nodes and node keys

The ciphertext of each encrypted node is marked with a non-encrypted node identifier, which identifies the node key required to decrypt the encrypted node. The node keys are also attributed with their node identifiers.

node identifier: A node identifier is an identifier which unambiguously maps a node key to the corresponding encrypted node and vice versa.

Figure 7-21 is an example for the binding between node keys and encrypted nodes via the node identifiers: The figure contains a pool of encrypted nodes and two node key collections for the users Alice and Bob.
Each node key collection is encrypted under the user's key, so that only Alice and Bob can see the respective node identifier and node keys.
After decrypting her node key collection, Alice possesses two node keys, marked with the node identifiers `"#fU5sH"` and `"#K7FFe"`. The two node identifiers provide Alice with the necessary knowledge required to select the correct two encrypted nodes from the pool of encrypted nodes for decryption.
The black surrounded parts like the node keys for Alice and Bob as well as the encrypted nodes are confidentiality protected. The encrypted nodes are protected by a symmetric cryptographic algorithm, the confidentiality protection mechanism for the node key collections is not illustrated here.
Using her node keys, Alice can decrypt the two encrypted nodes `#fU5sH` and `#K7FFe`.
In the same way, Bob can decrypt 'his' six encrypted nodes. Both can decrypt `#fU5sH` and `#K7FFe` the same way as Alice can do, but Bob can restore a larger part of the document as he also can decrypt (and restore) the nodes `#77nb4`, `#REST6`, `#x1m3L` and `#93HzG`.
The encrypted node with the node identifier `#8ZtW2` can be decrypted by neither Alice nor by Bob.

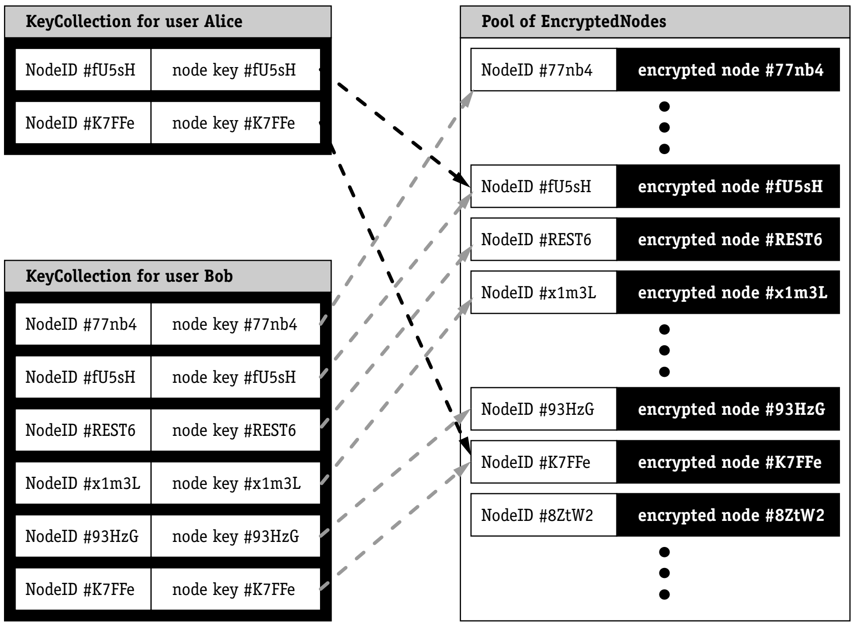


Figure 7-21: Multiple KeyCollections and the pool of encrypted nodes

An XML structure for the key collections and the pool of encrypted nodes from figure 7-21 is shown in table 7-2. In this example, the `<KeyCollection>` elements for each user contains an encrypted blob of `<NodeKey>` elements. In practice, this is a W3C XML encrypted `<xenc:CipherData>` element. In this example, the node keys are not shown encrypted in order to illustrate the mapping between the illustration and the XML structure.

```
<EncryptedPool
   xmlns="http://www.xmlsecurity.org/experimental#"
   StepSizeBits="128"
   >

  <EncryptedNodes>
    <EncryptedNode NodeID="77nb4">8sgpAqJYRb+T">qBmlxb7jlqmV...</EncryptedNode>
    <EncryptedNode NodeID="fU5sH">svljA3FHVt3BGgxOBEWCVUBEl2...</EncryptedNode>
    <EncryptedNode NodeID="REST6">gNj5pT/PYS3F4rtPCqJ4S3BGgx...</EncryptedNode>
    <EncryptedNode NodeID="x1m3L">tTkrXkAVJRJFWSgtUtRju+Ndef...</EncryptedNode>
    <EncryptedNode NodeID="93HzG">LK96e+kbsyH1qPeH7zRAcIwxQQ...</EncryptedNode>
    <EncryptedNode NodeID="K7FFe">a37dOqlnzsxBK9hdCq1BWqupPS...</EncryptedNode>
    <EncryptedNode NodeID="8ZtW2">kBJxUawVDLxUGEWtTE+l7+G3fd...</EncryptedNode>
  </EncryptedNodes>

  <KeyCollections>
    <KeyCollection UserID="Alice">
      <xenc:CipherData> <!-- encrypted under Alice's key -->
        <NodeKey NodeID="fU5sH">Xd6GdCQQR76Pz5ErO1Vpxgkks1sdfg...</NodeKey>
        <NodeKey NodeID="K7FFe">hyQCq5BIht1je1fDnf+R4zBcq5oxcv...</NodeKey>
```

Example 7-2: XML structure for KeyCollections and the pool of encrypted nodes

```
      </xenc:CipherData>
    </KeyCollection>

    <KeyCollection UserID="Bob">
      <xenc:CipherData> <!-- encrypted under Bob's key -->
        <NodeKey NodeID="77nb4">6ElRSPVk9aP2gjrPvz4gg7vplylq3d...</NodeKey>
        <NodeKey NodeID="fU5sH">Xd6GdCQQR76Pz5ErO1Vpxgkks1sdfg...</NodeKey>
        <NodeKey NodeID="REST6">bjI4dETkYQ4AVVzyHJizMJ/tgQsdfg...</NodeKey>
        <NodeKey NodeID="x1m3L">MdxBbGOYzBE0xvaiq9St+bIIyPsd<g...</NodeKey>
        <NodeKey NodeID="93HzG">UkC4l4P9rCks3u9TznBDfWgmsdfgsv...</NodeKey>
        <NodeKey NodeID="K7FFe">hyQCq5BIht1je1fDnf+R4zBcq5oxcv...</NodeKey>
      </xenc:CipherData>
    </KeyCollection>
  </KeyCollections>
</EncryptedPool>
```

Example 7-2: XML structure for KeyCollections and the pool of encrypted nodes

7.6.3 Collaboration of users

Each user who is allowed to decrypt a particular encrypted node is given the node key for that encrypted node. All node keys for a user are grouped in the KeyCollection owned by this user. A KeyCollection is mathematically a set of node keys. Two users which are allowed to decrypt an encrypted node have the same node key in their own KeyCollection.

Multiple users with (partly) disjunctive KeyCollections can collaborate to decrypt a larger part of the tree: By merging their KeyCollections, 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.

Generally, the disclosure of information by intended users is a problem that is impossible to solve. One common solution is to constrain the environment in a way that the contents can only be viewed on trusted terminals which are under physical control of the encryptor. In such an environment, guard personnel can ensure that no physical copies like handwritten papers or photographs of the confidential contents are taken. This raises the protection level to a degree that only information can be disclosed which has been remembered by the user. Generally, trusted insiders can always disclose confidential information to non-authorized parties.

7.7 XML Structure

The complete data related to XML Pool Encryption is stored at a single place inside the public document, the EncryptedPool element. The EncryptedPool element contains both the pool of encrypted nodes and the key collections for the users. The following example explains the structure of the Encrypted-Pool element (where "?" denotes zero or one occurrence; "+" denotes one or more occurrences; and "*" denotes zero or more occurrences).

```
<EncryptedPool StepSizeBits="...">
   <EncryptedNodes>
      <EncryptedNode NodeID="...">+
         Base64-encoded ciphertext
      </EncryptedNode>
   </EncryptedNodes>
   <KeyCollections>?
      <KeyCollection UserID="...">+
         <!-- This is encrypted under the user’s key -->
         <xenc:CipherData>
           <NodeKey NodeID="...">+
             Base64-encoded node key
           </NodeKey>
         </xenc:CipherData>
      </KeyCollection>
   </KeyCollections>
</EncryptedPool>
```

Figure 7-22 on page 117 is the graphical representation of that structure.

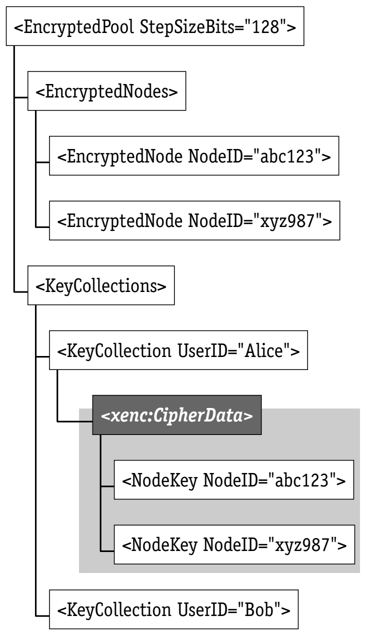


Figure 7-22: XML Pool Encryption structure

The EncryptedPool element has an attribute StepSizeBits from which the stepsize for the given public document can be calculated. For instance, if the value of this attribute is StepSizeBits="128", then the stepsize is $S = 2^{128}$. The EncryptedPool element has two child elements, the EncryptedNodes element and the KeyCollections element.
The EncryptedNodes element contains one or more EncryptedNode elements. Each EncryptedNode element has a NodeID attribute which contains the node identifier. The text inside the element itself is the base64-encoded ciphertext of the encrypted node.
The KeyCollections element contains one or more KeyCollection elements. Each KeyCollection element has a UserID attribute which helps users to find out which KeyCollection contains their node keys. Inside the KeyCollection element is a W3C XML Encryption element (xenc:CipherData) which contains the node keys, which were encrypted under the user's key.
Besides the actual node key, each NodeKey element has a NodeID attribute to link to the encrypted node.

7.8 Dummy Nodes

The number of encrypted nodes in a pool of encrypted nodes allows an attacker to determine how many confidential nodes have been removed from the plaintext document. Given the security service *prevention of traffic flow analysis* (see also "Security mechanisms for traffic flow confidentiality" on page 16), a similar service can be defined for encrypted trees: An attacker should not be able to gain knowledge about how many confidential nodes have been in the plaintext document. It should be hidden from both legitimate users and attackers how the original structure had been. A legitimate user who is given all node keys can decrypt and reconstruct the full document, but he should 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* node key.
2. Attackers and users with access to a *reduced set* of node keys.
3. Users with access to the *all* node keys.

An attacker *without any node key* has only access to the public document which contains public nodes. The attacker has access to the pool of encrypted nodes. From this pool of encrypted nodes, the attacker can count how many encrypted nodes exist. The stepsize allows to calculate how many confidential nodes can exist in the given public document.
For a cryptographically strong value for the stepsize, the size of a pool of encrypted nodes is multiple orders smaller than the possible number of confidential nodes.
An attacker or user with access to a *reduced set* of node keys can decrypt some encrypted nodes and therefore reconstruct parts of the document. After

the decryption, the attacker can count how many encrypted nodes remain undecrypted. In contrast to the previous attacker, he knows *some* interstitial label values, so he can make a better assumption on how many nodes have space in particular areas of the tree.
For a cryptographically strong value for the stepsize, the number of decrypted nodes size is multiple orders smaller than the possible number of confidential nodes.
A user with full access to *all* node keys can fully reconstruct the original plaintext document. Is such a user an attacker? It seems that this user already has access to all nodes, but that is not the case: It can be hidden from this user *that* he has already decrypted all nodes.
The interstitial label values assigned to confidential nodes are randomly chosen. It can be prevented that the user gets this assurance, if the stepsize is chosen to be cryptographically strong (e.g. 64 bit) and if the DACP adds *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 security service. The node key required to decrypt a dummy node is not given to any user, therefore all users must assume that the dummy node contains a confidential node which they are not allowed to see. Only the DACP is able to distinguish between encrypted nodes and dummy nodes.

7.9 Syntax for the algorithms

The following sections provide a detailed description of the algorithms for node selection, pool encryption and pool decryption. The algorithms are described using a textual and graphical algorithm description. The textual description highlights the general idea of the procedures while the graphical representation contains details that are more specific.
In order to keep the descriptions and the flowcharts short and precise, at some places object-oriented, Java-like statements are used. Many of the operations work on nodes in a tree, more specifically on XML nodes in a DOM tree. For manipulating these nodes or retrieving properties from these nodes, standard DOM method calls are used on these nodes. For example, the check whether a specific node called `xNode` has children will be represented by the `xNode.hasChildren()` DOM method, both in the textual and the graphical descriptions.

7.10 Node selection procedure

The XML Pool Encryption model introduces two classes of entities: DACPs (*'document access control service'*) and users (see page 85). A DACP encrypts an XML document. A DACP may act on behalf of a user. One or more users decrypt the document (or portions thereof). This section formally describes the processes which are carried out to encrypt and decrypt the document.

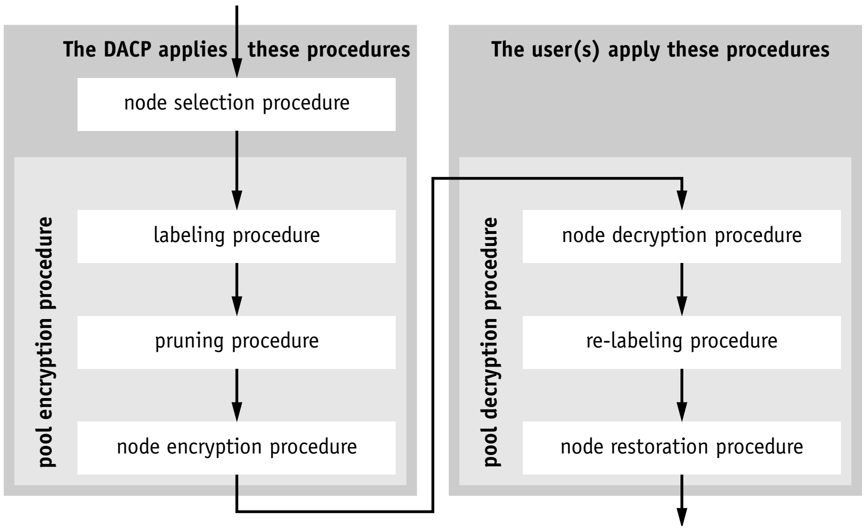

The DACP applies these procedures
node selection procedure
pool encryption procedure
labeling procedure
pruning procedure
node encryption procedure
The user(s) apply these procedures
pool decryption procedure
node decryption procedure
re-labeling procedure
node restoration procedure

Figure 7-23: XML Pool Encryption procedures

7.10.1 Overview

The DACP transforms a plaintext document into a public document which contains encrypted nodes. The intent of the DACP is to protect confidential portions of that document so that only authorized users can read them. The DACP classifies the plaintext document's contents (the nodes) by given confidentiality requirements, the ConfidentialitySpecification.
The ConfidentialitySpecification contains a list of all confidential nodes and describes which users are permitted to see which of the confidential nodes.
Nodes which are not confidential are called public nodes. The public nodes can be read by any user who possesses the public document, regardless whether the user possesses node keys or not. The confidential nodes are only accessible to users who possess the matching node key.

7.10.2 Algorithm

The node selection procedure receives two input arguments:

❏ the plaintext document plaintextDocument

❏ a ConfidentialitySpecification which maps users to confidential nodes.

All nodes which are confidential as defined by the ConfidentialitySpecification are selected by the node selection procedure. After the node selection procedure, a set called setOfConfidentialNodes exists that is an unordered collection of the plaintext document's confidential nodes. The setOfConfidential-Nodes contains numberOfConfidentialNodes confidential nodes.

The node selection procedure returns

- ❑ the setOfConfidentialNodes.

7.10.3 Example

Based on figure 7-9 on page 99, an example for a ConfidentialitySpecification could look like in table 7-4:

User	XPath expression to select the confidential nodes
Alice	(//E | //F | //J | //M | //N | //P | //U | //V)
Bob	(//N | //O | //P)

Table 7-4: Example for a ConfidentialitySpecification

The above ConfidentialitySpecification defines that the user Alice is allowed to decrypt the nodes E, F, J, M, N, P, U and V. The user Bob is only allowed to decrypt the smaller subset N, P and the node O. Alice can decrypt 8 nodes, Bob can decrypt 3 nodes. The union of these both node sets contains 9 nodes:

setOfConfidentialNodes = (//E | //F | //J | //M | //N | //O | //P | //U | //V)

All other nodes in the document are public nodes.

7.11 Pool encryption procedure

The pool encryption procedure receives following input arguments:

- ❑ the plaintextDocument,
- ❑ the stepSize parameter,
- ❑ the ConfidentialitySpecification with the setOfConfidentialNodes,
- ❑ the encryption algorithm required to encrypt a node and
- ❑ a cryptographically secure randomBitGenerator.

The pool encryption procedure performs the following steps:

1. the labeling procedure is executed,
2. the pruning procedure is executed,
3. the node encryption procedure is executed,

The pool encryption procedure produces the following results:

- ❏ the encryptedDocument,
- ❏ the poolOfEncryptedNodes and
- ❏ for each user a KeyCollection.

7.11.1 Labelling procedure

7.11.1.1 Overview

The labeling procedure applies the Modified Adjacency List Mode (MALM) to the plaintext document. The description of this procedure and examples are found earlier in this chapter, beginning with ""Modified Adjacency List Mode" (MALM)" on page 102.

7.11.1.2 Algorithm

The labeling procedure receives following input arguments:

- ❏ the plaintextDocument,
- ❏ the stepSize parameter,
- ❏ the setOfConfidentialNodes and
- ❏ a cryptographically secure randomBitGenerator.

The labeling procedure produces the following result:

- ❏ The getLabelByNode map for all nodes in the plaintext document. This map is a mapping structure which returns the label which belongs to a given node. ("What label values has node X?")

The labeling procedure is performed by a TreeLabeler, which provides two methods:

- ❏ TreeLabeler.process traverses the complete document tree (figure 7-24 on page 123) and
- ❏ TreeLabeler.label which assignes a label to a node (figure 7-25 on page 124).

TreeLabeler.process. The TreeLabeler.process method

1. assigns the left value to the current node X,
2. recursively calls TreeLabeler.process on every child of the current node X if X has children and
3. assigns the right value to the current node.

Note that TreeLabeler.process is a recursive method, as it performs the tree traversal in a recursive way. For that reason, the program flow diagram explicitly shows that recursion.

