XML Encryption Syntax and Processing Version 1.1

W3C Working Draft 26 February 2009

This version:
http://www.w3.org/TR/2009/WD-xmlenc-core1-20090226/
Latest version:
http://www.w3.org/TR/xmlenc-core1/
Previous version:
http://www.w3.org/TR/2002/REC-xmlenc-core-20021210/
Latest XML Encryption recommendation:
http://www.w3.org/TR/xmlenc-core/
Editors
Donald Eastlake <dee3@torque.pothole.com>
Joseph Reagle <reagle@w3.org>
Authors
Takeshi Imamura <IMAMU@jp.ibm.com>
Blair Dillaway <blaird@microsoft.com>
Ed Simon <edsimon@xmlsec.com>
Kelvin Yiu <kelviny@microsoft.com>
Contributors
See participants.

Abstract

This document specifies a process for encrypting data and representing the result in XML. The data may be arbitrary data (including an XML document), an XML element, or XML element content. The result of encrypting data is an XML Encryption element which contains or references the cipher data.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.

This is a First Public Working Draft of "XML Encryption 1.1."

At the time of this publication, the most recent W3C Recommendation of XML Encryption 1 is the 10 December 2002 XML Encryption Recommendation. A diff-marked version of this specification is available; it shows differences between the latest recommendation and this version of the specification.

Conformance-affecting changes against this previous recommendation mainly affect the set of mandatory to implement cryptographic algorithms, by adding Elliptic Curve Diffie-Hellman Key Agreement. There is currently no consensus about the inclusion of this algorithm as mandatory to implement, and the Working Group seeks early community input into what algorithms should be supported. Arguments for and against specific approaches are called out in an editorial note in section 5.1 Algorithm Identifiers and Implementation Requirements.

This document was developed by the XML Security Working Group. The Working Group expects to advance this Working Draft to Recommendation Status.

Please send comments about this document to public-xmlsec-comments@w3.org (with public archive).

Publication as a Working Draft does not imply endorsement by the W3C Membership. This is a draft document and may be updated, replaced or obsoleted by other documents at any time. It is inappropriate to cite this document as other than work in progress.

This document was produced by a group operating under the 5 February 2004 W3C Patent Policy. W3C maintains a public list of any patent disclosures made in connection with the deliverables of the group; that page also includes instructions for disclosing a patent. An individual who has actual knowledge of a patent which the individual believes contains Essential Claim(s) must disclose the information in accordance with section 6 of the W3C Patent Policy.

Table of Contents

---

1 Introduction

This document specifies a process for encrypting data and representing the result in XML. The data may be arbitrary data (including an XML document), an XML element, or XML element content. The result of encrypting data is an XML Encryption `EncryptedData` element which contains (via one of its children's content) or identifies (via a URI reference) the cipher data.

When encrypting an XML element or element content the `EncryptedData` element replaces the element or content (respectively) in the encrypted version of the XML document.

When encrypting arbitrary data (including entire XML documents), the `EncryptedData` element may become the root of a new XML document or become a child element in an application-chosen XML document.

1.1 Editorial and Conformance Conventions

This specification uses XML schemas [XML-schema] to describe the content model.

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this specification are to be interpreted as described in RFC2119 [KEYWORDS]:

> "they MUST only be used where it is actually required for interoperation or to limit behavior which has potential for causing harm (e.g., limiting retransmissions)"

Consequently, we use these capitalized keywords to unambiguously specify requirements over protocol and application features and behavior that affect the interoperability and security of implementations. These key words are not used (capitalized) to describe XML grammar; schema definitions unambiguously describe such requirements and we wish to reserve the prominence of

these terms for the natural language descriptions of protocols and features. For instance, an XML attribute might be described as being "optional." Compliance with the XML-namespace specification [XML-NS] is described as "REQUIRED."

1.2 Design Philosophy

The design philosophy and requirements of this specification (including the limitations related to instance validity) are addressed in the XML Encryption Requirements [EncReq].

1.3 Versions, Namespaces, URIs, and Identifiers

No provision is made for an explicit version number in this syntax. If a future version is needed, it will use a different namespace. The experimental XML namespace [XML-NS] URI that MUST be used by implementations of this (dated) specification is:

```
xmlns:xenc='http://www.w3.org/2001/04/xmlenc#'
```

This namespace is also used as the prefix for algorithm identifiers used by this specification. While applications MUST support XML and XML namespaces, the use of internal entities [XML, section 4.2.1], the "`xenc`" XML namespace prefix [XML-NS, section 2] and defaulting/scoping conventions are OPTIONAL; we use these facilities to provide compact and readable examples. Additionally, the entity `&xenc;` is defined so as to provide short-hand identifiers for URIs defined in this specification. For example "`&xenc;Element`" corresponds to "http://www.w3.org/2001/04/xmlenc#Element".

This specification makes use of the XML Signature [XML-DSIG] namespace and schema definitions

```
xmlns:ds='http://www.w3.org/2000/09/xmldsig#'
```

URIs [URI] MUST abide by the [XML-Schema] `anyURI` type definition and the [XML-DSIG, 4.3.3.1 The URI Attribute] specification (i.e., permitted characters, character escaping, scheme support, etc.).

1.4 Acknowledgements

The contributions of the following Working Group members to this specification are gratefully acknowledged in accordance with the contributor policies and the active WG roster.

- Joseph Ashwood
- Simon Blake-Wilson, Certicom
- Frank D. Cavallito, BEA Systems
- Eric Cohen, PricewaterhouseCoopers
- Blair Dillaway, Microsoft (Author)
- Blake Dournaee, RSA Security
- Donald Eastlake, Motorola (Editor)
- Barb Fox, Microsoft
- Christian Geuer-Pollmann, University of Siegen
- Tom Gindin, IBM
- Jiandong Guo, Phaos
- Phillip Hallam-Baker, Verisign
- Amir Herzberg, NewGenPay
- Merlin Hughes, Baltimore
- Frederick Hirsch
- Maryann Hondo, IBM
- Takeshi Imamura, IBM (Author)
- Mike Just, Entrust, Inc.

- Brian LaMacchia, Microsoft
- Hiroshi Maruyama, IBM
- John Messing, Law-on-Line
- Shivaram Mysore, Sun Microsystems
- Thane Plambeck, Verisign
- Joseph Reagle, W3C (Chair, Editor)
- Aleksey Sanin
- Jim Schaad, Soaring Hawk Consulting
- Ed Simon, XMLsec (Author)
- Daniel Toth, Ford
- Yongge Wang, Certicom
- Steve Wiley, myProof

Additionally, we thank the following for their comments during and subsequent to Last Call:

- Martin Dürst, W3C
- Dan Lanz, Zolera
- Susan Lesch, W3C
- David Orchard, BEA Systems
- Ronald Rivest, MIT

Contributions for version 1.1 were received from the members of the XML Security Working Group:

> ***TBD. See public list of participants for now.***

2 Encryption Overview and Examples (Non-normative)

This section provides an overview and examples of XML Encryption syntax. The formal syntax is found in *Encryption Syntax* (section 3); the specific processing is given in *Processing Rules* (section 4).