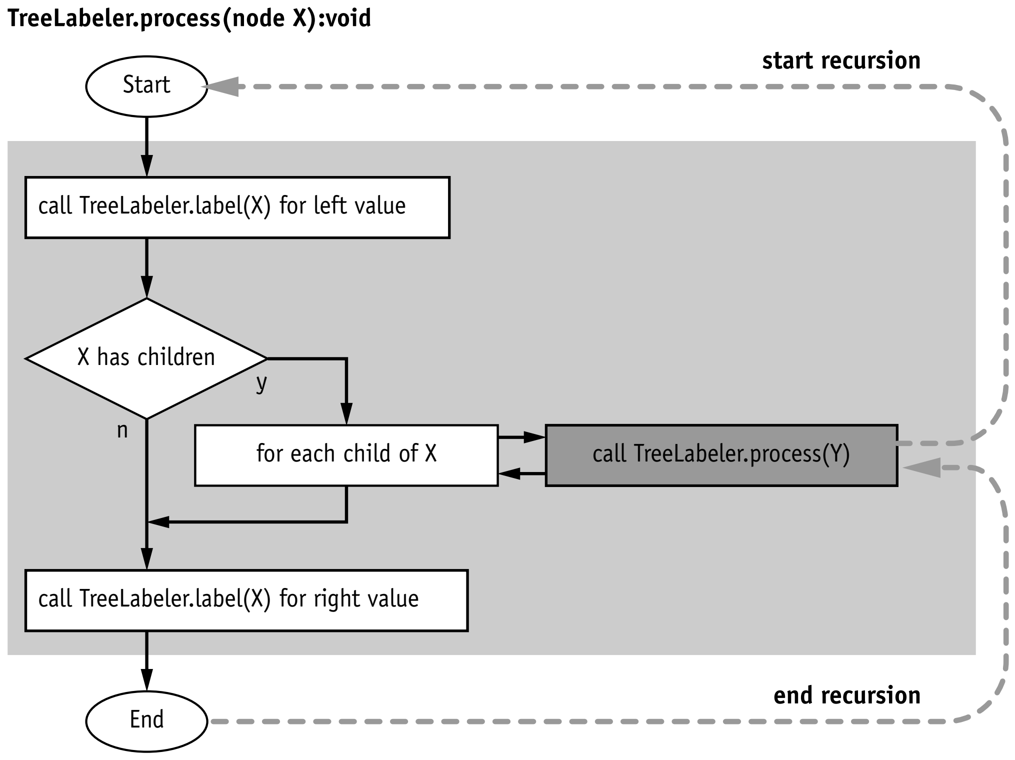

TreeLabeler.process(node X):void
Start
start recursion
call TreeLabeler.label(X) for left value
X has children
y
n
for each child of X
call TreeLabeler.process(Y)
call TreeLabeler.label(X) for right value
end recursion
End

Figure 7-24: TreeLabeler.process algorithm

TreeLabeler.label. The TreeLabeler.label method assigns left values and right values to a node's label.

1. If the current node is a confidential node, the node is pushed into the queue.
   When encountering a confidential node during the tree traversal, it is not yet clear whether other confidential nodes directly follow the current one. Therefore, all confidential nodes are stored in the queue.
2. If the current node is a public node, it is checked whether the queue contains confidential nodes.
3. If the queue contains confidential nodes, the ValueGenerator creates one interstitial label value for each confidential node. Then these interstitial label values are assigned to the labels of the confidential nodes in the queue and the queue is emptied.
4. The current even label value is assigned the to the label of the current node.

The inner structure of the TreeLabeler.label() method is shown in figure 7-25 on page 124.

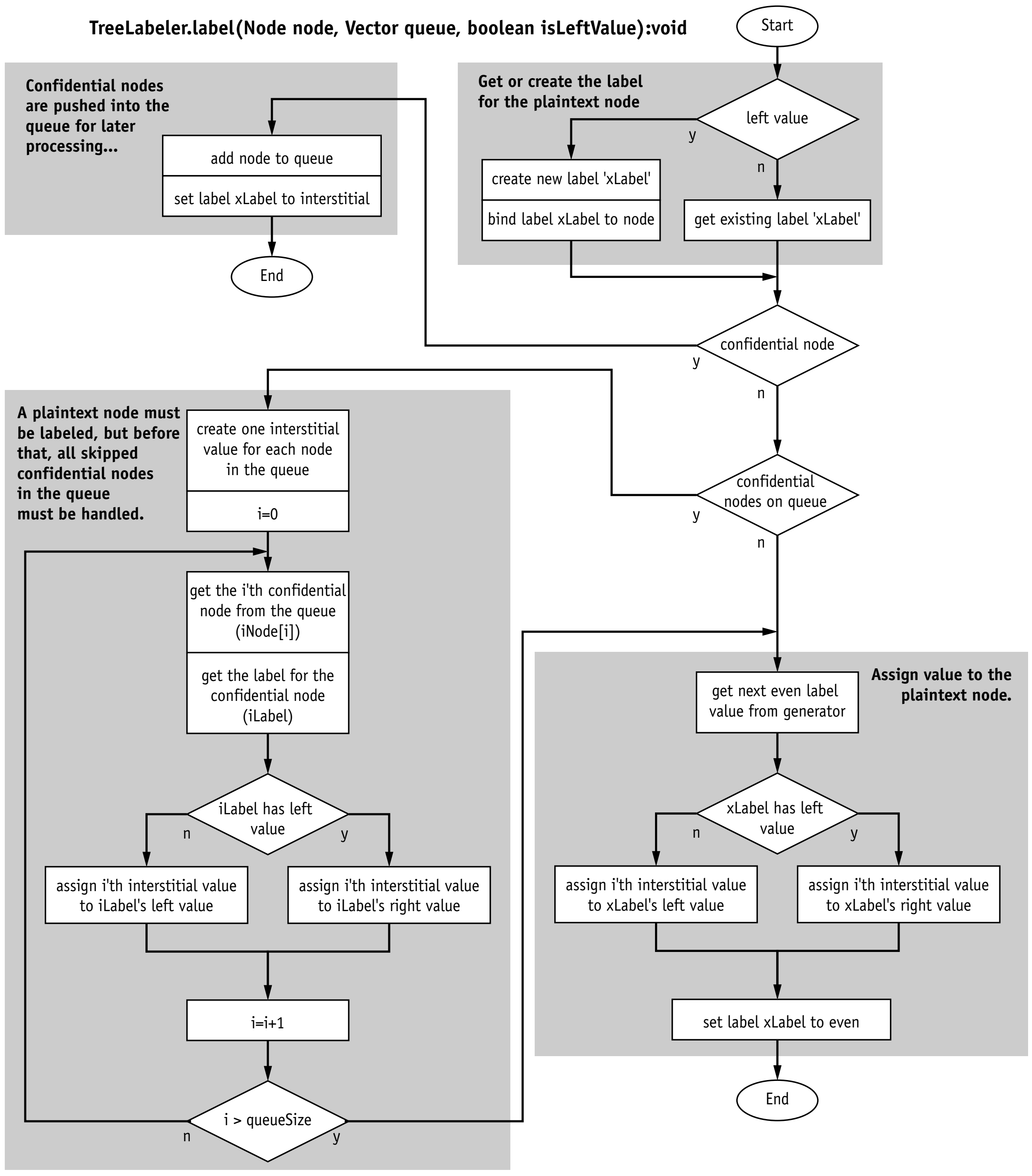

TreeLabeler.label(Node node, Vector queue, boolean isLeftValue):void
Start
Get or create the label for the plaintext node
left value
y
n
create new label 'xLabel'
bind label xLabel to node
get existing label 'xLabel'
Confidential nodes are pushed into the queue for later processing...
add node to queue
set label xLabel to interstitial
End
confidential node
y
n
confidential nodes on queue
y
n
A plaintext node must be labeled, but before that, all skipped confidential nodes in the queue must be handled.
create one interstitial value for each node in the queue
i=0
get the i'th confidential node from the queue (iNode[i])
get the label for the confidential node (iLabel)
iLabel has left value
n
y
assign i'th interstitial value to iLabel's left value
assign i'th interstitial value to iLabel's right value
i=i+1
i > queueSize
n
y
Assign value to the plaintext node.
get next even label value from generator
xLabel has left value
n
y
assign i'th interstitial value to xLabel's left value
assign i'th interstitial value to xLabel's right value
set label xLabel to even
End

Figure 7-25: TreeLabeler.label algorithm

7.11.2 Pruning procedure

7.11.2.1 Overview

During the pruning procedure, all confidential nodes are removed from the plaintext document (figure 7-26).

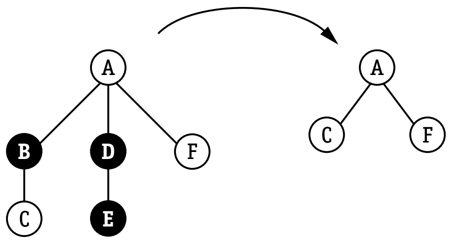


Figure 7-26: Pruning the tree

7.11.2.2 Algorithm

For each confidential node in the setOfConfidentialNodes, the pruning procedure replaces the confidential node by its child nodes. This is done by executing the fosterChildrenToGrandparents algorithm on each confidential node. Figure 7-27 shows an example for removing the confidential node B from the document:

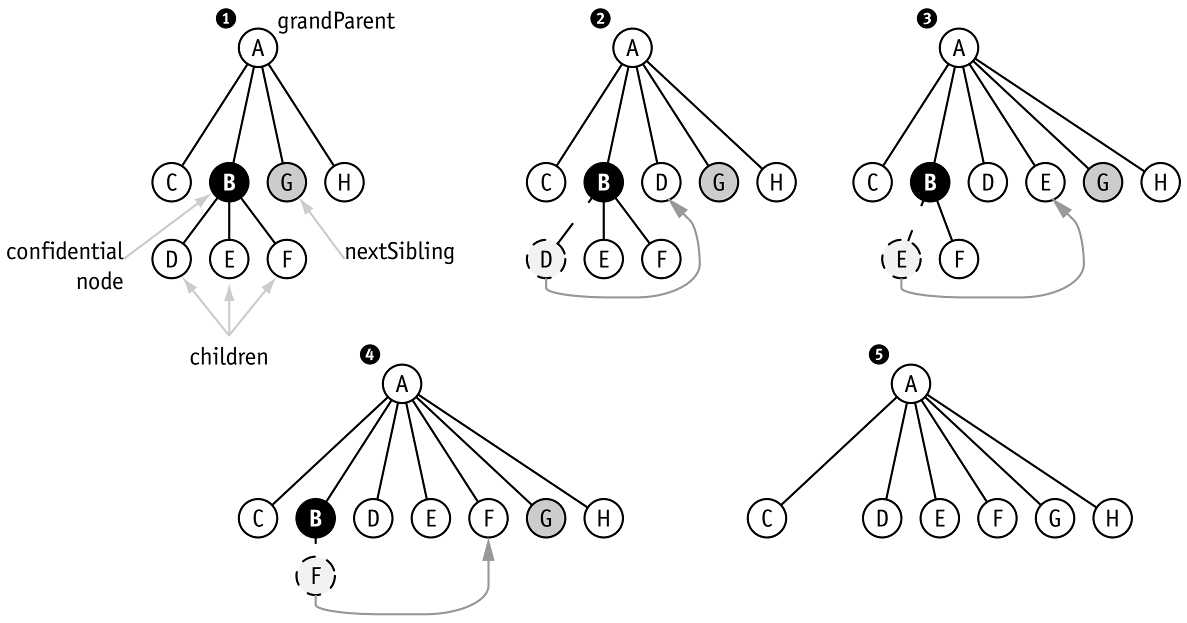


Figure 7-27: fosterChildrenToGrandparents example

In figure 7-27/1, there is the confidential node B which has three child nodes (D, E and F). The next sibling of B is the node G. In the subsequent steps (7-27/2 to 7-27/4), the first child of B is removed from B's children and inserted before G as a child of A. After fostering the children (D, E and F) to

their grandparent, the confidential node B is removed from the document (7-27/5).
The algorithm is shown in figure 7-28 below:

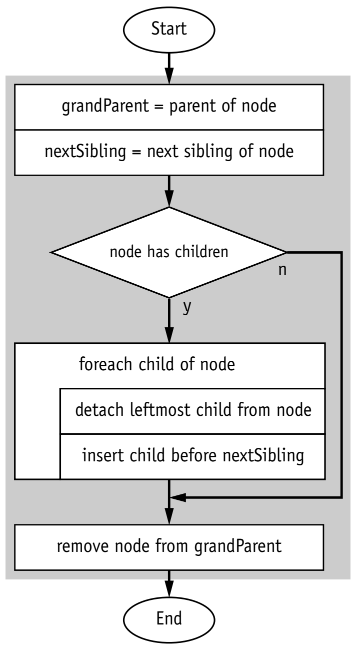

Start
grandParent = parent of node
nextSibling = next sibling of node
node has children
n
y
foreach child of node
detach leftmost child from node
insert child before nextSibling
remove node from grandParent
End

Figure 7-28: fosterChildrenToGrandparents algorithm

After the pruning procedure, the public document contains only the public nodes. The labels of the public nodes are all even labels, i.e. the label values of the even label's are a multiple of the stepsize S. As the result, the complete labeling of the public document can be reconstructed if only the stepsize S is known.
The pruning procedure receives following input arguments:

- the plaintextDocument and,
- the setOfConfidentialNodes.

7.11.3 Node encryption procedure

7.11.3.1 Overview

After the pruning procedure, the removed confidential nodes are encrypted by the node encryption procedure. Each confidential node and its associated label are encrypted under a unique node key. This step produces the encrypted nodes which are collected in a pool of encrypted nodes. The DACP must generate one unique node key for each confidential node.

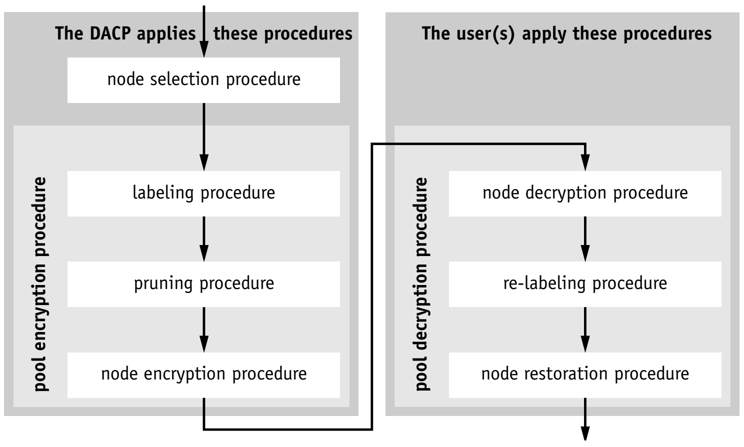
7.11.3.2 Algorithm

The plaintext document contains now labeled confidential nodes which have to be encrypted. For each of these confidential nodes,

1. the DACP generates a unique bit sequence using a pseudo-random bit generator. This bit sequence is base64-encoded into a text string. This text string is called currentNodeKeyIdentifier.
2. The DACP generates a unique node key called currentNodeKey.
3. The label of the confidential node, called currentConfidentialNodeLabel, is concatenated with the confidential node's plaintext data. This is the input octets for the encryption algorithm and is called currentPlaintext.
4. The currentPlaintext is encrypted using the currentNodeKey. The encrypted result is called currentCiphertext.
5. The currentCiphertext is base64-encoded and forms the `<EncryptedNode>` element's content. The currentNodeKeyIdentifier is added to the `<EncryptedNode>` element as an attribute.

Using this process, all confidential nodes are encrypted. The `<Encrypted-Node>` elements are all grouped in a single `<PoolOfEncryptedNodes>` element. Then each currentNodeKey is combined with it's currentNodeKeyIdentifier as already shown in example 7-2 on page 115.

7.12 Pool decryption procedure



The DACP applies these procedures
node selection procedure
pool encryption procedure
labeling procedure
pruning procedure
node encryption procedure
The user(s) apply these procedures
pool decryption procedure
node decryption procedure
re-labeling procedure
node restoration procedure

The pool decryption procedure receives following input arguments:

- ❏ the publicDocument,
- ❏ the stepSize parameter (that is stored inside the publicDocument),
- ❏ the `<PoolOfEncryptedNode>` element (that is stored inside the publicDocument),
- ❏ the encryption algorithm required to decrypt the nodes (that is stored inside the publicDocument) and

- ❑ a set of currentNodeKeys and their associated currentNodeKeyIdentifiers (which are available after the user decrypted his KeyCollection)

The pool decryption procedure performs the following steps:

1. the node decryption procedure is executed (which results in the decrypted nodes and their labels),
2. the labeling procedure is executed (which assigns labels to the public nodes),
3. the node restoration procedure is executed (to restore the decrypted nodes in the document).

The pool decryption procedure produces the following result:

- ❑ the decryptedDocument.

7.12.1 Node decryption procedure

7.12.1.1 Overview

To perform the node decryption procedure, a user must possess

- ❑ the public document (including the pool of encrypted nodes and the node key collections for the different users) and
- ❑ the key to decrypt his own KeyCollection, in which the node keys for 'his' encrypted nodes.

Each node key has a node identifier which serves as a unique link between the encrypted node and the node key and vice versa. The encrypted node is decrypted using the node key. The result of this decryption process is the node's plaintext data and its label.
The node decryption procedure decrypts all encrypted nodes for which the user possesses a node key.

7.12.1.2 Algorithm

1. Locate the `<KeyCollection>` element inside the public document that is intended for the user (based on the `UserID` attribute).
2. Decrypt the content of the `<KeyCollection>` element using the user's key. This decryption returns the `<NodeKey NodeID="..">` elements, i.e. the node keys and the associated node identifiers which had been encrypted for that user.
3. For each node key, decrypt the corresponding encrypted node using the node key. That produces the set of decrypted nodes, i.e. nodes' plaintext data and their labels.
   From the node's plaintext data, the node's XML structure is reconstructed.

After this operation, the user possesses a set of labeled decrypted nodes.

7.12.2 Node restoration procedure

7.12.2.1 Overview

Before the decrypted nodes can be restored, the user must re-label the public document. The `<EncryptedPool>` element contains the stepsize parameter, so that the public document can be labeled using the MALM algorithm. After that operation, the user possesses a labeled public document. The node restoration procedure takes this labeled public document and the decrypted nodes as input and inserts the decrypted nodes into the appropriate positions.

7.12.2.2 Algorithms for the node restoration

The node restoration procedure requires three functions:

- ❑ The restoreNode function does the restoration of the decrypted node.
- ❑ The getNearestAncestor function is a helper function for the restoreNode function. It is necessary for locating the parent of a decrypted node.
- ❑ The parentalizeOrphan function supports restoreNode with properly reconnecting child nodes of a decrypted node.

The restoreNode function inserts the decrypted node into the public document. It requires that the public document is labeled and that the label of the decrypted node is known. In order to perform this task, the restoreNode function has to

1. identify which node inside the decrypted document is the parent node of the decrypted node (done by the getNearestAncestor function) and
2. if that parent node already has child nodes,
   - ❍ determine where exactly in the sequence of existing child nodes the new node must be prepended, inserted between or appended and
   - ❍ determine whether some of the decrypted node's siblings are not siblings but must be turned into children of the decrypted node, and if so, which ones (both done by the parentalizeOrphan function).

Before describing the details of the restoreNode function, the algorithms for the getNearestAncestor function and the parentalizeOrphan function are shown.

7.12.2.2.1 getNearestAncestor algorithm

The getNearestAncestor algorithm is a helper algorithm for the restoreNode. getNearestAncestor determines the nearest ancestor for the decrypted node that is to be restored. The purpose is that the result of getNearestAncestor serves as a parent node for the decrypted node that is re-inserted into the document.

The algorithm works as follows:

- ❏ A variable bestMatch stores the nearest ancestor. At the beginning, bestMatch is initialized with the symbolic value 'NOT_FOUND'.
  bestMatch always contains either the 'NOT_FOUND' value or an ancestor of the decrypted node.
- ❏ Each node in the document must be 'inspected', i.e. it must be checked whether the node in question is the nearest existing ancestor for the decrypted node.
  1. Each node which is *not* an ancestor of the decrypted node is skipped (this ensures that bestMatch always contains an ancestor).
  2. If the node in question is a descendant of the bestMatch, the value of bestMatch is overwritten with the node.
     The node is an ancestor (this was ensured in step 1), and if the node is a descendant of bestMatch, this means that the node is a nearer ancestor than bestMatch.
- ❏ After inspecting each node in the document, the bestMatch is returned.

The algorithm is illustrated in figure 7-29 on page 131.

Both getNearestAncestor and restoreNode must be able to decide whether a given node from the document is an ancestor, a descendant, or a following sibling of the decrypted node. The obvious problem is that the decrypted node is not in the document, so that the labels must be used.

To determine the relationship between two nodes which can only be inspected based on their labels, the relationships from table 7-3 on page 97 are used.

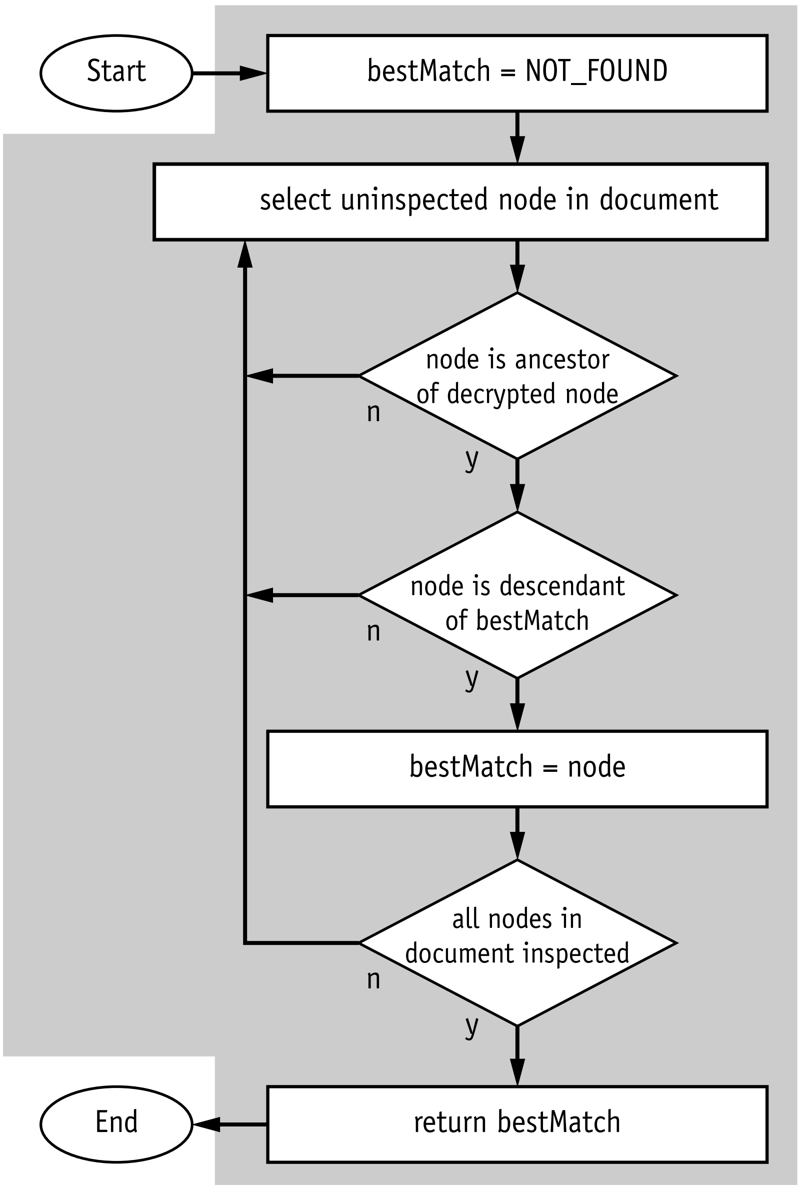

Start
bestMatch = NOT_FOUND
select uninspected node in document
node is ancestor of decrypted node
n
y
node is descendant of bestMatch
n
y
bestMatch = node
all nodes in document inspected
n
y
End
return bestMatch

Figure 7-29: getNearestAncestor algorithm

7.12.2.2.2 parentalizeOrphan algorithm

The parentalizeOrphan algorithm is a helper algorithm for the restoreNode algorithm. The parentalizeOrphan algorithm is used when adding a decrypted node to a parent which already has children from which some must become children of the decrypted node.
parentalizeOrphan adds the decrypted node to the children of its parent, determines all children of the decrypted node and moves them into the right place.

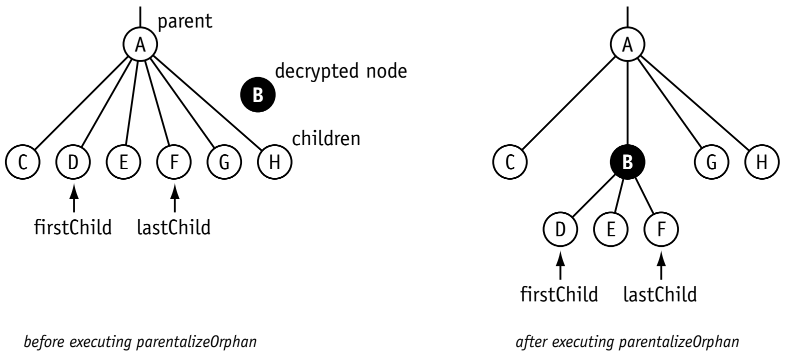


Figure 7-30: parentalizeOrphan example

In the example in figure 7-30, the decrypted node B must be added to its parent node A. The restoreNode function (described in the next section) determines that A's child nodes D, E and F must be made children of B. restoreNode determines the first and the last child of A's children which must be relocated into B. B must be put into the place which was previously occupied by D, E and F.
Before executing parentalizeOrphan, the next sibling of the node C is D.
After executing parentalizeOrphan, the next sibling of the node C is B.

Before executing parentalizeOrphan, the previous sibling of G is F.
After executing parentalizeOrphan, the previous sibling of G is B.

The parentalizeOrphan algorithm works as follows:

- ❑ The decrypted node is inserted after lastChild.
- ❑ All nodes between firstChild and lastChild are relocated into the decrypted node.

Figure 7-31 shows this process in more detail:

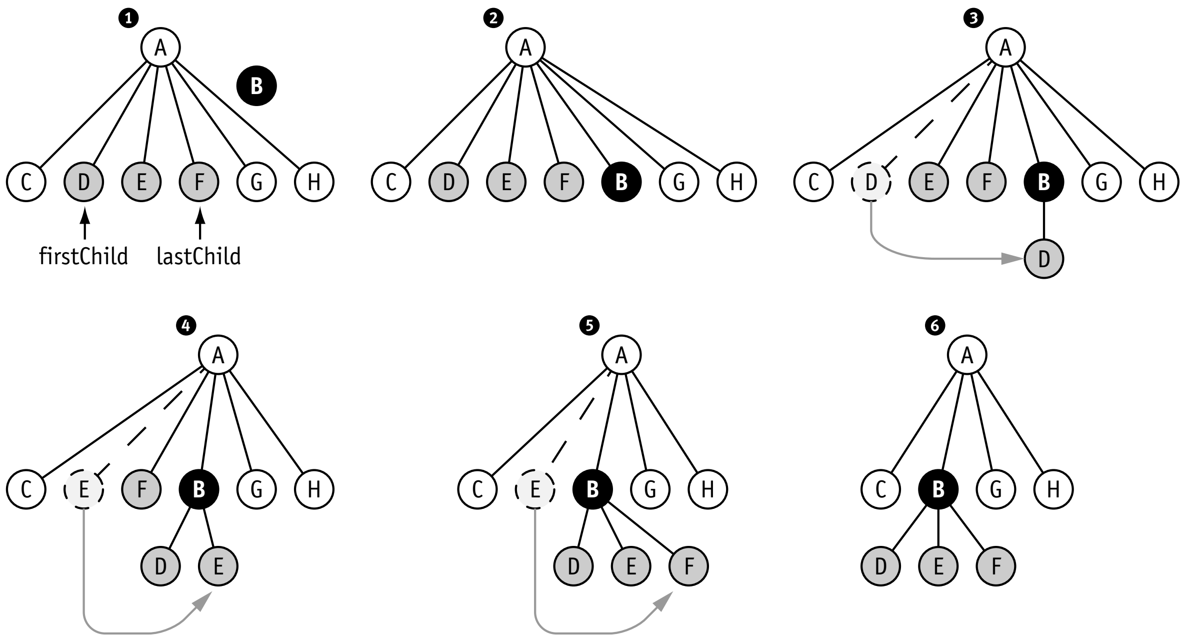
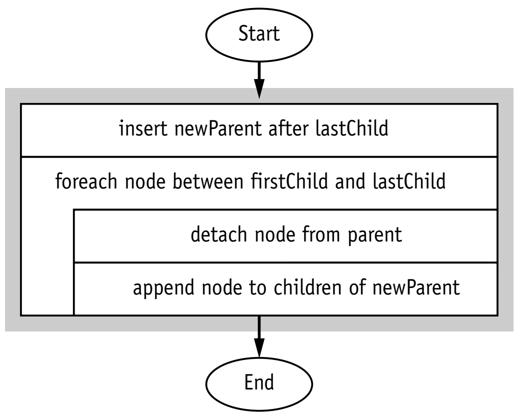


Figure 7-31: Detailed parentalizeOrphan example

When parentalizeOrphan is called, the node B is not yet inserted (7-31/1). B is inserted after the lastChild F (7-31/2). Beginning with node D (7-31/3), all nodes which must be made children of B are moved into their new position (7-31/4 and 7-31/5). The result of that process is shown in figure 7-31/6.



Start
insert newParent after lastChild
foreach node between firstChild and lastChild
detach node from parent
append node to children of newParent
End

Figure 7-32: parentalizeOrphan algorithm

7.12.2.2.3 restoreNode algorithm

Figure 7-33 shows the restoreNode function which does restoration.

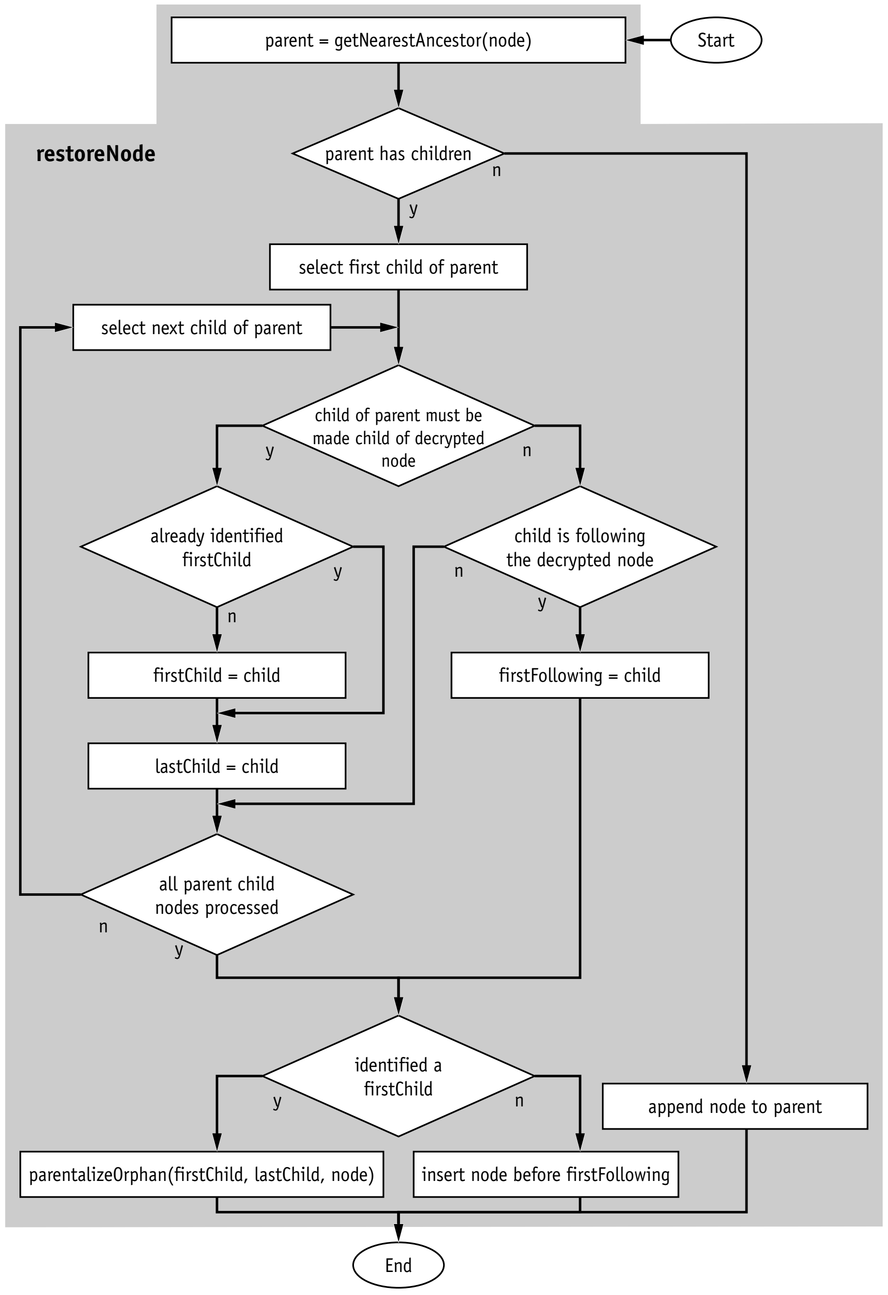

parent = getNearestAncestor(node)
Start
restoreNode
parent has children
n
y
select first child of parent
select next child of parent
child of parent must be made child of decrypted node
y
n
already identified firstChild
y
n
child is following the decrypted node
n
y
firstChild = child
firstFollowing = child
lastChild = child
all parent child nodes processed
n
y
identified a firstChild
y
n
append node to parent
parentalizeOrphan(firstChild, lastChild, node)
insert node before firstFollowing
End

Figure 7-33: restoreNode algorithm

The purpose of the restoreNode function is to perform the node restoration procedure. For doing this, it requires both the getNearestAncestor and parentalizeOrphan functions.

The input to the restoreNode function is

❑ a labeled document and

❑ a decrypted node.

The algorithm does the following:

1. The parent of the decrypted node is determined by the `getNearestAncestor` function.
2. If the parent has *no* children, the decrypted node is appended to the parent directly.
3. If the parent *has* children, it must be determined if some the current children of the parent node must become children of the decrypted node and where in the children list the decrypted node must be inserted.
   For doing this, the algorithm iterates over the children of the parent. The term 'childOfParent' refers to the parent's child node which is currently analyzed.
    1. If the childOfParent is a descendant of the decrypted node, it must be marked for relocating into the decrypted node. This is done storing the first descendant node in the firstChild variable. Additionally, each descendant is stored in the lastChild variable (overriding previous values).
    2. If a childOfParent is following the decrypted node, it is stored in the firstFollowing variable. Once such a node is detected, the iterating over the child nodes of the parent stops.
4. If the algorithm detected a firstChild, then parentalizeOrphan is called to foster all child nodes between firstChild and lastChild into the decrypted node and to put the decrypted node into the right place.
5. If the algorithm detected no firstChild, the decrypted node is added to the parent node before firstFollowing.

7.13 A restoration example

The following example shows how the algorithms described in the previous sections restore a public document after three nodes and their associated labels have been decrypted (figure 7-34).

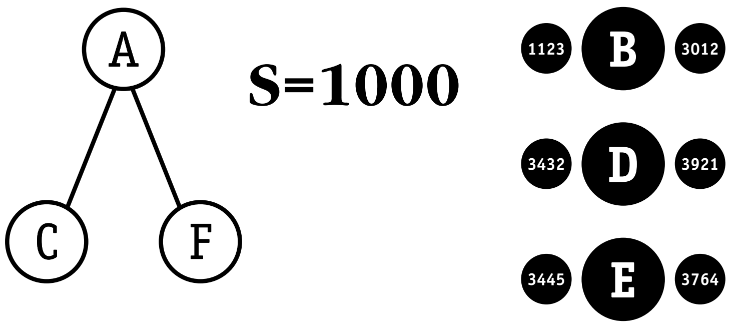


Figure 7-34: Input to the node restoration procedure

The first step is to re-label the public document, using the given stepsize S. A “Modified ALM” traversal is performed on the public document. After that traversal, all public nodes in the public document are labeled with the even label values (see figure 7-35).

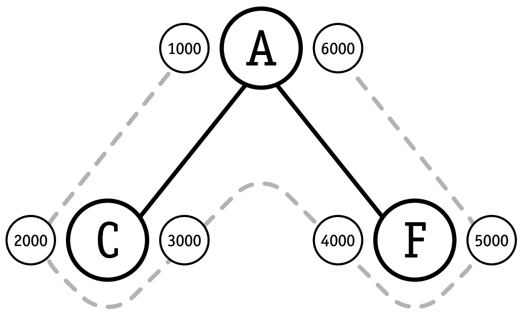


Figure 7-35: Re-labeled public document

After that re-labeling step, all nodes (public nodes and decrypted nodes) have labels, but the decrypted nodes are not yet restored in the document.
This section shows how the three decrypted nodes have to be restored in the document. The sequence in which the decrypted nodes are restored is not important to the result. In this example, the node B is restored at first, then E and D follow.

7.13.1 First node restoration example

The first node to be restored is node B. The node B has the label (1123,3012). According to figure 7-33 on page 134, the first step of the restoreNode algorithm is to call getNearestAncestor in order to determine the parent node for B.

The getNearestAncestor algorithm (figure 7-29 on page 131) iterates over the nodes in the document and detects that the node A (1000,6000) is the nearest ancestor of B (1123,3012).
The node A has children, therefore restoreNode iterates over the child nodes of A. The first node, C (2000,3000) is a descendant of B (1123,3012). Therefore, C is marked to be the firstChild of B. Additionally, it is marked to be the lastChild of B. The next node F (4000,5000) is following B (1123,3012), therefore F is marked to be firstFollower of B and the iteration over A's children terminates (see the restoreNode algorithm in figure 7-33 on page 134).

During the iteration over A's children, the firstChild value has been set, therefore restoreNode uses the parentalizeOrphan algorithm (figure 7-32 on page 133) to integrate the decrypted node B into the document.

parentalizeOrphan(firstChild=C,lastChild=C,node=B) is called: B is inserted after the lastChild (in that case after the C node). After that, all child nodes from firstChild to lastChild (in that case only the C node itself) are detached from their parent node and are made children of B.
The result of restoring node B is shown in figure 7-36.