Expressed in shorthand form, the `EncryptedData` element has the following structure (where "?" denotes zero or one occurrence; "+" denotes one or more occurrences; "*" denotes zero or more occurrences; and the empty element tag means the element must be empty):

```
<EncryptedData Id? Type? MimeType? Encoding?>
  <EncryptionMethod/>?
  <ds:KeyInfo>
    <EncryptedKey>?
    <AgreementMethod>?
    <ds:KeyName>?
    <ds:RetrievalMethod>?
    <ds:*>?
  </ds:KeyInfo>?
  <CipherData>
    <CipherValue>?
    <CipherReference URI?>?
  </CipherData>
  <EncryptionProperties>?
</EncryptedData>
```

The `CipherData` element envelopes or references the raw encrypted data. If enveloping, the raw encrypted data is the `CipherValue` element's content; if referencing, the `CipherReference` element's `URI` attribute points to the location of the raw encrypted data

2.1 Encryption Granularity

Consider the following fictitious payment information, which includes identification information and information appropriate to a payment method (e.g., credit card, money transfer, or electronic check):

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <Number>4019 2445 0277 5567</Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

This markup represents that John Smith is using his credit card with a limit of $5,000USD.

2.1.1 Encrypting an XML Element

Smith's credit card number is sensitive information! If the application wishes to keep that information confidential, it can encrypt the `CreditCard` element:

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

By encrypting the entire `CreditCard` element from its start to end tags, the identity of the element itself is hidden. (An eavesdropper doesn't know whether he used a credit card or money transfer.) The `CipherData` element contains the encrypted serialization of the `CreditCard` element.

2.1.2 Encrypting XML Element Content (Elements)

As an alternative scenario, it may be useful for intermediate agents to know that John used a credit card with a particular limit, but not the card's number, issuer, and expiration date. In this case, the content (character data or children elements) of the `CreditCard` element is encrypted:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
    <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
     Type='http://www.w3.org/2001/04/xmlenc#Content'>
      <CipherData>
        <CipherValue>A23B45C56</CipherValue>
      </CipherData>
    </EncryptedData>
  </CreditCard>
</PaymentInfo>
```

2.1.3 Encrypting XML Element Content (Character Data)

Or, consider the scenario in which all the information *except* the actual credit card number can be in the clear, including the fact that the Number element exists:

```
<?xml version='1.0'?>
<PaymentInfo xmlns='http://example.org/paymentv2'>
  <Name>John Smith</Name>
  <CreditCard Limit='5,000' Currency='USD'>
```

```
    <Number>
      <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
       Type='http://www.w3.org/2001/04/xmlenc#Content'>
        <CipherData>
          <CipherValue>A23B45C56</CipherValue>
        </CipherData>
      </EncryptedData>
    </Number>
    <Issuer>Example Bank</Issuer>
    <Expiration>04/02</Expiration>
  </CreditCard>
</PaymentInfo>
```

Both `CreditCard` and `Number` are in the clear, but the character data content of `Number` is encrypted.

2.1.4 Encrypting Arbitrary Data and XML Documents

If the application scenario requires all of the information to be encrypted, the whole document is encrypted as an octet sequence. This applies to arbitrary data including XML documents.

```
<?xml version='1.0'?>
<EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
 MimeType='text/xml'>
  <CipherData>
    <CipherValue>A23B45C56</CipherValue>
  </CipherData>
</EncryptedData>
```

2.1.5 Super-Encryption: Encrypting EncryptedData

An XML document may contain zero or more `EncryptedData` elements. `EncryptedData` cannot be the parent or child of another `EncryptedData` element. However, the actual data encrypted can be anything, including `EncryptedData` and `EncryptedKey` elements (i.e., super-encryption). During super-encryption of an `EncryptedData` or `EncryptedKey` element, one must encrypt the entire element. Encrypting only the content of these elements, or encrypting selected child elements is an invalid instance under the provided schema.
For example, consider the following:

```
<pay:PaymentInfo xmlns:pay='http://example.org/paymentv2'>
  <EncryptedData Id='ED1' xmlns='http://www.w3.org/2001/04/xmlenc#'
   Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <CipherData>
      <CipherValue>originalEncryptedData</CipherValue>
    </CipherData>
  </EncryptedData>
</pay:PaymentInfo>
```

A valid super-encryption of "`//xenc:EncryptedData[@Id='ED1']`" would be:

```
<pay:PaymentInfo xmlns:pay='http://example.org/paymentv2'>
  <EncryptedData Id='ED2' xmlns='http://www.w3.org/2001/04/xmlenc#'
   Type='http://www.w3.org/2001/04/xmlenc#Element'>
    <CipherData>
      <CipherValue>newEncryptedData</CipherValue>
    </CipherData>
  </EncryptedData>
</pay:PaymentInfo>
```

where the `CipherValue` content of '`newEncryptedData`' is the base64 encoding of the encrypted octet sequence resulting from encrypting the `EncryptedData` element with `Id='ED1'`.

2.2 EncryptedData and EncryptedKey Usage

2.2.1 EncryptedData with Symmetric Key (KeyName)

```
[s1] <EncryptedData xmlns='http://www.w3.org/2001/04/xmlenc#'
      Type='http://www.w3.org/2001/04/xmlenc#Element'/>
[s2]   <EncryptionMethod
        Algorithm='http://www.w3.org/2001/04/xmlenc#tripledes-cbc'/>
[s3]   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
[s4]     <ds:KeyName>John Smith</ds:KeyName>
[s5]   </ds:KeyInfo>
[s6]   <CipherData><CipherValue>DEADBEEF</CipherValue></CipherData>
[s7] </EncryptedData>
```

[s1] The type of data encrypted may be represented as an attribute value to aid in decryption and subsequent processing. In this case, the data encrypted was an 'element'. Other alternatives include 'content' of an element, or an external octet sequence which can also be identified via the MimeType and Encoding attributes.

[s2] This (3DES CBC) is a symmetric key cipher.

[s4] The symmetric key has an associated name "John Smith".

[s6] CipherData contains a CipherValue, which is a base64 encoded octet sequence. Alternately, it could contain a CipherReference, which is a URI reference along with transforms necessary to obtain the encrypted data as an octet sequence

2.2.2 EncryptedKey (ReferenceList, ds:RetrievalMethod, CarriedKeyName)

The following EncryptedData structure is very similar to the one above, except this time the key is referenced using a ds:RetrievalMethod:

```
[t01] <EncryptedData Id='ED'
       xmlns='http://www.w3.org/2001/04/xmlenc#'>
[t02]   <EncryptionMethod
         Algorithm='http://www.w3.org/2001/04/xmlenc#aes128-cbc'/>
[t03]   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
[t04]     <ds:RetrievalMethod URI='#EK'
           Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"/>
[t05]     <ds:KeyName>Sally Doe</ds:KeyName>
[t06]   </ds:KeyInfo>
[t07]   <CipherData><CipherValue>DEADBEEF</CipherValue></CipherData>
[t08] </EncryptedData>
```

[t02] This (AES-128-CBC) is a symmetric key cipher.

[t04] ds:RetrievalMethod is used to indicate the location of a key with type &xenc;EncryptedKey. The (AES) key is located at '#EK'.

[t05] ds:KeyName provides an alternative method of identifying the key needed to decrypt the CipherData. Either or both the ds:KeyName and ds:KeyRetrievalMethod could be used to identify the same key.

Within the same XML document, there existed an EncryptedKey structure that was referenced within [t04]:

```
[t09] <EncryptedKey Id='EK' xmlns='http://www.w3.org/2001/04/xmlenc#'>
[t10]   <EncryptionMethod
         Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
```

```
[t11]   <ds:KeyInfo xmlns:ds='http://www.w3.org/2000/09/xmldsig#'>
[t12]     <ds:KeyName>John Smith</ds:KeyName>
[t13]   </ds:KeyInfo>
[t14]   <CipherData><CipherValue>xyzabc</CipherValue></CipherData>
[t15]   <ReferenceList>
[t16]     <DataReference URI='#ED'/>
[t17]   </ReferenceList>
[t18]   <CarriedKeyName>Sally Doe</CarriedKeyName>
[t19] </EncryptedKey>
```

`[t09]` The `EncryptedKey` element is similar to the `EncryptedData` element except that the data encrypted is always a key value.

`[t10]` The `EncryptionMethod` is the RSA public key algorithm.

`[t12]` `ds:KeyName` of "John Smith" is a property of the key necessary for decrypting (using RSA) the `CipherData`.

`[t14]` The `CipherData`'s `CipherValue` is an octet sequence that is processed (serialized, encrypted, and encoded) by a referring encrypted object's `EncryptionMethod`. (Note, an EncryptedKey's `EncryptionMethod` is the algorithm used to encrypt these octets and does not speak about what type of octets they are.)

`[t15-17]` A `ReferenceList` identifies the encrypted objects (`DataReference` and `KeyReference`) encrypted with this key. The `ReferenceList` contains a list of references to data encrypted by the symmetric key carried within this structure.

`[t18]` The `CarriedKeyName` element is used to identify the encrypted key value which may be referenced by the `KeyName` element in `ds:KeyInfo`. (Since ID attribute values must be unique to a document,`CarriedKeyName` can indicate that several `EncryptedKey` structures contain the same key value encrypted for different recipients.)

3 Encryption Syntax

This section provides a detailed description of the syntax and features for XML Encryption. Features described in this section MUST be implemented unless otherwise noted. The syntax is defined via [XML-Schema] with the following XML preamble, declaration, internal entity, and import:

```
Schema Definition:

<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE schema  PUBLIC "-//W3C//DTD XMLSchema 200102//EN"
 "http://www.w3.org/2001/XMLSchema.dtd"
 [
   <!ATTLIST schema
     xmlns:xenc CDATA #FIXED 'http://www.w3.org/2001/04/xmlenc#'
     xmlns:ds CDATA #FIXED 'http://www.w3.org/2000/09/xmldsig#'>
   <!ENTITY xenc 'http://www.w3.org/2001/04/xmlenc#'>
   <!ENTITY % p ''>
   <!ENTITY % s ''>
  ]>

<schema xmlns='http://www.w3.org/2001/XMLSchema' version='1.0'
        xmlns:ds='http://www.w3.org/2000/09/xmldsig#'
        xmlns:xenc='http://www.w3.org/2001/04/xmlenc#'
        targetNamespace='http://www.w3.org/2001/04/xmlenc#'
        elementFormDefault='qualified'>

  <import namespace='http://www.w3.org/2000/09/xmldsig#'
          schemaLocation='http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/xmldsig-core-s
```

3.1 The `EncryptedType` Element

`EncryptedType` is the abstract type from which `EncryptedData` and `EncryptedKey` are derived. While these two latter element types are very similar with respect to their content models, a syntactical distinction is useful to processing. Implementation MUST generate laxly schema valid [XML-schema] `EncryptedData` or `EncryptedKey` as specified by the subsequent schema declarations. (Note the laxly schema valid generation means that the content permitted by `xsd:ANY` need not be valid.) Implementations SHOULD create these XML structures (`EncryptedType` elements and their descendants/content) in Normalization Form C [NFC, NFC-Corrigendum].

```
Schema Definition:

<complexType name='EncryptedType' abstract='true'>
  <sequence>
    <element name='EncryptionMethod' type='xenc:EncryptionMethodType'
             minOccurs='0'/>
    <element ref='ds:KeyInfo' minOccurs='0'/>
    <element ref='xenc:CipherData'/>
    <element ref='xenc:EncryptionProperties' minOccurs='0'/>
  </sequence>
  <attribute name='Id' type='ID' use='optional'/>
  <attribute name='Type' type='anyURI' use='optional'/>
  <attribute name='MimeType' type='string' use='optional'/>
  <attribute name='Encoding' type='anyURI' use='optional'/>
 </complexType>
```

`EncryptionMethod` is an optional element that describes the encryption algorithm applied to the cipher data. If the element is absent, the encryption algorithm must be known by the recipient or the decryption will fail.

`ds:KeyInfo` is an optional element, defined by [XML-DSIG], that carries information about the key used to encrypt the data. Subsequent sections of this specification define new elements that may appear as children of `ds:KeyInfo`.

`CipherData` is a mandatory element that contains the `CipherValue` or `CipherReference` with the encrypted data.

`EncryptionProperties` can contain additional information concerning the generation of the `EncryptedType` (e.g., date/time stamp).

`Id` is an optional attribute providing for the standard method of assigning a string id to the element within the document context.

`Type` is an optional attribute identifying type information about the plaintext form of the encrypted content. While optional, this specification takes advantage of it for mandatory processing described in *Processing Rules: Decryption* (section 4.2). If the `EncryptedData` element contains data of `Type` 'element' or element 'content', and replaces that data in an XML document context, it is strongly recommended the `Type` attribute be provided. Without this information, the decryptor will be unable to automatically restore the XML document to its original cleartext form.

`MimeType` is an optional (advisory) attribute which describes the media type of the data which has been encrypted. The value of this attribute is a string with values defined by [MIME]. For example, if the data that is encrypted is a base64 encoded PNG, the transfer `Encoding` may be specified as 'http://www.w3.org/2000/09/xmldsig#base64' and the `MimeType` as 'image/png'. This attribute is purely advisory; no validation of the `MimeType` information is required and it does not indicate the encryption application must do any additional processing. Note, this information may not be necessary if it is already bound to the identifier in the `Type` attribute. For example, the Element and Content types defined in this specification are always UTF-8 encoded text.

3.2 The EncryptionMethod Element

EncryptionMethod is an optional element that describes the encryption algorithm applied to the cipher data. If the element is absent, the encryption algorithm must be known by the recipient or the decryption will fail.