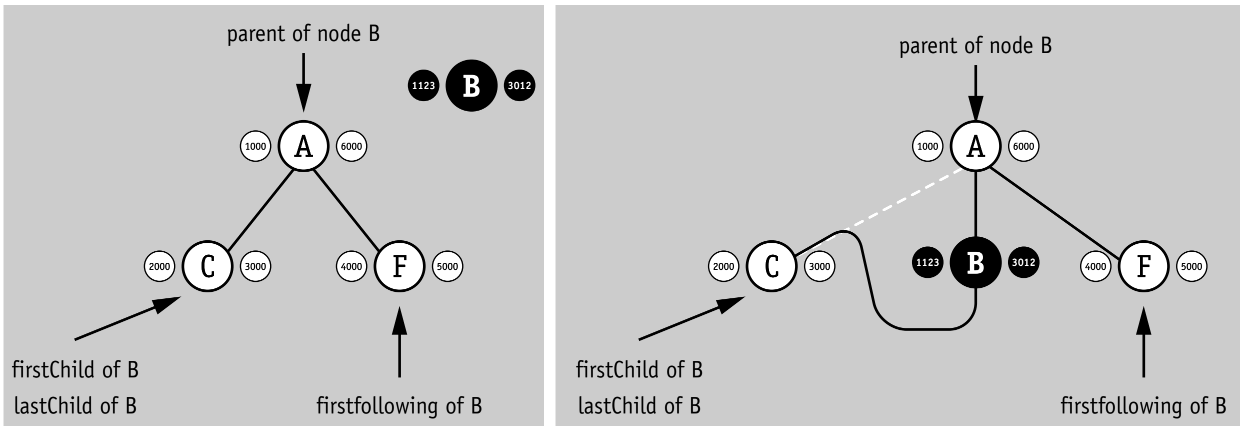


Figure 7-36: First node restoration example

7.13.2 Second node restoration example

The next node to be restored is node E. The node E has the label (3445,3764). The first step of the restoreNode algorithm is to call getNearestAncestor in order to determine the parent node for E. getNearestAncestor (figure 7-29 on page 131) iterates over the nodes in the document and detects that the node A (1000,6000) is the nearest ancestor of E (3445,3764).

The node A has children, therefore restoreNode iterates over the child nodes of A. The first node, B (1123,3012) is a preceding node to E (3445,3764). The next node F (4000,5000) is following B (1123,3012), therefore F is marked to be firstFollower of B.

During the iteration over A's children, both the firstChild and the lastChild values have not been set, therefore restoreNode just inserts the decrypted node before the firstFollowing node E (algorithm on figure 7-32 on page 133).

The result of restoring node E is shown in figure 7-37.

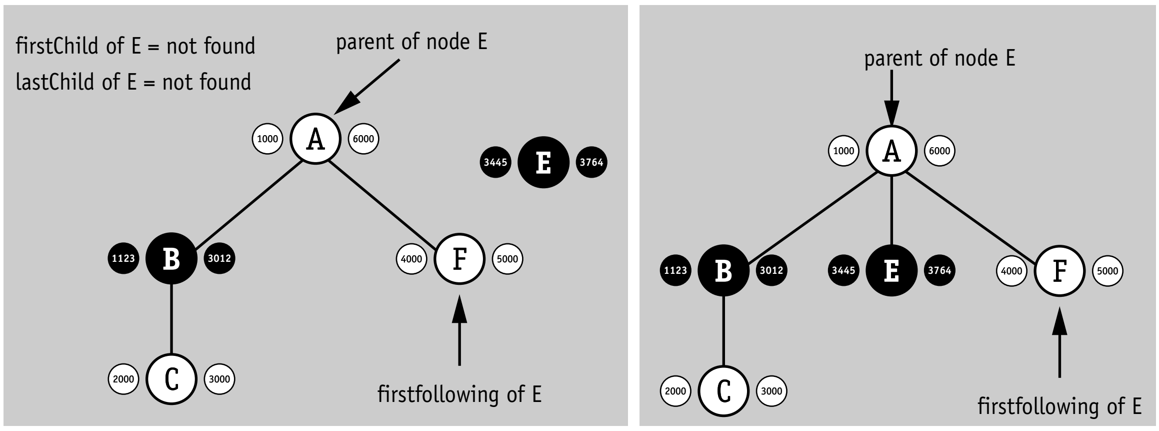


Figure 7-37: Second node restoration example

7.13.3 Third node restoration example

The last node to be restored is node D. The node D has the label (3432,3921). getNearestAncestor (figure 7-29 on page 131) detects that the node A (1000,6000) is the nearest ancestor of D (3432,3921).

The node A has children, therefore restoreNode iterates over the child nodes of A. The first node, B (1123,3012) is a preceding node to D (3432,3921). The next node E (3445,3764) is a descendant of D (3432,3921). Therefore, E is marked to be the firstChild of D. Additionally, E is marked to be the lastChild of D. The next child of A is F. The node F (4000,5000) is following D (3432,3921), therefore F is marked to be firstFollower of D and the iteration over A's children terminates.

During the iteration over A's children, the firstChild value has been set, therefore parentalizeOrphan(firstChild=E,lastChild=E,node=D) is called: D is inserted after the lastChild (in that case after the E node). After that, all child nodes from firstChild to lastChild (in that case only the E node itself) are detached from their parent node and are made children of D.

The result of restoring node D is shown in figure 7-38.

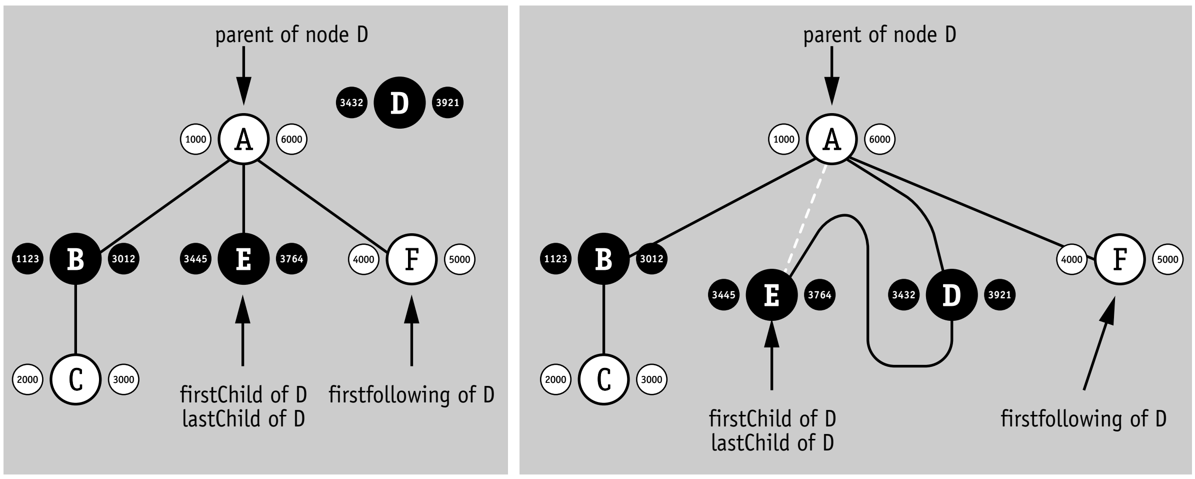


Figure 7-38: Third node restoration example

After restoring these three nodes, the example document looks like in figure 7-39:

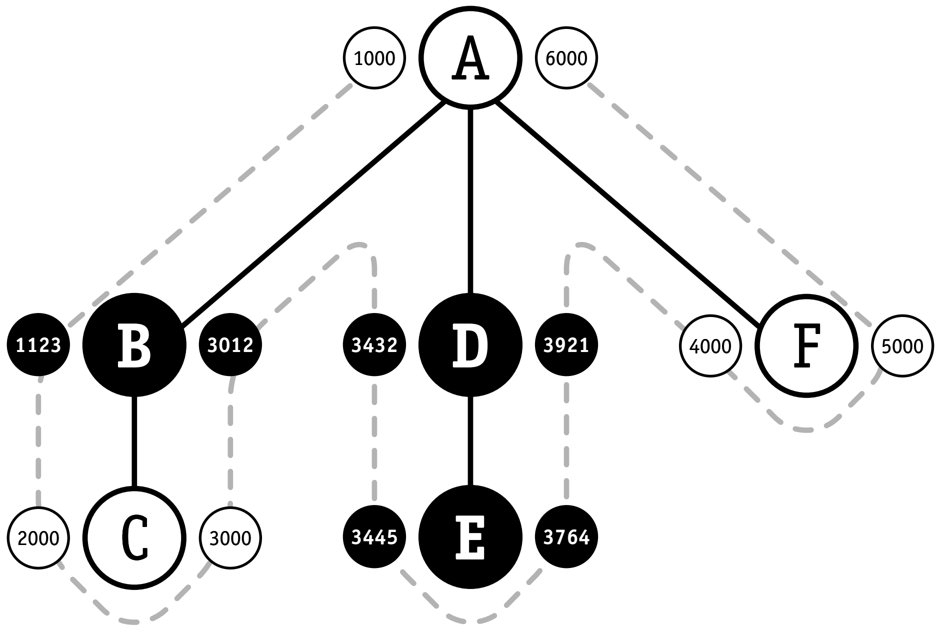


Figure 7-39: Result of the node restoration procedure examples

7.14 Encryption granularity

The XML Pool Encryption system must be capable to encrypt XML information set items. For a list of infoset items and their respective properties, see table 3-1 on page 34.
For each item type, the following sections evaluate whether it makes sense to encrypt the item and which infoset properties of the item have to be encrypted, i.e. which node data is the confidential content of a given node. For instance, a text node's properties are the character information items which consist the text itself.

7.14.1 Document information item

Each document has exactly one single "document information item", but this item only serves as a synthetic node which must exist in order to have a single root for the tree. The document information item has no physical representation in the serialized document and does not contain confidential information. Therefore, the document information item is not considered for encryption.

7.14.2 Comment information items

Comment information items are leaf nodes in the tree, i.e. they do not have a [children] property. The only properties are the [content] and the [parent] property. The [content] is the plaintext to be encrypted and the [parent] information is the position of the node. As a comment cannot have any children, is is sufficient if the right value P_R of a comment is only 1 greater than the left value P_L:

$$P_{R,\text{Comment}} = P_{L,\text{Comment}} + 1$$

As the right value of a comment is always exactly 1 greater than the left value, the right value is redundant. Therefore, it is not necessary to store and encrypt it if a comment is encrypted. The left value contains all necessary label information for a comment.
The plaintext serialization format for comments must contain (a) the type information being set to ‘comment’, (b) the left value P_L and (c) the value of the comments [content] property.

7.14.3 Processing Instruction information items

Processing Instruction (PI) information items are leaf nodes in the tree, i.e. they do not have a [children] property. The only properties are the [target] and [content] properties, as well as the [parent] property. The [target] and [content] properties are the plaintext to be encrypted. The [parent] information is the position of the node. As a PI cannot have any children, the right-value P_R of a PI is defined to be only 1 greater than the left value P_L:

$$P_{R,\text{PI}} = P_{L,\text{PI}} + 1$$

As the right value of a processing instruction is always exactly 1 greater than the left value, the right value is redundant. Therefore, it is not necessary to store and encrypt it if a processing instruction is encrypted. The left value contains all necessary label information for a processing instruction.
The plaintext serialization format for PIs must contain (a) the type information being set to ‘processing instruction’, (b) the left value P_L and (c) the value of both the [target] and [content] properties.

7.14.4 Element information items

Element information items have various properties: [parent] and [children] are mapped to the position information of the element.
[prefix], [local name] and [namespace name] contain information about the syntax and the semantics of the element itself.
Encryption of an element information item includes therefore (a) the type information being set to ‘element’, (b) the left and the right values (P_L, P_R) and (c) the prefix, localname and namespace URI of the element.

7.14.4.1 Attribute handling

The element has the [attributes] property which contains all attributes in the given element. Depending on the decision of the user, the attributes of a given element can be included in the encryption process of the element or be excluded from the encryption process, so that each attribute must be encrypted separately from the element.

Inclusion of attributes in the encrypted element node. When encrypting an element information item like in example 7-3 on page 142, the encrypted node contains the previously mentioned minimum set of information and the two attribute information items. After successful decryption, the user does have access to both the element information item and the two attributes.

```
<element attrA="foo" attrB="bar" />
```

Example 7-3: Element with attributes

Separate encryption of attributes. If the encryptor chose to encrypt element and attributes separately, the user only sees the element information item itself. The both attribute information items are stored in separate encrypted nodes, like described in “Attribute information items” on page 143.

7.14.4.2 Namespace handling

The element information item has two properties which are related to the namespace space: the [namespace attributes] property which contains all namespace declarations which have an attribute representation in the given element and the [in-scope namespaces] property which summarizes all namespaces which are in scope, i.e. all namespaces declared in the given element and in ancestor elements.
Given example 7-4, in which the <toBeEncrypted> element is to be encrypted, while the <foo> element and the text child of <toBeEncrypted> remain unencrypted.

```
<foo>
   <toBeEncrypted xmlns:bar="http://example.org/#bar">
      bar:someText
   </toBeEncrypted>
</foo>
```

Example 7-4: Not visibly utilized namespace

The text node makes use of the bar prefix which is bound in the <toBeEncrypted> element. After encrypting the <toBeEncrypted> element, the xmlns:bar namespace declaration is no longer visible. Applications which process the public document have no access to this important context information. As namespaces can only be represented through special attributes, the context cannot be stored for a text node itself.
For handling this problem, two different approaches exist:

1. Elevate the namespace declaration to the parent element, so that the resulting plaintext looks like in example 7-5. Unfortunately, this approach has two drawbacks:
    1. The information set of the <foo> element is changed, as its [namespace attributes] and [in-scope namespaces] properties have been polluted with a new namespace binding.
    2. If the <foo> element already has an xmlns:bar binding with a different namespace URI, this process would destroy the infoset.
2. The only viable solution for this case is to include all [in-scope namespaces] from the confidential node in its ancestor public node.

```
<foo xmlns:bar="http://example.org/#bar">
      bar:someText
</foo>
```

Example 7-5: Plaintext after namespace elevation

An encryption implementation cannot determine whether a given text node (or attribute node value) requires a namespace binding. This is the reason why exclusive c14n [BER02] introduces the "*InclusiveNamespaces PrefixList*". This list enables applications to get explicit knowledge on which namespace prefixes are used in text nodes or attribute values.

In example 7-6, the namespace prefix bar is visibly utilized by the unencrypted bar:baz attribute in the <foo2> element. This situation is very similar to canonicalizing a document subset, in which the namespace declaration is simply inherited by the <foo2> element.

```
<foo>
   <toBeEncrypted xmlns:bar="http://example.org/#bar">
      <foo2 bar:baz="something" />
   </toBeEncrypted>
</foo>
```

Example 7-6: Visibly utilized namespace

7.14.5 Attribute information items

The position of an attribute information item is implicitly given through the [owner element] property, i.e. the attribute has the same position as the owner element.
[prefix], [local name] and [namespace name] and [normalized value] contain information about the syntax of the attribute itself. Additionally, the [attribute type] defines the type of the attribute. The attribute type is especially important for ID-type attributes, as the attributes don't have type after plain decryption, so that the [attribute type] allows to re-register the ID of an attribute in some XML parser implementations.

The [references] property is not considered, as it can only be set as a result of a DTD-validation process. The [specified] property is not considered as it is true for all attributes being encrypted.
Encryption of an attribute information item includes therefore (a) the type information being set to 'attribute', (b) the left and the right values (P_L, P_R) of the owner element, (c) the prefix, localname and namespace URI of the attribute, (d) the normalized attribute value, and the attribute type.
Additionally to this information, the [in-scope namespaces] of the [owner element] is included in the attribute.

7.14.6 Namespace information items

Namespace information items carry the [prefix] and [namespace name] properties. Namespace information items are not encrypted separately, but are always part of an element or attribute information item.

7.14.7 Character information items

Character information items are leaf nodes in the tree, i.e. they do not have a [children] property. The [parent] property contains the position information of the parent element. Based on the location of the character information item in the [children] property of the [parent], the position of the character information item is determined. The [character code] property contains the character data itself.
As a character information item cannot have any children, the right-value P_R is defined to be only 1 greater than the left value P_L:

$$P_{R,\,\mathrm{Char}} = P_{L,\,\mathrm{Char}} + 1$$

For that reason, the right value is redundant and obsoletes the need to serialize it.
The plaintext serialization format for character information items must contain (a) the type information being set to 'character', (b) the left value P_L and (c) the value of the [character code] property.
In order to allow efficient implementations, multiple character information items can be grouped in a single encryption step.

7.14.8 Document Type Decl information items

A DTD describes constraints for the structure of an XML document. A DTD in itself should not contain confidential material. Therefore, the encryption of DTD information items is ruled out for the described prototype.

7.14.9 Unexpanded Entity Reference information items

The system is supposed to work on fully-parsed XML instances, so that no "unexpanded entity references" occur.

7.14.10Unparsed Entity information items

The system is supposed to work on fully-parsed XML instances, so that no “unexpanded entities” occur.

7.14.11Notation information items

The system is supposed to work on fully-parsed XML instances, so that no “notations” occur.

7.15 Correctness of the Modified Adjacency List Mode

7.15.1 Introduction

The XML Pool Encryption Process involves six different steps for encryption and decryption:

1. The plaintext document is labeled, pruned and the confidential node are encrypted to create the public document.
2. The encrypted nodes are decrypted, the public document is re-labeled and the decrypted nodes are re-inserted into the document.

The purpose of this section is to prove that the pool decryption procedure correctly recreates the plaintext document if the user is able to decrypt all encrypted nodes. It is obvious that both plaintext document and decrypted document differ from each other when the user did not restore all encrypted nodes.
The pool decryption procedure must be the inverse of the node encryption procedure, i.e. all nodes which have been encrypted must be decrypted. This reduces the procedures to be covered in this proof to the labeling/re-labeling and the pruning/restoration procedures.

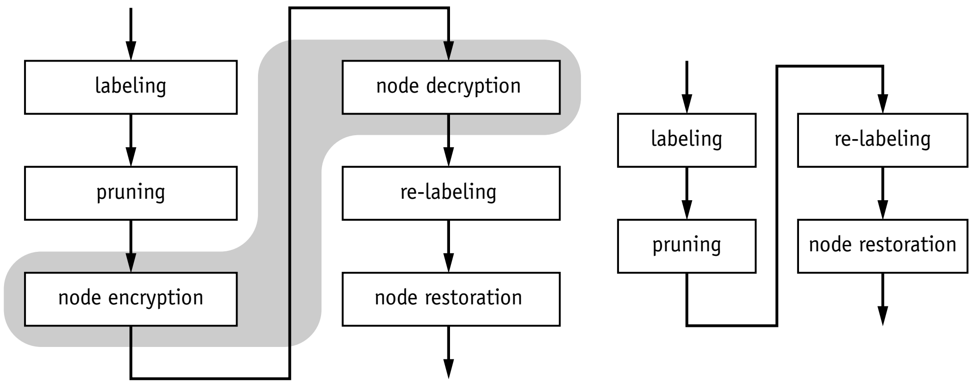


Figure 7-40: Removing the node encryption from the process

Theorem to be proved by contradiction:

- ❑ *Theorem T1:* The application of the pool encryption procedure and pool decryption procedure non-ambiguously and correctly encrypt and decrypt a given document.

By taking node encryption and node decryption out of the equation, T1 is transformed into T2 which has to be proved:

- ❑ *Theorem T2:* The application of the labeling, pruning, re-labeling and restoration non-ambiguously and correctly restores a given document.

During the pruning procedure, confidential nodes are removed from the plaintext document. During this process, the confidential node node is substituted by its descendants.

- ❑ *Theorem T3:* The pruning procedure does not change ancestor/descendant and following/preceding relationships.

For the proof, the following facts are necessary:

- ❑ *F1:* The MALM's labeling procedure generates a strictly monotonic increasing sequence of label values.
- ❑ *F2:* The even labels of the public nodes are the same after the labeling procedure (encryption) and after the node restoration procedure's re-labeling (decryption).
- ❑ *F3:* For each label value, it is non-ambiguously whether it is the left value or the right value and to which node it belongs.
- ❑ *F4:* A decrypted node which is restored in the correct position does not invalidate any on the given facts (F1-F3).

7.15.2 Proof of correctness

- ❑ *Assumption A1:* It is possible to restore a decrypted node in a wrong position without being detected.

- ❑ *Conclusion*: A traversal of the tree reveals that the sequence of the label values is not strictly monotonic increasing.

F1 is contradicted by the conclusion, so the assumption A1 is false.

7.15.3 Proof of non-ambiguous reconstruction

- ❏ *Assumption A2:* There exist two different decrypted documents which both are labeled correctly.

- ❏ *Conclusion*: A traversal of the tree reveals that the sequence of the label values is not strictly monotonic increasing.

The fact F1 is contradicted by the conclusion, so the assumption A2 is false.

7.16 Editing documents after encryption

This section discusses problems which relate to inserted nodes in a public document, which changed the document's structure.
During the labeling procedure of the plaintext document, the public nodes in the document are bound to specific label values. Structural changes in the public document result in different label values for the public nodes.

7.16.1 Destroying the label mechanism

Figure 7-41 illustrates the problem when nodes are inserted into a pool-encrypted element without further precautions.

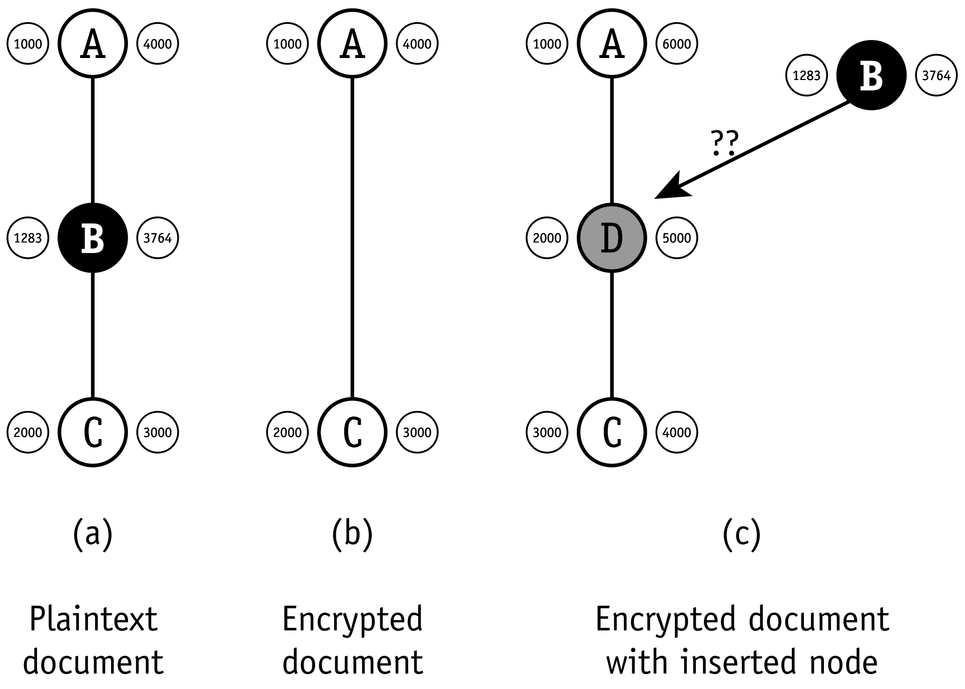


Figure 7-41: Different labeling after insertion of node

The plaintext document contains the three nodes A, B and C. Node B is a confidential node. Figure 7-41(a) shows the labeled plaintext document with $S = 1000$. The public document is shown in figure 7-41(b). In figure 7-41(c), the node D has been inserted between the nodes A and C. After that step, the document is to be decrypted and node B is to be restored. The document is re-labeled as shown in figure 7-41(c). The nodes A and C do now have different even label values. With the given labeling, it is impossible to restore the node B. Figure 7-42 illustrates how the intervals of the position information are now comprised (for details on this representation, also see figure 7-7 on page 96).

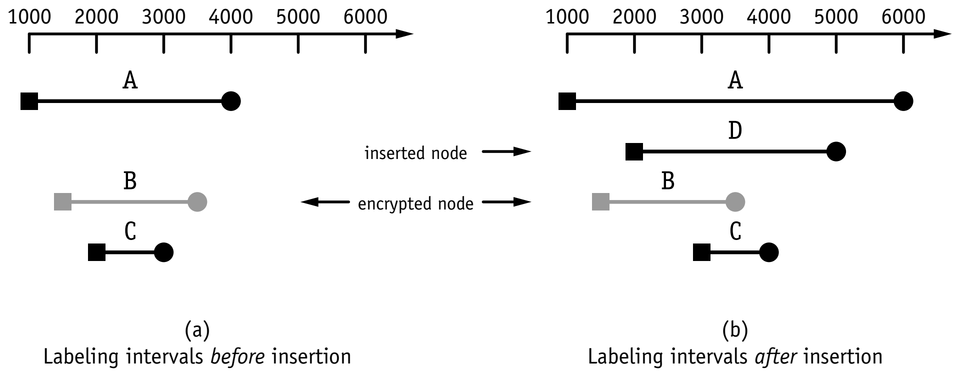

1000 2000 3000 4000 5000 6000
A
B
C
inserted node
encrypted node
D
(a)
Labeling intervals *before* insertion
(b)
Labeling intervals *after* insertion

Figure 7-42: Range mismatch

It can be seen in figure 7-42(b) that the re-generated label values of the nodes D, B and C do overlap after insertion of node D.
Without further precautions, a pool-encrypted document can be rendered undecryptable by editing and modification operations.

7.16.2 Enabling editing in public documents

To enable editing of public documents, the labeling procedure must be controlled to identify editable regions as such. The label control mechanism is based on adding three different attributes (mark types) to nodes in the tree which confine the editable area. A minimum of two mark types is needed to enable this mechanism; a third mark type is only introduced for performance reasons.
Figure 7-43 on page 150 illustrates the basic marking process. Figure 7-43(a) shows a simple plaintext document with four public nodes. These public nodes have been labeled with even label values. The second and the third node have been assigned the two marks "labeling=stop" and "labeling=stop" (shown as white/grey and grey/white circles). The grey ‘end’ of the marked nodes is at the side of the circle where the editable region borders on.
When labeling a marked tree, marked nodes are labeled with position information. The full tree is traversed during the position labeling process. The labeling process is temporarily disabled and re-enabled during tree traversal, based on the type of an encountered mark.
The DACP defines where modifications are permitted in the public document by assigning marks to the tree. For instance, nodes are added between the marked nodes in the middle of the tree as shown in figure 7-43(b). The grey

areas identify regions in which nodes or subtrees can be added, removed and modified without spoiling the labeling procedure.

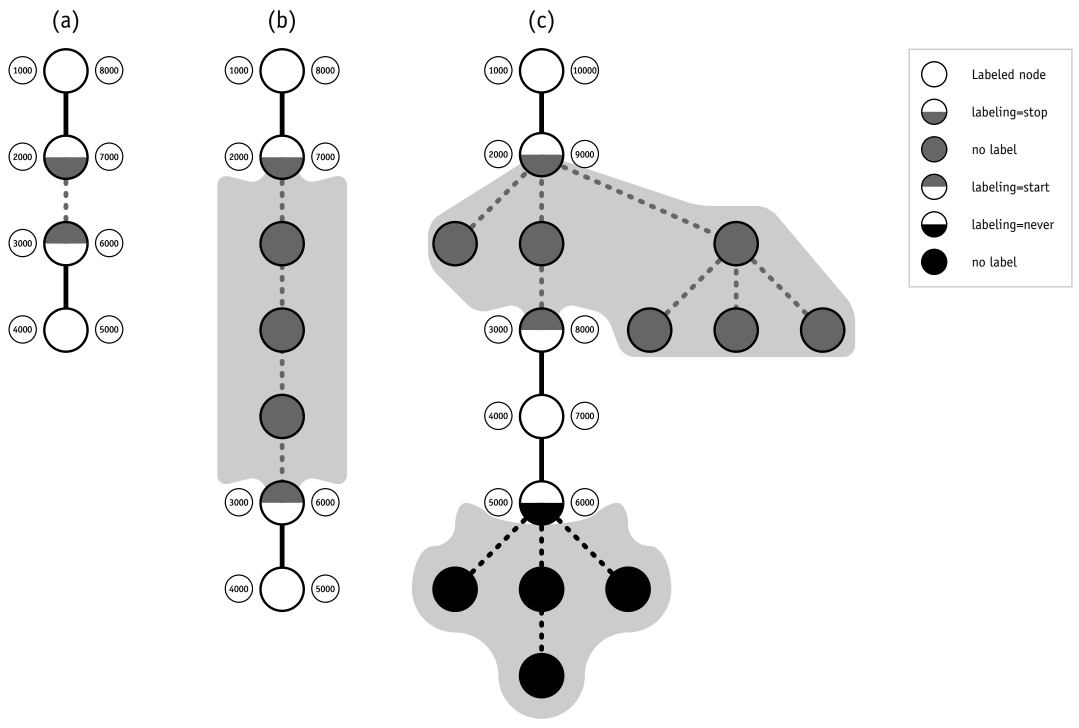


Figure 7-43: Editable region identification

The abstract description of the decision whether a given node is labeled with position information or not is as follows:

- ❑ A node is labeled with position information, if the first marked node on the ancestor-or-self axis has a labeling="start" mark or if no marked node at all is found in the axis.
- ❑ A node is *not* labeled with position information, if the first marked node on the ancestor axis has a labeling="stop" or a labeling="never" mark.

Both the labeling="start" and the labeling="stop" mark toggle the assignment of labels to even nodes during a traversal. The labeling="never" mark states that in the given subtree never ever a label will be assigned and that therefore no traversal has to be performed. A subtree, which is e.g. for commentary work in a public document, can be completely skipped during the traversal algorithm. So the labeling="never" mark is merely a performance improvement node. For being able to work, the labeling="start" and the labeling="stop" marks are sufficient.
In order to have a defined starting value, the document node is always marked labeling="start", so that labeling is enabled by default.

7.16.3 Trade-off between editability and structure awareness

Adding marks about editable regions in a pool-encrypted document partly exposes information about the plaintext document's structure to the user. The statement that a particular part of the public document can be edited includes the implicit fact that no encrypted nodes exist in the given part.
The edit marking scheme was introduced to give advice where new nodes can be added or where existing nodes can be modified or removed without making decryption impossible. The encrypted nodes cannot be decrypted and placed back into an editable region. Therefore, in an editable region, no encrypted nodes can exist.
As a result, increasing the size of editable regions exposes information about the plaintext document's structure to the users.

7.17 Schema validity and encryption

Documents which are valid with respect to a given schema are very fragile when the document structure is modified. Changes in the structure can destroy validity. Usually, a schema is written with a particular application in mind, i.e. when creating the schema, use cases and permitted operations on the data are already defined so that the schema can reflect these ideas in what data structures are allowed.

W3C XML Signature and XML Encryption as well as XML pool encryption are designed to secure arbitrary XML structures. Adding a digital signature (`<ds:Signature>` elements) into a document or encryption portions of a document by replacing the encrypted portion by an `<xenc:EncryptedData>` element introduces significant changes to the document. If these structure changes are performed without sufficient support in the schema definition, then schema validity breaks. This applies to any applications which introduces significant structural changes to a document.

W3C XML Encryption always must place an `<xenc:EncryptedData>` element into a document, if portions of the document are encrypted. This element is needed as an anchor for the decryption process as the user must be aware of the place where the decrypted structure is to be written back.

The interesting fact about XML Pool Encryption is that it does not necessarily introduce new namespaces into a document at multiple places. XML Pool encryption bundles all encrypted content and key management information from all over the document into one single place, the pool of encrypted nodes.

8 Properties of XML Pool Encryption

8.1 Confidentiality of arbitrary nodes

1. It must be possible to encrypt arbitrary nodes from the plaintext document.
2. The labeling procedure ensures that each node in the plaintext document has an unambiguous label.
3. The labels represent unambiguously the position of the node in the plaintext document.
4. Because of (3), each node can be restored by the node restoration procedure.
5. Because of (2) and (4), each node in the plaintext document can be encrypted.

During the node encryption procedure as defined by this thesis, both the interstitial label x_j and the confidential node's plaintext c_j are encrypted under the node key k_j. So an encrypted node e_j has the form $e_j = E_{k_j}(x_j \| c_j)$.

Note: To provide the confidentiality only for the node itself, it *would* be sufficient to encrypt only confidential node's plaintext c_j so that the encrypted node e_j would be a tuple of the interstitial label x_j and the encrypted plaintext: $e = x \| E(c)$. In that case, the interstitial label x_j would be stored in the clear, so that any user would know the positions of the encrypted nodes.

8.2 Confidentiality of the original structure

The labeling procedure requires the stepsize parameter to label the plaintext document and the public document. The stepsize is the only parameter necessary to fully label the public document. The stepsize parameter by itself does not contain confidential information about the plaintext document's structure.
All confidential information about the relation between the plaintext document's structure and the encrypted node is in the encrypted node's XML information and in its interstitial label. These items are encrypted, therefore the original structure is kept confidential.

1. The structure of the plaintext document must be hidden.
2. The structure is stored in the label values of the public nodes and the confidential nodes.
3. The label values of the public nodes can be restored with knowledge of the stepsize.
4. Because of (2) and (3), all structure information is the public document, the stepsize and the interstitial labels.
5. The public document is not confidential.
6. The stepsize is not confidential.
7. Because of (4), (5) and (6), all confidential structure information is in the interstitial labels. Therefore, the interstitial labels must be confidentiality protected.

8.3 Confidentiality of the total number of confidential nodes

1. The total number of confidential nodes must be hidden.
2. Confidential nodes are stored as encrypted nodes.
3. The number of confidential nodes cannot be changed by the pool encryption procedure.
4. Because of (2) and (3), the number of encrypted nodes must be changed. This is done using the concept of dummy nodes.
5. The existence of dummy nodes must be kept secret.
6. Because of (5), only the DACP must be able to separate 'real' encrypted nodes from dummy nodes.

8.4 Plausible deniability

Michael Roe introduces the term ‘*plausible deniability*’ in his Ph.D. thesis [Roe97, pp. 48]. He remarks that if the system aims to provide this service, it can be counterproductive to describe the plausible deniability property in the system documentation or in the requirements document of a new system. This is a problem because admitting that a particular system was designed with plausible deniability in mind is a strong indicator that users of the system use this feature, and that therefore, they are a priori guilty. Nevertheless, plausible deniability is explicitly mentioned in this chapter, because it is a feature which would be a ‘nice-to-have’.
Installing a system which provides plausible deniability always places the user at risk that the adversary becomes aware of the existence of the system: users which do use steganographic tools like steganographic file systems or steganographic multimedia software are always suspicious.

9 Conclusions

9.1 Summary

The present thesis describes existing security mechanisms for XML, including W3C XML Encryption and XML Access Control. These mechanisms have been classified based on their ability to protect arbitrary node sets. The lack of W3C's XML Encryption to encrypt arbitrary node sets led to the need for a further mechanism. This mechanism has been designed and implemented as the XML Pool Encryption system.
XML Pool Encryption solved the problem by combining different approaches from database technology and the XML world. Multiple examples have shown the internals of the algorithms. It has been proved that XML Pool Encryption solves the given requirements.
The concept of labeling a tree and encrypting the labels resulted in a system that can keep the full document structure confidential.

9.2 Future work

The current algorithms operate on an underlying DOM tree structure. These DOM tree structures are in-memory structures in which pointers create links between parent and child nodes. These pointers make the navigation on a tree very easy.
For XML Pool Encryption, the node restoration procedure is complicated, as it has to identify the child nodes and the next sibling of the decrypted node. The lookup of parent nodes for a decrypted node is a costly operation, as it has to iterate over a set of nodes.
The next step would be to create an implementation that completely drops the DOM concept but stores all types of nodes in a single big set. Using such a structure, the node restoration procedure would simply add the decrypted node to the set. On the other hand, this approach would make tree navigation more complex.
Another area of future research is the combination of XML Pool Encryption with security mechanisms like XML Signature.

Annex: Implementation

The implementation section shows how XML Pool Encryption is implemented in a prototype.

A.1 Implementation of XML Pool Encryption

The XML Pool Encryption system has been implemented using a Java library. The figure A-1 illustrates the various classes of this library.

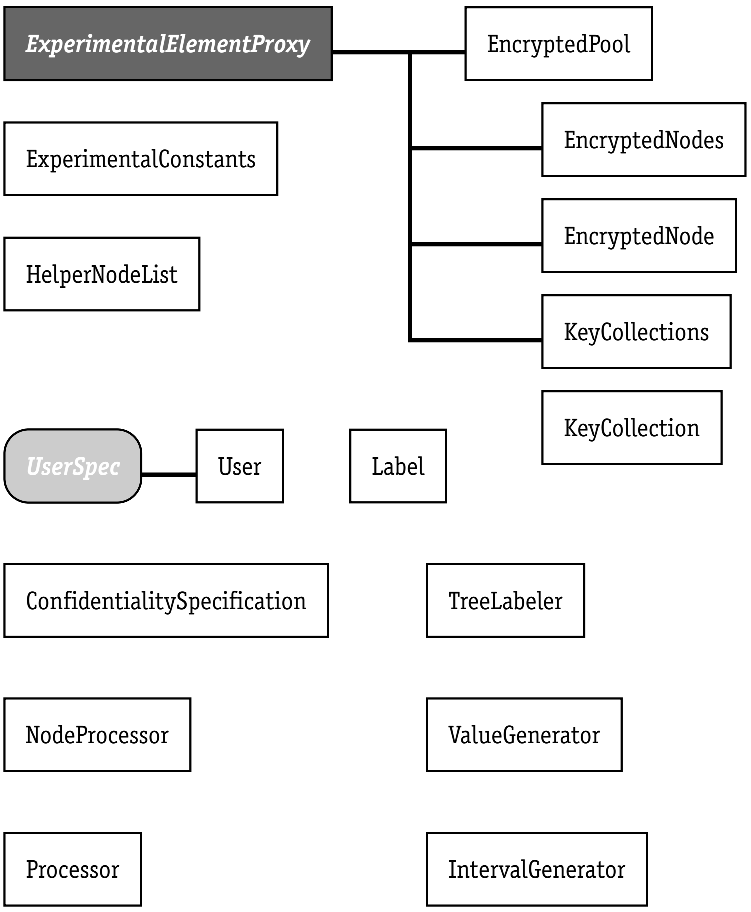


Figure A-1: Classes for XML Pool Encryption

ExperimentalElementProxy: is the base class for all objects which have an XML representation, i.e. which map to elements. For instance, the EncryptedNode class has an XML representation.

EncryptedPool: The EncryptedPool is a simple container for one EncryptedNodes element and one KeyCollections element.

EncryptedNodes: The EncryptedNodes class is the representation for the pool of encrypted nodes and contains the individual EncryptedNode elements.

EncryptedNode: The EncryptedNode class models the encrypted node objects, i.e. it can store the NodeID attribute and the ciphertext of the encrypted node.