```
Schema Definition:

<complexType name='EncryptionMethodType' mixed='true'>
  <sequence>
    <element name='KeySize' minOccurs='0' type='xenc:KeySizeType'/>
    <element name='OAEPparams' minOccurs='0' type='base64Binary'/>
    <any namespace='##other' minOccurs='0' maxOccurs='unbounded'/>
  </sequence>
  <attribute name='Algorithm' type='anyURI' use='required'/>
</complexType>
```

The permitted child elements of the `EncryptionMethod` are determined by the specific value of the `Algorithm` attribute URI, and the `KeySize` child element is always permitted. For example, the RSA-OAEP algorithm (section 5.4.2) uses the `ds:DigestMethod` and `OAEPparams` elements. (We rely upon the `ANY` schema construct because it is not possible to specify element content based on the value of an attribute.)

The presence of any child element under `EncryptionMethod` which is not permitted by the algorithm or the presence of a `KeySize` child inconsistent with the algorithm MUST be treated as an error. (All algorithm URIs specified in this document imply a key size but this is not true in general. Most popular stream cipher algorithms take variable size keys.)

3.3 The `CipherData` Element

The `CipherData` is a mandatory element that provides the encrypted data. It must either contain the encrypted octet sequence as base64 encoded text of the `CipherValue` element, or provide a reference to an external location containing the encrypted octet sequence via the `CipherReference` element.

```
Schema Definition:

<element name='CipherData' type='xenc:CipherDataType'/>
<complexType name='CipherDataType'>
   <choice>
     <element name='CipherValue' type='base64Binary'/>
     <element ref='xenc:CipherReference'/>
   </choice>
 </complexType>
```

3.3.1 The `CipherReference` Element

If `CipherValue` is not supplied directly, the `CipherReference` identifies a source which, when processed, yields the encrypted octet sequence.

The actual value is obtained as follows. The `CipherReference` `URI` contains an identifier that is dereferenced. Should the `CipherReference` element contain an OPTIONAL sequence of `Transforms`, the data resulting from dereferencing the URI is transformed as specified so as to yield the intended cipher value. For example, if the value is base64 encoded within an XML document; the transforms could specify an XPath expression followed by a base64 decoding so as to extract the octets.

The syntax of the `URI` and `Transforms` is similar to that of [XML-DSIG]. However, there is a difference between signature and encryption processing. In [XML-DSIG] both generation and validation processing start with the same source data and perform that transform in the same order. In encryption, the decryptor has only the cipher data and the specified transforms are enumerated for

the decryptor, in the order necessary to obtain the octets. Consequently, because it has different semantics `Transforms` is in the `&xenc;` namespace.

For example, if the relevant cipher value is captured within a `CipherValue` element within a different XML document, the `CipherReference` might look as follows:

```
<CipherReference URI="http://www.example.com/CipherValues.xml">
  <Transforms>
    <ds:Transform
     Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
        <ds:XPath xmlns:rep="http://www.example.org/repository">
          self::text()[parent::rep:CipherValue[@Id="example1"]]
        </ds:XPath>
    </ds:Transform>
    <ds:Transform Algorithm="http://www.w3.org/2000/09/xmldsig#base64"/>
  </Transforms>
</CipherReference>
```

Implementations MUST support the `CipherReference` feature and the same URI encoding, dereferencing, scheme, and HTTP response codes as that of [XML-DSIG]. The `Transform` feature and particular transform algorithms are OPTIONAL.

```
Schema Definition:

<element name='CipherReference' type='xenc:CipherReferenceType'/>
 <complexType name='CipherReferenceType'>
     <sequence>
       <element name='Transforms' type='xenc:TransformsType' minOccurs='0'/>
     </sequence>
     <attribute name='URI' type='anyURI' use='required'/>
 </complexType>

  <complexType name='TransformsType'>
     <sequence>
       <element ref='ds:Transform' maxOccurs='unbounded'/>
     </sequence>
   </complexType>
```

3.4 The `EncryptedData` Element

The `EncryptedData` element is the core element in the syntax. Not only does its `CipherData` child contain the encrypted data, but it's also the element that replaces the encrypted element, or serves as the new document root.

```
Schema Definition:

<element name='EncryptedData' type='xenc:EncryptedDataType'/>
<complexType name='EncryptedDataType'>
  <complexContent>
   <extension base='xenc:EncryptedType'>
   </extension>
  </complexContent>
</complexType>
```

3.5 Extensions to `ds:KeyInfo` Element

There are three ways that the keying material needed to decrypt `CipherData` can be provided:

1. The `EncryptedData` or `EncryptedKey` element specify the associated keying material via a child of `ds:KeyInfo`. All of the child elements of ds:`KeyInfo` specified in [XML-DSIG] MAY be used as qualified:
    1. Support for `ds:KeyValue` is OPTIONAL and may be used to transport public keys, such as

Diffie-Hellman Key Values (section 5.5.1). (Including the plaintext decryption key, whether a private key or a secret key, is obviously NOT RECOMMENDED.)
2. Support of `ds:KeyName` to refer to an `EncryptedKey` `CarriedKeyName` is RECOMMENDED.
3. Support for same document `ds:RetrievalMethod` is REQUIRED.

In addition, we provide two additional child elements: applications MUST support `EncryptedKey` (section 3.5.1) and MAY support `AgreementMethod` (section 5.5).

2. A detached (not inside `ds:KeyInfo`) `EncryptedKey` element can specify the `EncryptedData` or `EncryptedKey` to which its decrypted key will apply via a `DataReference` or `KeyReference` (section 3.6).
3. The keying material can be determined by the recipient by application context and thus need not be explicitly mentioned in the transmitted XML.

3.5.1 The `EncryptedKey` Element

Identifer

`Type="http://www.w3.org/2001/04/xmlenc#EncryptedKey"`

(This can be used within a `ds:RetrievalMethod` element to identify the referent's type.)

The `EncryptedKey` element is used to transport encryption keys from the originator to a known recipient(s). It may be used as a stand-alone XML document, be placed within an application document, or appear inside an `EncryptedData` element as a child of a `ds:KeyInfo` element. The key value is always encrypted to the recipient(s). When `EncryptedKey` is decrypted the resulting octets are made available to the `EncryptionMethod` algorithm without any additional processing.