KeyCollections: The KeyCollections element contains multiple KeyCollection elements.

KeyCollection: Each KeyCollection contains an UserID attribute to identify the intended user and a set of node keys with their respective NodeID attributes.

The following example illustrates the relationship between EncryptedPool, EncryptedNodes, EncryptedNode, KeyCollections and KeyCollection elements.

```
<EncryptedPool StepSizeBits="...">
   <EncryptedNodes>
      <EncryptedNode NodeID="...">+
         Base64-encoded ciphertext
      </EncryptedNode>
   </EncryptedNodes>
   <KeyCollections>?
      <KeyCollection UserID="...">+
         <!-- This is encrypted under the user's key -->
         <xenc:CipherData>
           <NodeKey NodeID="...">+
             Base64-encoded node key
           </NodeKey>
         </xenc:CipherData>
      </KeyCollection>
   </KeyCollections>
</EncryptedPool>
```

UserSpec: The UserSpec provides an Interface to retrieve a UserID from an object.

User: The User class is a wrapper class which implements the UserSpec interface and simply wraps a String as simple user identifier.

ExperimentalConstants: contains various constants like element names.

HelperNodeList: An ordered set or Nodes; implements the org.w3c.dom.NodeList interface.

ConfidentialitySpecification: The ConfidentialitySpecification allows to define a mapping between UserSpec objects and Nodes. The ConfidentialitySpecification is required by the node selection procedure. The DACP creates the ConfidentialitySpecification.

NodeProcessor: The NodeProcessor encrypts and decryptes single EncryptedNode objects. It is responsible for correct serialization and de-serialization of a node's plaintext.

Processor: The Processor encrypts and decrypts full documents. It takes the plaintext document and the ConfidentialitySpecification to perform this step and returns the public document and an EncryptedPool.

TreeLabeler: The TreeLabeler performs the labeling procedure and keeps track of all public nodes and decrypted nodes. The TreeLabeler uses a ValueGenerator to create label values.

ValueGenerator: The ValueGenerator is the representation of the TreeLabelers internal state. The ValueGenerator creates even label values and interstitial sequences. The interstitial sequences are created by an IntervalGenerator.

IntervalGenerator: The IntervalGenerator creates interstitial sequences.

A.2 Syntax of pool encryption

This section describes the XML syntax of the pool encryption system in a non-formal way. The namespace bindings used in this section are listed in table A-1.

prefix	namespace URI
ds	http://www.w3.org/2000/09/xmldsig#
penc	http://www.xmlsecurity.org/experimental#
xenc	http://www.w3.org/2001/04/xmlenc#

Table A-1: Namespace prefix bindings

All elements in pool encryption are in the namespace. The XML skeleton of pool encryption is shown in figure A-2 (the lax syntax used here is as used in section 2 of [ER02]).

```
<EncryptedPool StepSizeBits>
  <EncryptedNodes>
    <EncryptedNode NodeID />+
  </EncryptedNodes>
  <KeyCollections>
    <KeyCollection UserID>+
      <xenc:CipherData />
    </EncryptedKeyCollection>
```

Figure A-2: XML Skeleton of an EncryptedPool

```
  </KeyCollections>
</EncryptedPool>
```

Figure A-2: XML Skeleton of an EncryptedPool

A.2.1 EncryptedPool

The `<EncryptedPool>` element is the container for both the pool of encrypted nodes and the key pools for the different users. For each document, exactly one `<EncryptedPool>` is permitted to be included. Multiple `<EncryptedPool>` elements can be merged into a single pool by a union operation of the distinct sets. The `<EncryptedPool>` element contains one `<EncryptedNodes>` element which contains the set E of the encrypted nodes C_i and the `<KeyCollections>` element which is the container for the encrypted key pools P_{u_j} for the individual users u_j (see section 7.6 on page 113).

A.2.2 EncryptedNodes

The `<penc:EncryptedNodes>` element is the representation of the set E and contains the encrypted nodes C_i. The `StepSize` attribute contains the step-size S which had been defined by the encryptor for the given document.

A.2.3 EncryptedNode

Each `<penc:EncryptedNode>` element contains a single encrypted node C_i. The individual nodes can be identified throught the `Id` attribute. This is necessary to allow a *node key* in a `<penc:KeyCollection>` to refer to the encrypted node it belongs to.
The ciphertext of the encrypted node is a base64-encoded string which is a text child of the `<penc:EncryptedNode>` element. The format of the plaintext is defined in section A.2.7 on page 163.

A.2.4 KeyCollections

The `<penc:KeyCollections>` element serves as container for (at least one) `<penc:EncryptedKeyCollection>` element.

A.2.5 EncryptedKeyCollection

Each `<penc:EncryptedKeyCollection>` element contains the encrypted pool P_{u_j} of node keys. After successful decryption of the pool's contents, the node keys are available to the user. The key collection itself is encrypted using W3C XML Encryption, therefore the `<penc:EncryptedKeyCollection>` contains an `<xenc:EncryptedData>` element. The `<xenc:EncryptedData>` element contains an encrypted `<penc:KeyCollection>` element. For details on W3C XML Encryption, see the introduction in section 5.3 on page 69 and the original specification [ER02].

A.2.6 KeyCollection

Decryption of the `<penc:EncryptedKeyCollection>`'s `<xenc:Encrypted-Data>` element yields access to the `<penc:KeyCollection>` element.

```
<penc:KeyCollection>
   <penc:NodeKey nodeRef keyValue />+
</penc:KeyCollection>
```

Figure A-3: XML Skeleton of KeyCollection

The `<penc:KeyCollection>` contains `<penc:NodeKey>` elements which contain the literal key value of the node key k_{e_i} and a `nodeRef` attribute which refers to the encrypted node e_i in the `<penc:EncryptedNode>` element. e_i can be decrypted with the node key.

A.2.7 Serialization format for confidential nodes

The plaintext of a confidential node contains three general types of information:

1. The node's label, i.e. the left and right interstitial label values v_l and v_r. The two values are base64-encoded integer values.
2. The node's *type*, i.e. whether the following infoset belongs to e.g. an element, a comment, a (sequence of) character information items ...
3. The node's information set. The serialization format used here is the same as defined by Canonical XML, so that the node(s) can simply be re-parsed.

These three fields are separated by space values. An example of the plaintext of an element could look like in example A-1[1].

```
AAAAAAAAAANmEJzxhqBYIg==·AAAAAAAAAAPN1IuH3jzhWA==·ELE·
<foo:bar·xmlns:foo="http://www.foo.com/#"·/>
```

Example A-1: Decrypted <penc:EncryptedNode>'s content

The first base64-encoded value is a 128 bit integer representing the left interstitial value with an even part of 0x03 (leading zeros) and an interstitial part of `0x66 0x10 0x9C 0xF1 0x86 0xA0 0x58 0x22`. The second base64-encoded value is a 128 bit with an even part of 0x03 and an interstitial part of `0xCD 0xD4 0x8B 0x87 0xDE 0x3C 0xE1 0x58`. So the position of the node is (l,r) =(62694782973784381474, 70171865109631918424) in decimal representation. The type information is set to "ELE" which means element content. The parseable element contains a `bar` element with a `foo` prefix and the assiciated namespace value.

1. The single line of plaintext is decomposed into four lines for simplicity.

A.3 The Apache XML Signature Implementation

The following chapter contains descriptions of the single components used for the XML Signature library. If not otherwise stated by explicit package names which refers to standard packages like java.util or org.apache.xpath, all used pachage and class names have to be prefixed by org.apache.xml.security to get the *real* pachage name:

- ❑ java.util.Comparator is a standard class called Comparator in the java.util.* package
- ❑ utils.* refers to the package org.apache.xml.security.utils.*
- ❑ c14n.implementations.Canonicalizer20010315WithComments refers to the org.apache.xml.security.c14n.implementations.Canonicalizer20010315WithComments class

A.3.1 org.apache.xml.security.* package

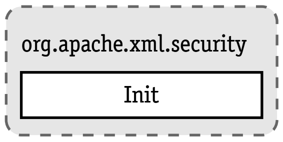


Figure A-4: The org.apache.xml.security.* package

- ❑ Init: The Init class contains initialization functionality for the complete library. This includes the following initializations:
  - ❍ The global configuration is read from the classpath; this is done by retrieving the `resource/config.xml` file from the JAR file
  - ❍ The utils.PRNG singleton is initialized with a java.security.SecureRandom object
  - ❍ The log4j logging system is set up based on `config.xml`
  - ❍ The internationalization is set up based on the locale from `config.xml` (see utils.I18n)
  - ❍ The `here()` function as defined by the XML Signature recommendation is registered in the Xalan XPath engine
  - ❍ Register the available canonicalization algorithms (see c14n.Canonicalizer)
  - ❍ Register the available transformation algorithms (see transforms.Transform)
  - ❍ Register the JCE mappings (see utils.JCEMapper)

- ❍ Register the available signature and MAC algorithms (see algorithms.SignatureAlgorithm)
- ❍ Register the available resource resolvers (see utils.resolver.ResourceResolver)
- ❍ Register the available `<KeyInfo>` content handlers
- ❍ Register the available `<KeyInfo>` resolvers
- ❍ Define default prefixes for different namespaces
- ❍ Register the available encryption algorithms

A.3.2 org.apache.xml.security.algorithm.**.* Package

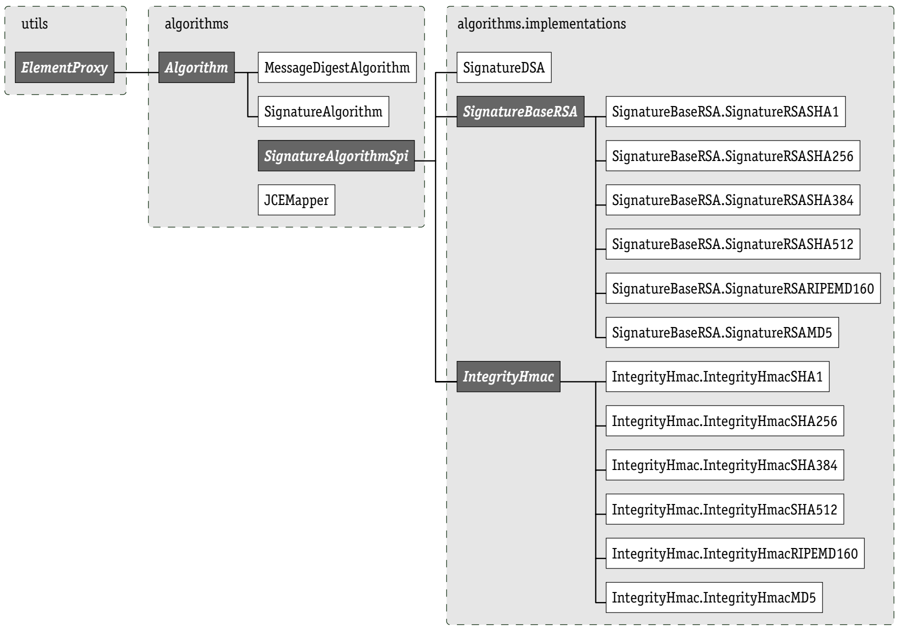

utils
ElementProxy
algorithms
Algorithm
MessageDigestAlgorithm
SignatureAlgorithm
SignatureAlgorithmSpi
JCEMapper
algorithms.implementations
SignatureDSA
SignatureBaseRSA
SignatureBaseRSA.SignatureRSASHA1
SignatureBaseRSA.SignatureRSASHA256
SignatureBaseRSA.SignatureRSASHA384
SignatureBaseRSA.SignatureRSASHA512
SignatureBaseRSA.SignatureRSARIPEMD160
SignatureBaseRSA.SignatureRSAMD5
IntegrityHmac
IntegrityHmac.IntegrityHmacSHA1
IntegrityHmac.IntegrityHmacSHA256
IntegrityHmac.IntegrityHmacSHA384
IntegrityHmac.IntegrityHmacSHA512
IntegrityHmac.IntegrityHmacRIPEMD160
IntegrityHmac.IntegrityHmacMD5

Figure A-5: The org.apache.xml.security.algorithm.**.* package

The algorithms and algorithms.implementations packages contain XML Signature related algorithm factories and the implementations of these algorithms. The message digests, signature algorithms and HMACs are all used in the same fashion: The JCEMapper is used to identify the Java JCE/JCA class which does the real cryptographic work; an instance of this workhorse is included inside the implementation class; additionally, it handles the construction/parsing from and serialization to Elements.

- ❑ The JCEMapper class is a registry for mapping algorithm URIs to JCE/JCA algorithm names. The Sun JCE/JCA architecture does not enforce that all cryptographic service providers use the same algorithm identifiers for the same algorithm. The JCEMapper is given an algorithm URI and returns the name of the specific algorithm and the corresponding provider ID.

This includes some reflection magic to check which providers are available in the classpath and to register them as required.

- The abstract Algorithm class is the base class for all algorithms which directly map to an Element in an XML instance and in which the used algorithm is defined using a URI attribute.
- The MessageDigestAlgorithm class is the factory and proxy class which is used for handling <ds:DigestMethod> elements.
- The SignatureAlgorithm class is the factory and proxy class which is used for handling <ds:SignatureMethod> elements. It is a wrapper for SignatureAlgorithmSpi objects.
- The SignatureAlgorithmSpi class is an abstract class which is extended by the implementations of DSA, RSA and HMAC classes which reside in the algorithms.implementations package.
- The SignatureDSA class implements the DSS/DSA (Digital Signature Standard/Digital Signature Algorithm), which is defined using a fixed hash algorithm (SHA-1). No variants of this class exist.
- The abstract SignatureBaseRSA and IntegrityHmac classes do implement the basic functionality for RSA based signatures and HMACs.
    - The final implementation is done by the nested inner classes, which are named by their corresponding message digest algorithm: e.g., an RSA signature which utilizes the RIPE-MD160 hash algorithm is implemented by the inner SignatureBaseRSA$SignatureRSARIPEMD160 class.

A.3.3 org.apache.xml.security.c14n.**.* Package

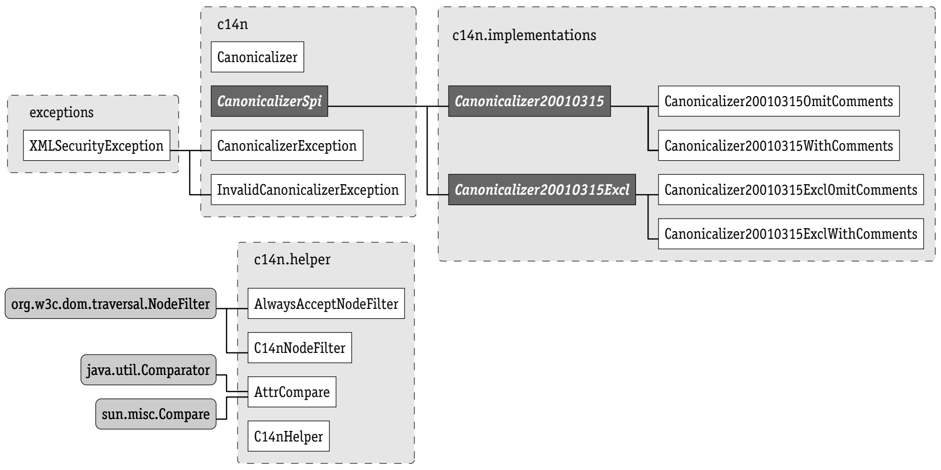


Figure A-6: The org.apache.xml.security.algorithm.**.* Package

The c14n and c14n.implementations packages include the core functionalities for canonicalization, the c14n.helper package includes some helper classes:

- ❑ Canonicalizer is the factory and proxy class for the c14n algorithms. Additionally, it is the registry for available c14n algorithms.
- ❑ CanonicalizerSpi is the abstract base class which all canonicalizer implementations inherit.
  - ❍ Canonicalizer20010315 implements the Canonical XML v1.0 recommendation. It is defined abstract because the inherited classes
    - ✰ Canonicalizer20010315WithComments and
    - ✰ Canonicalizer20010315OmitComments define whether comments are to be included or omitted.
  - ❍ Canonicalizer20010315Excl implements the Exclusive Canonical XML v1.0 recommendation. It is defined abstract because the inherited classes
    - ✰ Canonicalizer20010315ExclWithComments and
    - ✰ Canonicalizer20010315ExclOmitComments define whether comments are to be included or omitted.
- ❑ The CanonicalizationException is thrown if errors occur during canonicalization processing, e.g. relative namespaces in the input document.
- ❑ The InvalidCanonicalizerException is thrown if a non-registered canonicalizer is to be instantiated in the Canonicalizer.
- ❑ In the c14n.helper package, the following classes are defined:

- ❍ The AlwaysAcceptNodeFilter is an org.w3c.dom.traversal.NodeFilter which always returns true.
- ❍ The C14nNodeFilter is an org.w3c.dom.traversal.NodeFilter which returns true for all non-comment nodes. For comment nodes, it returns an internal boolean value (include or omit comments).
- ❍ The AttrCompare defines a java.util.Comparator which is needed by c14n for the ordering of attributes and namespaces.
- ❍ The C14nHelper class bundles some static utility functions needed during c14n.

A.3.4 org.apache.xml.security.keys.(content).* package

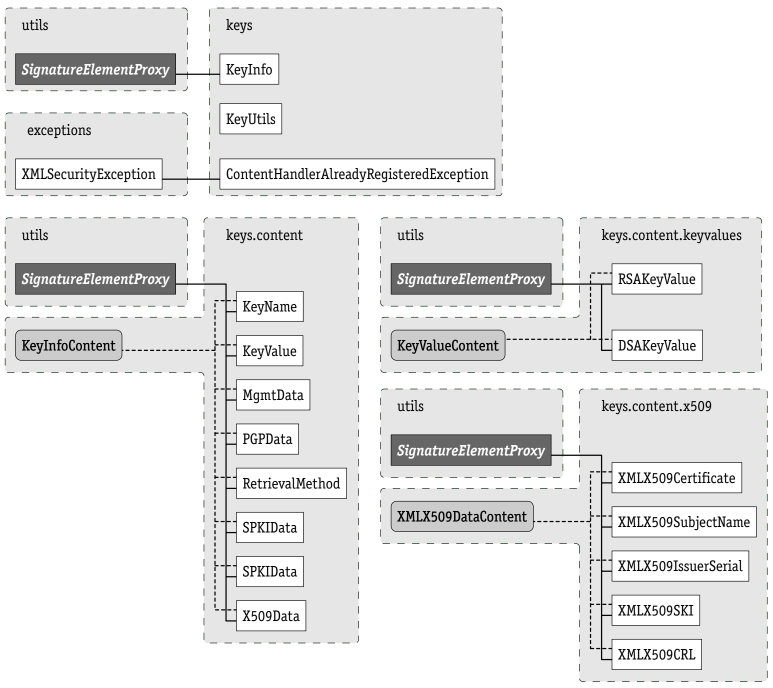


Figure A-7: The org.apache.xml.security.keys.(content).* package

The keys package (see figure A-7 on page 168) contains the KeyInfo object and some utilities; the keys.content, keys.content.keyvalues and keys.content.x509 packages (also in figure A-7) contain the content model for the different child elements of the `<ds:KeyInfo>` element:

- ❑ The KeyUtils class contains only a few static functions for debugging.
- ❑ The KeyInfo class models the `<ds:KeyInfo>` element. This includes a factory for `<ds:KeyInfo>` elements, easy access to all child elements and most important, functionality to use resolvers from the keys.resolver package to retrieve public keys and certificates from a `<ds:KeyInfo>`.

This enables to user to use simple statements like keyInfo.getPublicKey() to fetch a java.security.PublicKey from a KeyInfo object.

- ❑ The classes in the keys.content, keys.content.keyvalues and keys.content.x509 packages map directly onto the corresponding elements from the XML Signature specification. All these classes provide methods for easily accessing the contents of these elements.
- ❑ The interfaces KeyInfoContent, KeyValueContent and XMLX509DataContent are used for 'tagging' the elements in the given packages. This tagging specifies which element are allowed as children of `<ds:KeyInfo>`, `<ds:KeyValue>` and `<ds:X509Data>` elements. These interfaces do not introduce any methods.
- ❑ The ContentHandlerAlreadyRegisteredException is thrown if the user registers a key resolver, which has already been assigned.

A.3.5 org.apache.xml.security.keys.keyresolver.* package

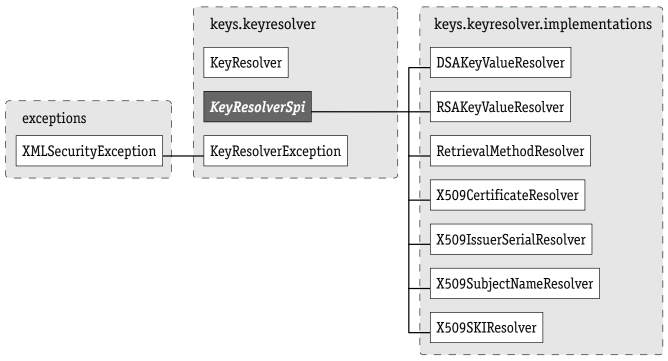


Figure A-8: The org.apache.xml.security.keys.keyresolver.* package

The key resolvers from the keys.keyresolver packare (see figure A-8 on page 169) are objects which extract public keys and certificates from elements (well, not from the elements but from their corresponding KeyInfoContent, KeyValueContent or XMLX509DataContent object):

- ❑ The KeyResolver class is the registry and factory to create arbitrary key resolvers and acts as a proxy for the underlying KeyResolverSpi.
  The KeyResolver class is given a `<ds:KeyInfo>` element and optionally some StorageResolvers. After this initialization, the user can simply ask for a public key or certificate. This query is delegated to all registered key resolver implementations, which respond whether they can resolve a given element to a public key or certificate.
- ❑ The KeyResolverException is thrown is an error occured during key resolving by the implementations.

- The KeyResolverSpi is the abstract base class to the different key resolver implementations. The different implementations are:
  - The DSAKeyValueResolver can extract DSA public keys from `<ds:DSAKeyValue>` elements.
  - The RSAKeyValueResolver can extract RSA public keys from `<ds:RSAKeyValue>` elements.
  - The RetrievalMethodResolver can retrieve public keys and certificates from other locations. The location is specified using the `<ds:RetrievalMethod>` element which points to the location. This includes the handling of raw (binary) X.509 certificate which are not encapsulated in an XML structure. If the retrieval process encounters an element which the RetrievalMethodResolver cannot handle itself, resolving of the extracted element is delegated back to the KeyResolver mechanism.
  - The X509CertificateResolver can extract public keys and X.509 certificates from `<ds:X509Certificate>` elements which carry base64 encoded certificates.
  - The X509IssuerSerialResolver extracts the issuer distinguished name/serial number pair from a `<ds:X509IssuerSerial>` element and iterates through a collection of certificates. If it finds the matching certificate, it returns the public key or the certificate itself.
  - The collection of certificates is modeled through a StorageResolver (discussed below).
  - The X509SubjectNameResolver extracts the subjects distinguished name from a `<ds:X509SubjectName>` element and iterates through a collection of certificates. If it finds the matching certificate, it returns the public key or the certificate itself.
  - The X509SKIResolver extracts the subject key indentifier from an `<ds:X509SKI>` element and iterates through a collection of certificates. If it finds the matching certificate, it returns the public key or the certificate itself.

A.3.6 org.apache.xml.security.keys.storage.* package

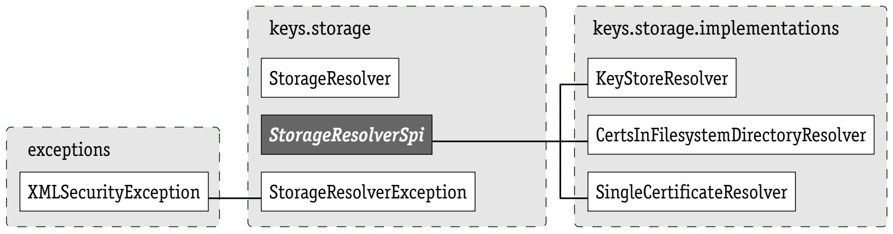


Figure A-9: The org.apache.xml.security.keys.storage.* package

Each system has different ways on how certificates are organized. Certificates can be stored in e.g. a LDAP directory, as plain, binary files in a directory on the local file system or in a KeyStore, which is the Java way to collect cryptographic keys and certificates. Some of the key resolvers need access to the certificates in order to match a snippet of information like a SKI or an IssuerSerial against the certificates to find the correct one.

❑ The StorageResolver is the interface to the certificate collections. The simplest way to access such a collection of certificates is an Iterator. The StorageResolver provides an interface to iterate over such a collection.

❑ The StorageResolverException is thrown if something goes wrong during the iteration process.

❑ The StorageResolverSpi is the abstract service provider interface class which all implementations must extend.
The library ships with the following implementations:

  ❍ The KeyStoreResolver iterates completely over the certificates contained in a given KeyStore.

  ❍ The CertsInFilesystemDirectoryResolver iterates over the files in a given directory in the file system and makes all binary certificates available to the caller. This is needed for unit testing against some test vectors.

  ❍ The SingleCertificateResolver makes a single Certificate available through the StorageResolver interface. This enables the user to supply multiple certificates (it's possible to add multiple StorageResolverSpi implementations to the StorageResolver in order to iterate over all of them).

A.3.7 org.apache.xml.security.signature.* package

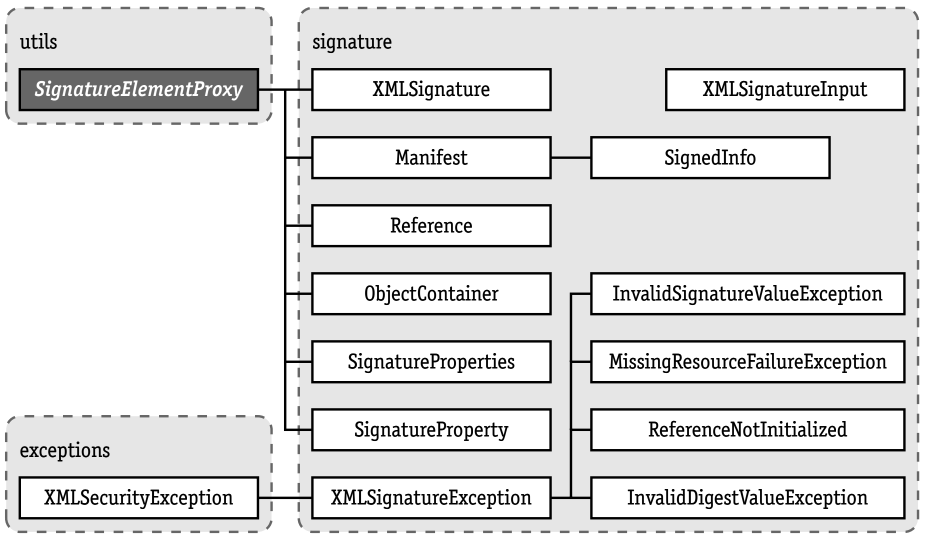


Figure A-10: The org.apache.xml.security.keys.signature.* package

The signature package contains the objects which model the XML Signature recommendation. All (but the XMLSignatureInput and the exceptions) extend the SignatureElementProxy, cause they implement elements from the signature namespace.

❑ The XMLSignature class handles `<ds:Signature>` elements. It's the main object needed for creating and verifying arbitrary XML Signatures. It provides proxy methods for invoking methods on included objects, e.g. it provides the addDocument() method for adding References to the underlying SignedInfo.

❑ The Manifest class handles `<ds:Manifest>` elements. It includes all functionality needed for reference validation.

❑ The SignedInfo class handles `<ds:SignedInfo>` elements and extends the Manifest. It adds canonicalization and signing functionality to the Manifest.

❑ The Reference class handles `<ds:Reference>` elements. This class includes methods for de-referencing contents from given URIs, applying transforms to the de-referenced contents and calculating the corresponding message digests. The de-referencing of URIs is done by a collection of ResourceResolvers. These ResourceResolvers are maintained by the Manifest or SignedInfo which 'owns' the Reference.

❑ The ObjectContainer class handles `<ds:Object>` elements[1].

1. It has been called ObjectContainer instead of Object to avoid name clashes with the lava.lang.Object object

- ❑ The SignatureProperties class handles `<ds:SignatureProperties>` elements.
- ❑ The SignatureProperty class handles `<ds:SignatureProperty>` elements.
- ❑ The XMLSignatureException is the base class for all XML Signature related exceptions.
- ❑ The most complex class in the signature package is the XMLSignatureInput class. This class represents the data which is de-referenced by the ResourceResolvers and which is then used as input and output of transforms. According to the XML Signature recommendation, the data type of the result of de-referencing a URI can be either an octet stream or an XPath node-set. This duality is represented by the XMLSignatureInput: the class allows easy and transparent conversion from octet streams to XPath node sets and vice versa, based on what the consumer (the TransformSpi) of such an object requires as input.

A.3.8 org.apache.xml.security.transforms.* package

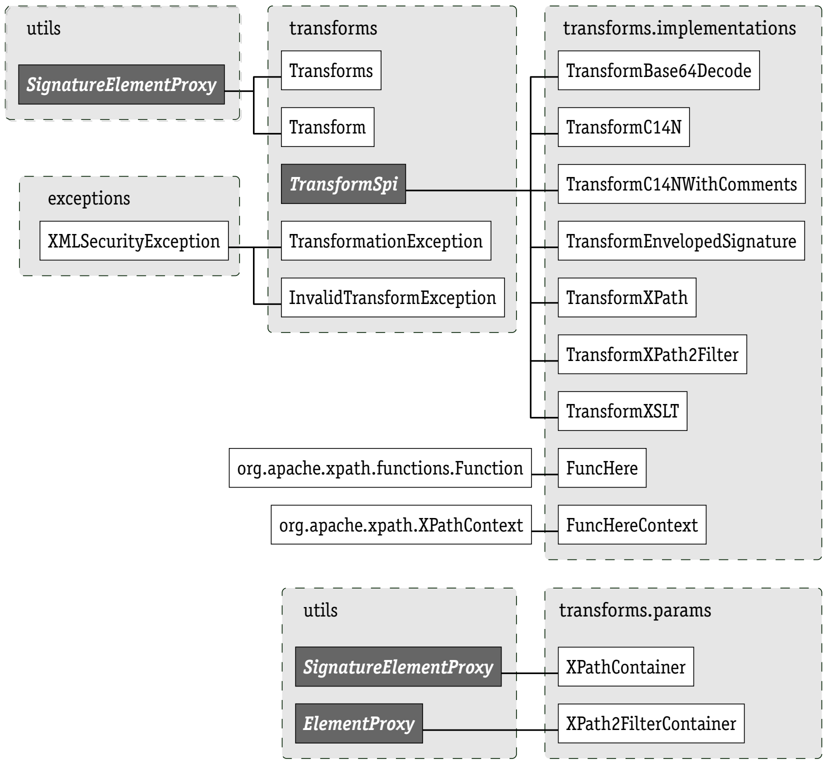


Figure A-11: The org.apache.xml.security.transforms.* package

The transform package contains the objects which model the XML Signature related transformations:

- ❑ The Transforms class handles `<ds:Transforms>` elements.

- The Transform class handles `<ds:Transform>` elements. Additionally, it's a factory and proxy for TransformSpi implementations.
- The TransformationException is thrown in case of an error during the transformation process.
- The InvalidTransformException is thrown if the Transform factory is requested to instantiate a non-registered or otherwise not available transform.
- The TransformSpi class is an abstract class which is extended by the transformation implementations.

The transform.implementations package contains the individual implementations of the transformations:

- The TransformBase64Decode transform implements "http://www.w3.org/2000/09/xmldsig#base64" which is decoding of base64 encoded data to an octet stream.
- The TransformC14N transform implements "http://www.w3.org/TR/2001/REC-xml-c14n-20010315" which is canonicalization (omitting comments) according to [Boy01].
- The TransformC14NWithComments transform implements "http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments" which is canonicalization (including comments) according to [Boy01].
- The TransformXPath transform implements "http://www.w3.org/TR/1999/REC-xpath-19991116" which is XPath based nodeset filtering.
  - The transforms.params.XPathContainer class is needed for initializing this transform.
- The TransformXPath2Filter transform implements [BHR02] which is a new XPath filter based on subtree operations.
  - The transforms.params.XPath2FilterContainer class is needed for initializing this transform.
- The TransformXSLT transform implements XSL transformations. This is transformation of one XML tree structure into another XML tree structure based on an eXtensible Stylesheet Language style sheet.
- The FuncHere class implements the `here()` function as defined by the XML Signature recommendation. It extends the org.apache.xpath.functions.Function class so that it can be used in Xalan.
- The FuncHereContext is based on the org.apache.xpath.XPathContext with the difference that it's constructors can be supplied a DTMManager. The re-use is necessary so that subsequent XPath transforms can use the same DTMManager for evaluating XPath expressions[1].

A.3.9 org.apache.xml.security.utils.* package

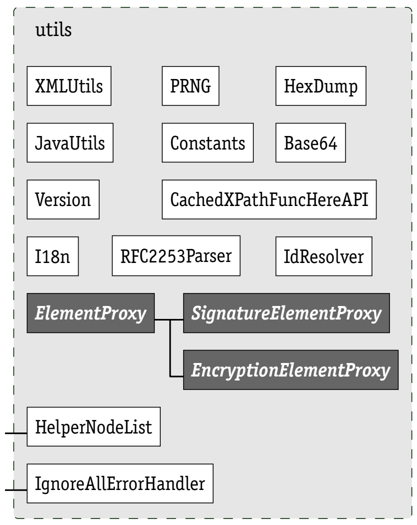


Figure A-12: The org.apache.xml.security.utils.* package

The utils package contains the various utility classes:

- ❏ The XMLUtils class offers static XML-related utility methods. This is mainly needed for functionality which is not offered by the DOM standard.
- ❏ The JavaUtils class offers a few static Java-related utility methods.
- ❏ The PRNG class is a singleton implementation which encapsulates a java.lang.SecureRandom object. Creation of such objects is a costly process, and this mechanism allows various classes in the library to use an existing PRNG.
- ❏ The HexDump class converts Strings containing hexadecimal digits into octet arrays and vice versa.
- ❏ The Base64 class converts Strings containing radix-64 (base64) encoded data into octet arrays and vice versa.
- ❏ The Constants class is a pool for various constants which are used throughout the library.
- ❏ The Version class exports the current version of the library (and also version numbers for Xerces and Xalan).
- ❏ The I18n class is used for internationalization of the software package. Error and exception messages are converted by this class.

1. The here() function is utilized through use of the utils.XPathFuncHereAPI

- The RFC2253Parser is a utility class for converting UTF-8 data into RFC2253 compliant strings and vice versa.
- The HelperNodeList class implements the NodeList interface and is able to be filled with own nodes, i.e. it provides an appendChild(Node) method.
- The IgnoreAllErrorHandler class is an ErrorHandler which silently discards all occuring SAXParseExceptions which are maybe thrown during a parsing run.
- The CachedXPathFuncHereAPI class is analogous to the org.apache.xpath.CachedXPathAPI with a single difference: the String containing the XPath which is to be evaluated must be supplied in a Text or Attribute Node. This enables the class to handle the `here()` function appropriately.
- The IdResolver class helps for resolving ID names to the appropriate elements. During building a new DOM tree from scratch, the Document object does not support the Document.getElementById(String ID) method; IDs can only be resolved by the Document if the underlying structure has been read by a validating parser, so this doesn't help during the construction of the DOM and therefore during the signing process[1]. The IdResolver utilizes internal knowledge to resolve particular ID type attributes: e.g. all "`Id`" attributes with their owner element in the XML Signature namespace are of type ID. So the IdResolver subsequently queries methods which are customized for specific XML Schemas so that they know (even without validating parsing), if their respective elements have the ID in question. If a match is found, the appropriate element is returned. Additionally, the ID is registered in the underlying Document so that subsequent calls are directly handled by the Document itself.
- The ElementProxy class is the abstract base class for all types of classes which directly map to a particular element. These classes have the property that they can be constructed from an existing element (needed for creating the class from an existing structure, e.g. during signature validation) or they can be constructed from a Document (the factory for creating the real DOM objects) and some additional parameters which contain the information.
- The SignatureElementProxy class is the abstract base class for all classes which model Elements from the XML Signature namespace.
- The EncryptionElementProxy class is the abstract base class for all classes which model Elements from the XML Encryption namespace.

1. A summary on ID-related issues can be found online in [Dodds01]

A.3.10 org.apache.xml.security.utils.resolver.**.* package

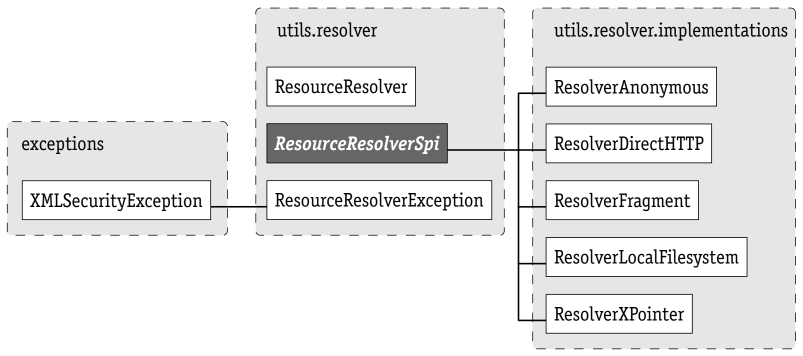


Figure A-13: The org.apache.xml.security.utils.resolver.**.* package

The utils.resolvers package contains a framework for resolving arbitrary information which is normally identified by a given URI. This is usually needed for de-referencing contents from a `ds:Reference/@URI` attribute or an `xenc:CipherReference/@URI` attribute.