```
Schema Definition:

<element name='EncryptedKey' type='xenc:EncryptedKeyType'/>
<complexType name='EncryptedKeyType'>
  <complexContent>
    <extension base='xenc:EncryptedType'>
      <sequence>
        <element ref='xenc:ReferenceList' minOccurs='0'/>
        <element name='CarriedKeyName' type='string' minOccurs='0'/>
      </sequence>
      <attribute name='Recipient' type='string' use='optional'/>
    </extension>
  </complexContent>
</complexType>
```

`ReferenceList` is an optional element containing pointers to data and keys encrypted using this key. The reference list may contain multiple references to `EncryptedKey` and `EncryptedData` elements. This is done using `KeyReference` and `DataReference` elements respectively. These are defined below.

`CarriedKeyName` is an optional element for associating a user readable name with the key value. This may then be used to reference the key using the `ds:KeyName` element within `ds:KeyInfo`. The same `CarriedKeyName` label, unlike an ID type, may occur multiple times within a single document. The value of the key is to be the same in all `EncryptedKey` elements identified with the same `CarriedKeyName` label within a single XML document. Note that because whitespace is significant in the value of the `ds:KeyName` element, whitespace is also significant in the value of the `CarriedKeyName` element.

`Recipient` is an optional attribute that contains a hint as to which recipient this encrypted key value is intended for. Its contents are application dependent.

The `Type` attribute inherited from `EncryptedType` can be used to further specify the type of the encrypted key if the `EncryptionMethod` `Algorithm` does not define a unambiguous encoding/representation. (Note, all the algorithms in this specification have an unambiguous

representation for their associated key structures.)

3.5.2 The `ds:RetrievalMethod` Element

The `ds:RetrievalMethod` [XML-DSIG] with a `Type` of '`http://www.w3.org/2001/04/xmlenc#EncryptedKey`' provides a way to express a link to an `EncryptedKey` element containing the key needed to decrypt the `CipherData` associated with an `EncryptedData` or `EncryptedKey` element. The `ds:RetrievalMethod` with this type is always a child of the `ds:KeyInfo` element and may appear multiple times. If there is more than one instance of a `ds:RetrievalMethod` in a `ds:KeyInfo` of this type, then the `EncryptedKey` objects referred to must contain the same key value, possibly encrypted in different ways or for different recipients.

```
Schema Definition:

<!--
    <attribute name='Type' type='anyURI' use='optional'
    fixed='http://www.w3.org/2001/04/xmlenc#EncryptedKey' />
-->
```

3.6 The `ReferenceList` Element

`ReferenceList` is an element that contains pointers from a key value of an `EncryptedKey` to items encrypted by that key value (`EncryptedData` or `EncryptedKey` elements).

```
Schema Definition:

<element name='ReferenceList'>
  <complexType>
    <choice minOccurs='1' maxOccurs='unbounded'>
      <element name='DataReference' type='xenc:ReferenceType'/>
      <element name='KeyReference' type='xenc:ReferenceType'/>
    </choice>
  </complexType>
</element>

<complexType name='ReferenceType'>
  <sequence>
    <any namespace='##other' minOccurs='0' maxOccurs='unbounded'/>
  </sequence>
  <attribute name='URI' type='anyURI' use='required'/>
</complexType>
```

`DataReference` elements are used to refer to `EncryptedData` elements that were encrypted using the key defined in the enclosing `EncryptedKey` element. Multiple `DataReference` elements can occur if multiple `EncryptedData` elements exist that are encrypted by the same key.

`KeyReference` elements are used to refer to `EncryptedKey` elements that were encrypted using the key defined in the enclosing `EncryptedKey` element. Multiple `KeyReference` elements can occur if multiple `EncryptedKey` elements exist that are encrypted by the same key.

For both types of references one may optionally specify child elements to aid the recipient in retrieving the `EncryptedKey` and/or `EncryptedData` elements. These could include information such as XPath transforms, decompression transforms, or information on how to retrieve the elements from a document storage facility. For example:

```
<ReferenceList>
  <DataReference URI="#invoice34">
    <ds:Transforms>
      <ds:Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
        <ds:XPath xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
            self::xenc:EncryptedData[@Id="example1"]
```

```
        </ds:XPath>
      </ds:Transform>
    </ds:Transforms>
  </DataReference>
</ReferenceList>
```

3.7 The `EncryptionProperties` Element

Identifier

`Type="http://www.w3.org/2001/04/xmlenc#EncryptionProperties"`

(This can be used within a `ds:Reference` element to identify the referent's type.)

Additional information items concerning the generation of the `EncryptedData` or `EncryptedKey` can be placed in an `EncryptionProperty` element (e.g., date/time stamp or the serial number of cryptographic hardware used during encryption). The `Target` attribute identifies the `EncryptedType` structure being described. `anyAttribute` permits the inclusion of attributes from the XML namespace to be included (i.e., `xml:space`, `xml:lang`, and `xml:base`).

```
Schema Definition:

<element name='EncryptionProperties' type='xenc:EncryptionPropertiesType'/>
<complexType name='EncryptionPropertiesType'>
  <sequence>
    <element ref='xenc:EncryptionProperty' maxOccurs='unbounded'/>
  </sequence>
  <attribute name='Id' type='ID' use='optional'/>
</complexType>