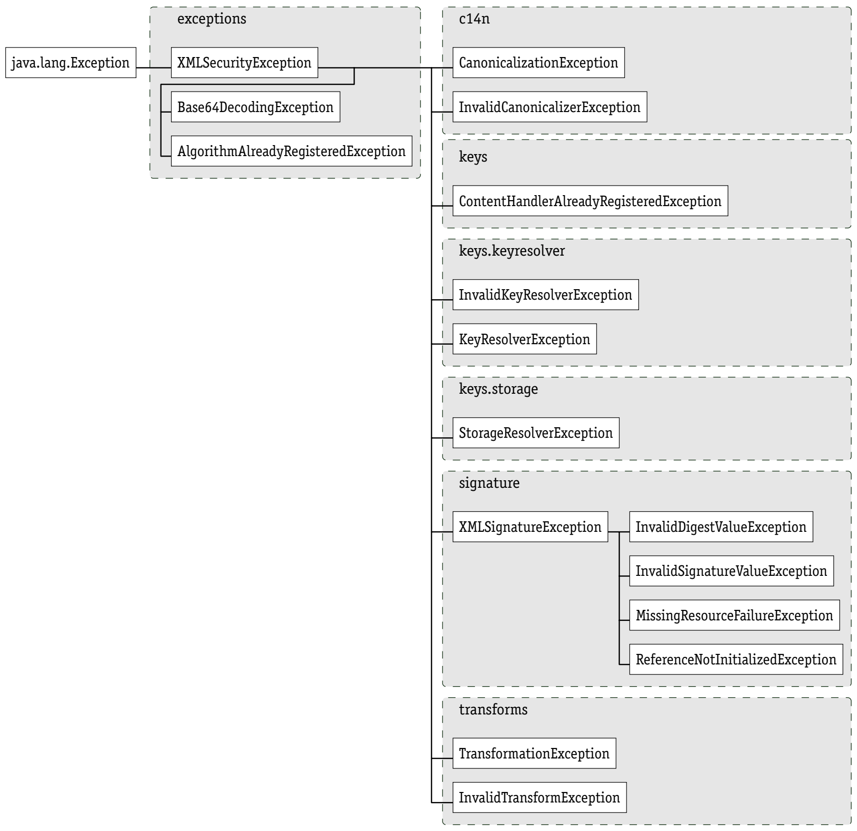
- ❑ The ResourceResolver is the interface to the different resolver implementations.
- ❑ The ResourceResolverException is thrown if something goes wrong during fetching the referenced contents.
- ❑ The ResourceResolverSpi is the abstract service provider interface class which all implementations must extend.

The utils.resolver.implementations package contains the individual implementations of the resource resolvers:

- ❑ The ResolverFragment resolves same-document URIs without comments. The `@URI=""` and `@URI="#someId"` are handled by this resolver. The `@URI=""` refers to all nodes in the document except the comment nodes, the `@URI="#someId"` refers to the Element with an ID type attribute with value "`someId`" and all it's descendants (without comments).
- ❑ The ResolverXPointer is the “with comments” version of the ResolverFragment Resolver. It handles the same-document URIs with comments. The `@URI="#xpointer(/)"` and `@URI="#xpointer(id('someId'))"` are handled by this resolver. The `@URI="#xpointer(/)"` refers to all nodes in the document including the comment nodes, the `@URI="#xpointer(id('someId'))"` refers to the Element with an ID type attribute with value "`someId`" and all it's descendants (including the comments).
- ❑ The ResolverDirectHTTP is capable to de-reference contents which are available via a HTTP connection. It can be configured to use an existing HTTP Proxy and can do proxy and server authentication of the client.

- ❏ The ResolverLocalFilesystem provides access to files which reside in the local file system.
- ❏ The ResolverAnonymous is an exceptional resolver, because it resolves non-existing @URI attributes: The XML Signature recommendation defines the ds:Reference/@URI attribute as optional and allows that one single Reference in a SignedInfo is without an @URI attribute. This can be used if the Signature is bound to a pre-defined piece of information which either cannot or should not be identified using the URI. The application has to initialize the ResolverAnonymous with an InputStream so that the de-referenced contents can be fetched from that given InputStream.

A.3.11 exceptions of the org.apache.xml.security hierarchy



java.lang.Exception
exceptions
XMLSecurityException
Base64DecodingException
AlgorithmAlreadyRegisteredException
c14n
CanonicalizationException
InvalidCanonicalizerException
keys
ContentHandlerAlreadyRegisteredException
keys.keyresolver
InvalidKeyResolverException
KeyResolverException
keys.storage
StorageResolverException
signature
XMLSignatureException
InvalidDigestValueException
InvalidSignatureValueException
MissingResourceFailureException
ReferenceNotInitializedException
transforms
TransformationException
InvalidTransformException

Figure A-14: The exceptions of the org.apache.xml.security hierarchy

Figure A-14 gives an overview to the exceptions in the complete library.

References

And01 Ross Anderson, “*Security Engineering: A Guide to Building Sependable Distributed Systems*”, John Wiley & Sons, 2001

ANS98 Ross Anderson, Roger Needham and Adi Shamir, “*The Steganographic File System*”, Information Hiding 1998, LNCS 1525, pp. 73-82, 1998
`http://www.cl.cam.ac.uk/ftp/users/rja14/sfs3.ps.gz`
`http://link.springer.de/link/service/series/0558/papers/1525/15250073.pdf`

BBC+ E. Bertino, M. Braun, S. Castano, E. Ferrari and M. Mesiti, “*Author-X: a Java-Based System for XML Data Protection*”

BCF01 Elisa Bertino, Silvana Castano and Elena Ferrari, “*Securing XML Documents with Author-X*”, IEEE Internet Computing, May 2001, pp. 21-31

BCFM00 Elisa Bertino, Silvana Castano, Elena Ferrari and Marco Mesiti, “*Specifying and Enforcing Access Control Policies for XML Document Sources*”, World Wide Web Journal (Baltzer Publ.), Vol.3, No.3, 2000.

BCFM99 Elisa Bertino, Silvana Castano, Elena Ferrari and Marco Mesiti, “*Controlled Access and Dissemination of XML Documents*”, ACM CIKM'99 2nd Workshop on Web Information and Data Management (WIDM'99), November 1999

BER02 W3C, John Boyer, Donald E. Eastlake and Joseph Reagle, “*Exclusive XML Canonicalization, Version 1.0*”, W3C Recommendation 18 July 2002,
`http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/`

BFM98 T. Berners-Lee, R. Fielding and L. Masinter, “*RFC 2396 - Uniform Resource Identifiers (URI): Generic Syntax*”, August 1998
`http://www.ietf.org/rfc/rfc2396.txt`

BHL99 W3C, Tim Bray, Dave Hollander and Andrew Layman (Editoren), “*Namespaces in XML*”, Januar 1999,
`http://www.w3.org/TR/1999/REC-xml-names-19990114`

BHLT02 W3C, Tim Bray, Dave Hollander, Andrew Layman and Richard Tobin (Editors), "*Namespaces in XML 1.1*", Working Draft, April 2002, `http://www.w3.org/TR/2002/WD-xml-names11-20020403/`

BHR02 W3C, John Boyer, Merlin Hughes and Joseph Reagle, "XML-Signature XPath Filter 2.0", W3C Recommendation 8 November 2002, `http://www.w3.org/TR/2002/REC-xmldsig-filter2-20021108/`

Bos99 W3C, Bert Bos, "*XML in 10 points*", March 1999, `http://www.w3.org/XML/1999/XML-in-10-points`

Boy01 W3C, John Boyer (Editor), "*Canonical XML Version 1.0*", W3C Recommendation 15 March 2001 `http://www.w3.org/TR/2001/REC-xml-c14n-20010315`

BPMM+00 W3C, Tim Bray, Jean Paoli, C. M. Sperberg-McQueen and Eve Maler (Editors), "*Extensible Markup Language (XML) 1.0 (Second Edition)*", W3C Recommendation 6 October 2000, `http://www.w3.org/TR/2000/REC-xml-20001006`

Bro02 David Brownell, "*SAX2*", O'Reilly, January 2002, ISBN 0-596-00237-8 `http://www.oreilly.com/catalog/sax2/` `http://sax.sourceforge.net/?selected=event`

BSI97 Bundesamt für Sicherheit in der Informationstechnik, „Mit Sicherheit in die Informationsgesellschaft", SecuMedia Verlag Ingelheim, 1997, ISBN 3-922746-29-2

CDFT98 J. Callas, L. Donnerhacke, H. Finney and R. Thayer, "*OpenPGP Message Format*", 1998, `http://www.ietf.org/rfc/rfc2440.txt`

Cel00 Joe Celko, "*Trees in SQL*", Oktober 2000 `http://www.intelligententerprise.com/001020/celko1_1.shtml`

Clark99 James Clark, "*XSL Transformations (XSLT), Version 1.0*", W3C Recommendation 16 November 1999, `http://www.w3.org/TR/1999/REC-xslt-19991116`

ClDe99 W3C, James Clark and Steve DeRose, "*XML Path Language (XPath) Version 1.0*", W3C Recommendation 16 November 1999, `http://www.w3.org/TR/1999/REC-xpath-19991116`

CoTo01	W3C, JOHN COWAN and RICHARD TOBIN (Editors), "*XML Information Set*", W3C Recommendation 24 October 2001, `http://www.w3.org/TR/2001/REC-xml-infoset-20011024`
DiAl99	T. DIERKS and C. ALLEN, "*The TLS Protocol Version 1.0*", January 1999, `http://www.ietf.org/rfc/rfc2246.txt`
Dodds01	LEIGH DODDS, "*Identity Crisis*", November 2001, `http://www.xml.com/pub/a/2001/11/07/id.html`
DVPS00	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`
DVPS01	E. DAMIANI, S. DE CAPITANI DI VIMERCATI, S. PARABOSCHI and P. SAMARATI, "*Controlling Access to XML Documents*", IEEE Internet Computing, November 2001, pp. 18-28
DVPS02	E. DAMIANI, S. DE CAPITANI DI VIMERCATE, S. PARABOSCHI and P. SAMARATI, "*Securing SOAP e-Services*", International Journal of Information Security (IJIS), 2002
EFLR+99	CARL M. ELLISON, BILL FRANTZ, BUTLER LAMPSON, RON RIVEST, BRIAN THOMAS and TATU YLONEN, "*SPKI Certificate Theory*", Request for Comments RFC 2693, 1999 `http://www.ietf.org/rfc/rfc2693.txt`
ERS02	W3C, DONALD EASTLAKE, JOSEPH REAGLE and DAVID SOLO (Editors), MARK BARTEL, JOHN BOYER, BARB FOX, BRIAN LAMACCHIA and ED SIMON, "*XML-Signature Syntax and Processing*", W3C Recommendation 12. February 2002, `http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/`
ER02	W3C, DONALD EASTLAKE AND JOSEPH REAGLE (Editors), TAKESHI IMAMURA, BLAIR DILLAWAY and ED SIMON, "*XML Encryption Syntax and Processing*", W3C Recommendation, 10. December 2002, `http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/`

FIPS180-1	U.S. Department of Commerce, National Institute of Standards and Technology, *"FIPS PUB 180-1: Secure Hash Standard (SHS)"*, April 1995
FIPS186-2	U.S. Department of Commerce, National Institute of Standards and Technology, *"FIPS PUB 186-2: Digital Signature Standard (DSS)"*, `http://csrc.nist.gov/publications/fips/fips186-2/fips186-2.pdf`
FrBo96	N. Freed and N. Borenstein, *"Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies"*, RFC 2045, November 1996 `http://www.ietf.org/rfc/rfc2045.txt`
FraTs02	Barbara Fraser and Theodore Ts'o, *"Description of the IP Security Working Group"*, February 2002, `http://www.ietf.org/html.charters/ipsec-charter.html`
GeRu99	Christian Geuer-Pollmann and Christoph Ruland, *„Das Simple Signature Protocol für WWW-Sicherheit"*, pp. 461-465, in: Bundesamt für Sicherheit in der Informationstechnik, *„IT-Sicherheit ohne Grenzen?"*, SecuMedia Verlag Ingelheim, 1999, ISBN 3-922746-32-2
Geu99	Christian Geuer-Pollmann, *„Das Simple Signature Protocol für WWW-Sicherheit"*, pp. 67-80, in: Rainer Baumgart, Kai Rannenberg and Gerhard Weck, *„IT-Sicherheit an der Schwelle des neuen Jahrtausends"*, Verlag Vieweg, Braunschweig, 1999, ISBN 3-528-05728-9
Geu02	Christian Geuer-Pollmann, *"XML Pool Encryption"*, Proceedings of the 2002 ACM Workshop on XML Security, Fairfax, VA, U.S.A, 2002 `http://doi.acm.org/10.1145/764792.764794`
GoMo02	Simon Godik and Tim Moses, *"XACML 1.0 - The OASIS extensible Access Control Markup Language (XACML)"*, Committee Specifications, 7 November 2002, `http://www.oasis-open.org/committees/xacml/` `http://www.oasis-open.org/committees/xacml/repository/cs-xacml-core-01.doc`
GRSM00	Christian Geuer-Pollmann, Christoph Ruland, Panagiotis Sklavos and Marina Moula, *"Digital Signatures for Web Contents"*, pp. 218-224, in Brian Stanfort-Smith and Paul T. Kidd, *"E-Business: Key Issues, Applications and Technologies"*, Proceedings of e2000, IOS Press, 2000, ISBN 1-58603-089-2

GroVei01	PAUL GROSSO and DANIEL VEILLARD, "*XML Fragment Interchange*", W3C Candidate Recommendation, 12 February 2001 `http://www.w3.org/TR/2001/CR-xml-fragment-20010212`
GrPo98	RÜDIGER GRIMM and ULRICH PORDESCH, „*Wie sicher ist die digitale Signatur*", Spektrum der Wissenschaft: „Dossier – Die Welt im Internet", 1998, pp 62-62
GrWä99	RÜDIGER GRIMM and JÜRGEN WÄSCH, „*XML und IT-Sicherheit*", pp. 141-162, in: RAINER BAUMGART, KAI RANNENBERG and GERHARD WECK, „*IT-Sicherheit an der Schwelle des neuen Jahrtausends*", Verlag Vieweg, Braunschweig, 1999, ISBN 3-528-05728-9
Hal02	PHILLIP HALLAM-BAKER, "*XML Key Management Specification (XKMS 2.0)*", W3C Editors Copy, October 2002 `http://www.w3c.org/2001/XKMS/Drafts/XKMS20021017/xkms-part-1.html`
HiVa01	MICHAEL HITCHENS and VIJAY VARADHARAJAN, "*RBAC for XML Document Stores*", pp. 131-143, in S. QUING, T. OKAMOTO and J. ZHOU (Editoren), ICICS 2001, LNCS 2229, Springer-Verlag, Heiderlberg, 2001
Hous99	RUSS HOUSLEY, "*Cryptographic Message Syntax*", June 1999, Request for Comments: 2630 `http://www.ietf.org/rfc/rfc2630.txt`
IEEE 754-1985	ANSI/IEEE STD 754-1985, "*Standard for Binary Floating-Point Arithmetic*", 1985, reaffirmed1990 `http://grouper.ieee.org/groups/754/`
ISO8879	ISO 8879, "*Information processing - Text and Office Systems - Standard Generalized Markup Language (SGML)*", 1986
ISO/IEC 9798-1	ISO/IEC 9798-1, "*Information technology — Security techniques — Entity authentication — Part 1: General*", 1997

ISO/IEC 10118-3	ISO/IEC 10118-3, "*Information technology - Security techniques — Hash-functions — Part 3: Dedicated hash-functions*", International Organization for Standardization, Geneva, Switzerland, 2003.
ISO/IEC 11770	ISO/IEC 11770, "*Information technology — Security techniques — Key management*"
ISO/IEC 18033-1	ISO/IEC 18033-1, "*Information technology — Security techniques — Encryption algorithms — Part 1: General*", 2002
ITU-T X.680 | ISO 8824	ITU-T Rec. X.680 | ISO/IEC 8824-1, "*Information technology — Abstract Syntax Notation One (ASN.1): Specification of basic notation*", 2002
ITU-T X.800 | ISO 7498-2	ITU-T Rec. X.800 | ISO 7498-2, "*Information processing systems — Open Systems Interconnection — Basic Reference Model — Part 2: Security architecture*", 1991
ITU-T X.810 | ISO 10181-1	ITU-T Rec X.810 | ISO/IEC 10181-1, "*Information technology — Security Frameworks in Open Systems: Overview*", 1996
ITU-T X.812 | ISO 10181-3	ITU-T Rec X.812 | ISO/IEC 10181-3, "*Information technology — Security Frameworks in Open Systems: Access control framework*", 1996
ITU-T X.813 | ISO 10181-4	ITU-T Rec X.813 | ISO/IEC 10181-4, "*Information technology — Security Frameworks in Open Systems: Non-repudiation framework*", 1996
ITU-T X.814 | ISO 10181-5	ITU-T Rec X.814 | ISO/IEC 10181-5, "*Information technology — Security Frameworks in Open Systems: Confidentiality framework*", 1996
KaSt98	BURT KALISKI and JESSICA STADDON, "PKCS #1: RSA Cryptography Specifications, Version 2.0", Request for Comments RFC 2437, 1997, `http://www.ietf.org/rfc/rfc2437.txt`
KoNe93	J. KOHL and C. NEUMAN, "*The Kerberos Network Authentication Service (V5)*", Request for Comments RFC 1510, 1993, `http://www.ietf.org/rfc/rfc1510.txt`
KrBeCa97	HUGO KRAWCZYK, MIHIR BELLARE and RAN CANETTI, "*HMAC: Keyed-Hashing for Message Authentication*", Request for Comments RFC 2104, 1997, `http://www.ietf.org/rfc/rfc2104.txt`

Kudoh02	OASIS, MICHIHARU KUDO, "*XACML Use Case for XML Fine-grained Access Control*", March 2002, `http://www.oasis-open.org/committees/xacml/docs/XACMLUseCaseXML.pdf`
LLN+02	W3C, ARNAUD LE HORS, PHILIPPE LE HÉGARET, GAVIN NICOL, LAUREN WOOD, MIKE CHAMPION and STEVE BYRNE, "*Document Object Model (DOM) Level 3 Core Specification*", W3C Working Draft 14 January 2002, `http://www.w3.org/TR/2002/WD-DOM-Level-3-Core-20020114`
MOV96	ALFRED MENEZES, PAUL VAN OORSCHOT and SCOTT VANSTONE, "*Handbook of applied cryptography*", CRC Press, 1996, ISBN 0-8493-8523-7
Ramos98	A. RAMOS, *"IETF Identification and Security Guidelines"*, Request for Comments RFC 2323, 1998, `http://www.ietf.org/rfc/rfc2323.txt`
Rams99	Blake Ramsdell, "S/MIME Version 3 Message Specification", Request for Comments RFC 2633, 1999, `http://www.ietf.org/rfc/rfc2633.txt`
Reagle02	W3C, JOSEPH REAGLE, "*XML Encryption Requirements*", Revision 1.20, `http://www.w3.org/TR/xml-encryption-req` `http://www.w3.org/Encryption/2001/Drafts/xml-encryption-req`
Resc99	E. RESCORLA, "*Diffie-Hellman Key Agreement Method*", Request For Comments RFC 2631, 1999 `http://www.ietf.org/rfc/rfc2631.txt`
Roe97	MICHAEL ROE, "*Cryptography and Evidence*", PhD thesis, Computer Laboratory, University of Cambridge, 1997, `http://research.microsoft.com/users/mroe/THESIS.PDF`
RSA78	R.L. RIVEST, A. SHAMIR and L.M. ADLEMAN, *"A method for obtaining digital signatures and public-key cryptosystems"*, Communications of the ACM, 1987
Schei01	KARL SCHEIBELHOFER, "*Signing XML Documents and the Concept of 'What You See Is What You Sign'*", Master thesis, IAIK, TU Graz, January 2001
Schn962	BRUCE SCHNEIER, "*Applied Cryptography*" (Second edition), John Wiley, 1996, ISBN 0-471-12845-7