  <element name='EncryptionProperty' type='xenc:EncryptionPropertyType'/>
  <complexType name='EncryptionPropertyType' mixed='true'>
    <choice maxOccurs='unbounded'>
      <any namespace='##other' processContents='lax'/>
    </choice>
    <attribute name='Target' type='anyURI' use='optional'/>
    <attribute name='Id' type='ID' use='optional'/>
    <anyAttribute namespace="http://www.w3.org/XML/1998/namespace"/>
  </complexType>
```

4 Processing Rules

This section describes the operations to be performed as part of encryption and decryption processing by implementations of this specification. The conformance requirements are specified over the following roles:

Application
: The application which makes request of an XML Encryption implementation via the provision of data and parameters necessary for its processing.

Encryptor
: An XML Encryption implementation with the role of encrypting data.

Decryptor
: An XML Encryption implementation with the role of decrypting data.

4.1 Encryption

For each data item to be encrypted as an `EncryptedData` or `EncryptedKey` (elements derived from `EncryptedType`), the **encryptor** must:

1. Select the algorithm (and parameters) to be used in encrypting this data.
2. Obtain and (optionally) represent the key.

1. If the key is to be identified (via naming, URI, or included in a child element), construct the `ds:KeyInfo` as appropriate (e.g., `ds:KeyName`, `ds:KeyValue`, `ds:RetrievalMethod`, etc.)
2. If the key itself is to be encrypted, construct an `EncryptedKey` element by recursively applying this encryption process. The result may then be a child of `ds:KeyInfo`, or it may exist elsewhere and may be identified in the preceding step.

3. Encrypt the data
   1. If the data is an 'element' [XML, section 3] or element 'content' [XML, section 3.1], obtain the octets by serializing the data in UTF-8 as specified in [XML]. (The application MUST provide XML data in [NFC].) Serialization MAY be done by the **encryptor**. If the **encryptor** does not serialize, then the **application** MUST perform the serialization.
   2. If the data is of any other type that is not already octets, the **application** MUST serialize it as octets.
   3. Encrypt the octets using the algorithm and key from steps 1 and 2.
   4. Unless the **decryptor** will implicitly know the type of the encrypted data, the **encryptor** SHOULD provide the type for representation.

      The definition of this type as bound to an identifier specifies how to obtain and interpret the plaintext octets after decryption. For example, the identifier could indicate that the data is an instance of another application (e.g., some XML compression application) that must be further processed. Or, if the data is a simple octet sequence it MAY be described with the `MimeType` and `Encoding` attributes. For example, the data might be an XML document (`MimeType="text/xml"`), sequence of characters (`MimeType="text/plain"`), or binary image data (`MimeType="image/png`").

4. Build the `EncryptedType` (`EncryptedData` or `EncryptedKey`) structure:

   An `EncryptedType` structure represents all of the information previously discussed including the type of the encrypted data, encryption algorithm, parameters, key, type of the encrypted data, etc.

   1. If the encrypted octet sequence obtained in step 3 is to be stored in the `CipherData` element within the `EncryptedType`, then the encrypted octet sequence is base64 encoded and inserted as the content of a `CipherValue` element.
   2. If the encrypted octet sequence is to be stored externally to the `EncryptedType` structure, then store or return the encrypted octet sequence, and represent the URI and transforms (if any) required for the decryptor to retrieve the encrypted octet sequence within a `CipherReference` element.

5. Process EncryptedData
   1. If the `Type` of the encrypted data is 'element' or element 'content', then the **encryptor** MUST be able to return the `EncryptedData` element to the **application**. The **application** MAY use this as the top-level element in a new XML document or insert it into another XML document, which may require a re-encoding.

      The **encryptor** SHOULD be able to replace the unencrypted 'element' or 'content' with the EncryptedData element. When an **application** requires an XML element or content to be replaced, it supplies the XML document context in addition to identifying the element or content to be replaced. The **encryptor** removes the identified element or content and inserts the `EncryptedData` element in its place.

      (Note: If the `Type` is "content" the document resulting from decryption will not be well-formed if (a) the original plaintext was not well-formed (e.g., PCDATA by itself is not well-formed) and (b) the `EncryptedData` element was previously the root element of the document)

   2. If the `Type` of the encrypted data *is not* 'element' or element 'content', then the **encryptor** MUST always return the `EncryptedData` element to the **application**. The **application** MAY use this as the top-level element in a new XML document or insert it into another

XML document, which may require a re-encoding.

4.2 Decryption

For each `EncryptedType` derived element, (i.e., `EncryptedData` or `EncryptedKey`), to be decrypted, the **decryptor** must:

1. Process the element to determine the algorithm, parameters and `ds:KeyInfo` element to be used. If some information is omitted, the **application** MUST supply it.
2. Locate the data encryption key according to the `ds:KeyInfo` element, which may contain one or more children elements. These children have no implied processing order. If the data encryption key is encrypted, locate the corresponding key to decrypt it. (This may be a recursive step as the key-encryption key may itself be encrypted.) Or, one might retrieve the data encryption key from a local store using the provided attributes or implicit binding.
3. Decrypt the data contained in the `CipherData` element.
    1. If a `CipherValue` child element is present, then the associated text value is retrieved and base64 decoded so as to obtain the encrypted octet sequence.
    2. If a `CipherReference` child element is present, the URI and transforms (if any) are used to retrieve the encrypted octet sequence.
    3. The encrypted octet sequence is decrypted using the algorithm/parameters and key value already determined from steps 1 and 2.
4. Process decrypted data of `Type` 'element' or element 'content'.
    1. The cleartext octet sequence obtained in step 3 is interpreted as UTF-8 encoded character data.
    2. The **decryptor** MUST be able to return the value of `Type` and the UTF-8 encoded XML character data. The **decryptor** is NOT REQUIRED to perform validation on the serialized XML.
    3. The **decryptor** SHOULD support the ability to replace the `EncryptedData` element with the decrypted 'element' or element 'content' represented by the UTF-8 encoded characters. The **decryptor** is NOT REQUIRED to perform validation on the result of this replacement operation.

       The application supplies the XML document context and identifies the `EncryptedData` element being replaced. If the document into which the replacement is occurring is not UTF-8, the **decryptor** MUST transcode the UTF-8 encoded characters into the target encoding.

5. Process decrypted data if `Type` is unspecified or is *not* 'element' or element 'content'.
    1. The cleartext octet sequence obtained in **Step 3** MUST be returned to the **application** for further processing along with the `Type`, `MimeType`, and `Encoding` attribute values when specified. `MimeType` and `Encoding` are advisory. The `Type` value is normative as it may contain information necessary for the processing or interpretation of the data by the application.
    2. Note, this step includes processing data decrypted from an `EncryptedKey`. The cleartext octet sequence represents a key value and is used by the application in decrypting other `EncryptedType` element(s).

4.3 XML Encryption

Encryption and decryption operations are transforms on octets. The **application** is responsible for the marshalling XML such that it can be serialized into an octet sequence, encrypted, decrypted, and be of use to the recipient.

For example, if the application wishes to canonicalize its data or encode/compress the data in an XML packaging format, the application needs to marshal the XML accordingly and identify the resulting type via the `EncryptedData Type` attribute. The likelihood of successful decryption and

subsequent processing will be dependent on the recipient's support for the given type. Also, if the data is intended to be processed both before encryption and after decryption (e.g., XML Signature [XML-DSIG] validation or an XSLT transform) the encrypting application must be careful to preserve information necessary for that process's success.

For interoperability purposes, the following types MUST be implemented such that an implementation will be able to take as input and yield as output data matching the production rules 39 and 43 from [XML]:

element 'http://www.w3.org/2001/04/xmlenc#Element'
"[39] element ::= `EmptyElemTag` | STag content ETag"
content 'http://www.w3.org/2001/04/xmlenc#Content'
"[43] content ::= `CharData`? ((element | Reference | CDSect | PI | Comment) `CharData`?)*"

The following sections contain specifications for decrypting, replacing, and serializing XML content (i.e., `Type` 'element' or element 'content') using the [XPath] data model. These sections are non-normative and OPTIONAL to implementers of this specification, but they may be normatively referenced by and MANDATORY to other specifications that require a consistent processing for applications, such as [XML-DSIG-Decrypt].

4.3.1 A Decrypt Implementation (Non-normative)

Where *P* is the context in which the serialized XML should be parsed (a document node or element node) and *O* is the octet sequence representing UTF-8 encoded characters resulting from step 4.3 in the *Decryption Processing* (section 4.2). *Y* is node-set representing the decrypted content obtained by the following steps:

1. Let *C* be the **parsing context** of a child of *P*, which consists of the following items:
   - Prefix and namespace name of each namespace that is in scope for *P*.
   - Name and value of each general entity that is effective for the XML document causing *P*.
2. Wrap the decrypted octet stream *O* in the context *C* as specified in *Text Wrapping*.
3. Parse the wrapped octet stream as described in *The Reference Processing Model* (section 4.3.3.2) of [XML-Signature], resulting in a node-set.
4. *Y* is the node-set obtained by removing the root node, the wrapping element node, and its associated set of attribute and namespace nodes from the node-set obtained in Step 3.

4.3.2 A Decrypt and Replace Implementation (Non-normative)

Where *X* is the [XPath] node set corresponding to an XML document and *e* is an `EncryptedData` element node in *X*.

1. *Z* is an [XPath] node-set that identical to X except where the element node *e* is an `EncryptedData` element type. In which case:
   1. Decrypt *e* in the context of its parent node as specified in the *Decryption Implementation* (section 4.3.1) yielding *Y*, an [XPath] node set.
   2. Include *Y* in place of *e* and its descendants in *X*. Since [XPath] does not define methods of replacing node-sets from different documents, the result MUST be equivalent to replacing e with the octet stream resulting from its decryption in the serialized form of *X* and reparsing the document. However, the actual method of performing this operation is left to the implementor.

4.3.3 Serializing XML (Non-normative)

Default Namespace Considerations

In *Encrypting XML* (section 4.1, step 3.1), when serializing an XML fragment special care SHOULD be taken with respect to default namespaces. If the data will be subsequently decrypted in the context of a parent XML document then serialization can produce elements in the wrong namespace. Consider the following fragment of XML:

```
<Document xmlns="http://example.org/">
  <ToBeEncrypted xmlns="" />
</Document>
```

Serialization of the element `ToBeEncrypted` fragment via [XML-C14N] would result in the characters "`<ToBeEncrypted></ToBeEncrypted>`" as an octet stream. The resulting encrypted document would be:

```
<Document xmlns="http://example.org/">
  <EncryptedData xmlns="...">
    <!-- Containing the encrypted
         "<ToBeEncrypted></ToBeEncrypted>" -->
  </EncryptedData>
</Document>
```

Decrypting and replacing the `EncryptedData` within this document would produce the following incorrect result:

```
<Document xmlns="http://example.org/">
  <ToBeEncrypted/>
</Document>
```

This problem arises because most XML serializations assume that the serialized data will be parsed directly in a context where there is no default namespace declaration. Consequently, they do not redundantly declare the empty default namespace with an `xmlns=""`. If, however, the serialized data is parsed in a context where a default namespace declaration is in scope (e.g., the parsing context of a *A Decrypt Implementation* (section 4.3.1)), then it may affect the interpretation of the serialized data.

To solve this problem, a canonicalization algorithm MAY be augmented as follows for use as an XML encryption serializer:

- A default namespace declaration with an empty value (i.e., `xmlns=""`) SHOULD be emitted where it would normally be suppressed by the canonicalization algorithm.

While the result may not be in proper canonical form, this is harmless as the resulting octet stream will not be used directly in a [XML-Signature] signature value computation. Returning to the preceding example with our new augmentation, the `ToBeEncrypted` element would be serialized as follows:

```
<ToBeEncrypted xmlns=""></ToBeEncrypted>
```

When processed in the context of the parent document, this serialized fragment will be parsed and interpreted correctly.

This augmentation can be retroactively applied to an existing canonicalization implementation by canonicalizing each apex node and its descendants from the node set, inserting `xmlns=""` at the appropriate points, and concatenating the resulting octet streams.

XML Attribute Considerations

Similar attention between the relationship of a fragment and the context into which it is being inserted should be given to the `xml:base`, `xml:lang`, and `xml:space` attributes as mentioned in the

Security Considerations of [XML-exc-C14N]. For example, if the element:

```
<Bongo href="example.xml"/>
```

is taken from a context and serialized with no `xml:base` [XML-Base] attribute and parsed in the context of the element:

```
<Baz xml:base="http://example.org/"/>
```

the result will be:

```
<Baz xml:base="http://example.org/"><Bongo href="example.xml"/></Baz>
```

`Bongo`'s `href` is subsequently interpreted as "`http://example.org/example.xml`". If this is not the correct URI, `Bongo` should have been serialized with its own `xml:base` attribute.

Unfortunately, the recommendation that an empty value be emitted to divorce the default namespace of the fragment from the context into which it is being inserted can not be made for the attributes `xml:base`, and `xml:space`. (*Error 41* of the XML 1.0 Second Edition Specification Errata clarifies that an empty string value of the attribute `xml:lang` *is* considered as if, "there is no language information available, just as if `xml:lang` had not been specified".)The interpretation of an empty value for the `xml:base` or `xml:space` attributes is undefined or maintains the contextual value. Consequently, applications SHOULD ensure (1) fragments that are to be encrypted are not dependent on XML attributes, or (2) if they are dependent and the resulting document is intended to be *valid* [XML], the fragment's definition permits the presence of the attributes and that the attributes have non-empty values.

4.3.4 Text Wrapping (Non-normative)

This section specifies the process for wrapping text in a given parsing context. The process is based on the proposal by Richard Tobin [Tobin] for constructing the infoset [XML-Infoset] of an external entity.

The process consists of the following steps:

1. If the parsing context contains any general entities, then emit a document type declaration that provides entity declarations.
2. Emit a `dummy` element start-tag with namespace declaration attributes declaring all the namespaces in the parsing context.
3. Emit the text.
4. Emit a `dummy` element end-tag.

In the above steps, the document type declaration and `dummy` element tags MUST be encoded in UTF-8.

Consider the following document containing an `EncryptedData` element:

```
<!DOCTYPE Document [
  <!ENTITY dsig "http://www.w3.org/2000/09/xmldsig#">
]>
<Document xmlns="http://example.org/">
  <foo:Body xmlns:foo="http://example.org/foo">
    <EncryptedData xmlns="http://www.w3.org/2001/04/xmlenc#"
                   Type="http://www.w3.org/2001/04/xmlenc#Element">
      ...
    </EncryptedData>
  </foo:Body>
</Document>
```

If the `EncryptedData` element is fed is decrypted to the text "`<One><foo:Two/></One>`", then the wrapped form is as follows:

```
<!DOCTYPE dummy [
  <!ENTITY dsig "http://www.w3.org/2000/09/xmldsig#">
]>
<dummy xmlns="http://example.org/"
       xmlns:foo="http://example.org/foo"><One><foo:Two/></One></dummy>
```

5. Algorithms

This section discusses algorithms used with the XML Encryption specification. Entries contain the identifier to be used as the value of the `Algorithm` attribute of the `EncryptionMethod` element or other element representing the role of the algorithm, a reference to the formal specification, definitions for the representation of keys and the results of cryptographic operations where applicable, and general applicability comments.

5.1 Algorithm Identifiers and Implementation Requirements

All algorithms listed below have implicit parameters depending on their role. For example, the data to be encrypted or decrypted, keying material, and direction of operation (encrypting or decrypting) for encryption algorithms. Any explicit additional parameters to an algorithm appear as content elements within the element. Such parameter child elements have descriptive element names, which are frequently algorithm specific, and SHOULD be in the same namespace as this XML Encryption specification, the XML Signature specification, or in an algorithm specific namespace. An example of such an explicit parameter could be a nonce (unique quantity) provided to a key agreement algorithm.

This specification defines a set of algorithms, their URIs, and requirements for implementation. Levels of requirement specified, such as "REQUIRED" or "OPTIONAL", refer to implementation, not use. Furthermore, the mechanism is extensible, and alternative algorithms may be used.

> ***There is currently no consensus on mandatory to implement algorithms; the current draft text represents one possible outcome. Positions of some Working Group members against the currently expressed set of mandatory to implement algorithms include:***
>
> - ***Given limited support in parts of the industry, Elliptic Curve Diffie-Hellman Key Agreement is not acceptable as a mandatory to implement algorithm, and might lead to lack of implementation of this version of the specification.***
> - ***There should be recommended algorithms, but no mandatory to implement algorithms. The rationale is that this gives greater flexibility to deployments. (Other WG members argued against this since it could harm interoperability not having mandatory algorithms.)***
>
> ***The opposing position is that, going forward, this specification needs to have credible algorithm agility for both hash and public-key algorithms: Should one set of algorithms prove weak, this would enable a quick switch-over. Therefore, there should be two mandatory to implement public-key algorithms from different families. At this time, elliptic curve based algorithms are the only credible contenders. They have the additional benefit of providing a reasonable balance between key sizes and security level.***

Table of Algorithms

The table below lists the categories of algorithms. Within each category, a brief name, the level of implementation requirement, and an identifying URI are given for each algorithm.

Block Encryption

1. REQUIRED TRIPLEDES
   http://www.w3.org/2001/04/xmlenc#tripledes-cbc
2. REQUIRED AES-128
   http://www.w3.org/2001/04/xmlenc#aes128-cbc
3. REQUIRED AES-256
   http://www.w3.org/2001/04/xmlenc#aes256-cbc
4. OPTIONAL AES-192
   http://www.w3.org/2001/04/xmlenc#aes192-cbc

Stream Encryption

1. none
   Syntax and recommendations are given below to support user specified algorithms.

Key Transport

1. REQUIRED RSA-v1.5
   http://www.w3.org/2001/04/xmlenc#rsa-1_5
2. REQUIRED RSA-OAEP
   http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p

Key Agreement

1. OPTIONAL Diffie-Hellman
   http://www.w3.org/2001/04/xmlenc#dh
2. REQUIRED Elliptic Curve Diffie-Hellman (Ephemeral-Static mode)
   http://www.w3.org/2009/xmlenc11#ECDH-ES

Symmetric Key Wrap

1. REQUIRED TRIPLEDES KeyWrap
   http://www.w3.org/2001/04/xmlenc#kw-tripledes
2. REQUIRED AES-128 KeyWrap
   http://www.w3.org/2001/04/xmlenc#kw-aes128
3. REQUIRED AES-256 KeyWrap
   http://www.w3.org/2001/04/xmlenc#kw-aes256
4. OPTIONAL AES-192 KeyWrap
   http://www.w3.org/2001/04/xmlenc#kw-aes192

Message Digest

1. REQUIRED SHA1
   http://www.w3.org/2000/09/xmldsig#sha1
2. RECOMMENDED SHA256
   http://www.w3.org/2001/04/xmlenc#sha256
3. OPTIONAL SHA512
   http://www.w3.org/2001/04/xmlenc#sha512
4. OPTIONAL RIPEMD-160
   http://www.w3.org/2001/04/xmlenc#ripemd160

Message Authentication

1. RECOMMENDED XML Digital Signature
   http://www.w3.org/2000/09/xmldsig#

Canonicalization

1. OPTIONAL Canonical XML (omits comments)
   http://www.w3.org/TR/2001/REC-xml-c14n-20010315
2. OPTIONAL Canonical XML with Comments
   http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
3. OPTIONAL Exclusive XML Canonicalization (omits comments)
   http://www.w3.org/2001/10/xml-exc-c14n#
4. OPTIONAL Exclusive XML Canonicalization with Comments
   http://www.w3.org/2001/10/xml-exc-c14n#WithComments

Encoding

1. REQUIRED base64
   http://www.w3.org/2000/09/xmldsig#base64

5.2 Block Encryption Algorithms

Block encryption algorithms are designed for encrypting and decrypting data in fixed size, multiple octet blocks. Their identifiers appear as the value of the `Algorithm` attributes of `EncryptionMethod` elements that are children of `EncryptedData`.

Block encryption algorithms take, as implicit arguments, the data to be encrypted or decrypted, the keying material, and their direction of operation. For all of these algorithms specified below, an initialization vector (IV) is required that is encoded with the cipher text. For user specified block encryption algorithms, the IV, if any, could be specified as being with the cipher data, as an algorithm content element, or elsewhere.

The IV is encoded with and before the cipher text for the algorithms below for ease of availability to the decryption code and to emphasize its association with the cipher text. Good cryptographic practice requires that a different IV be used for every encryption.

Padding

Since the data being encrypted is an arbitrary number of octets, it may not be a multiple of the block size. This is solved by padding the plain text up to the block size before encryption and unpadding after decryption. The padding algorithm is to calculate the smallest non-zero number of octets, say `N`, that must be suffixed to the plain text to bring it up to a multiple of the block size. We will assume the block size is `B` octets so `N` is in the range of 1 to `B`. Pad by suffixing the plain text with `N-1` arbitrary pad bytes and a final byte whose value is `N`. On decryption, just take the last byte and, after sanity checking it, strip that many bytes from the end of the decrypted cipher text.

For example, assume an 8 byte block size and plain text of `0x616263`. The padded plain text would then be `0x616263????????05` where the "??" bytes can be any value. Similarly, plain text of `0x2122232425262728` would be padded to `0x21222324252627??????????????08`.

5.2.1 Triple DES

Identifier:
http://www.w3.org/2001/04/xmlenc#tripledes-cbc (REQUIRED)

ANSI X9.52 [TRIPLEDES] specifies three sequential FIPS 46-3 [DES] operations. The XML Encryption TRIPLEDES consists of a DES encrypt, a DES decrypt, and a DES encrypt used in the Cipher Block Chaining (CBC) mode with 192 bits of key and a 64 bit Initialization Vector (IV). Of the key bits, the first 64 are used in the first DES operation, the second 64 bits in the middle DES operation, and the third 64 bits in the last DES operation.

Note: Each of these 64 bits of key contain 56 effective bits and 8 parity bits. Thus there are only 168 operational bits out of the 192 being transported for a TRIPLEDES key. (Depending on the criterion used for analysis, the effective strength of the key may be thought to be 112 bits (due to meet in the middle attacks) or even less.)

The resulting cipher text is prefixed by the IV. If included in XML output, it is then base64 encoded. An example TRIPLEDES EncryptionMethod is as follows:

```
<EncryptionMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#tripledes-cbc"/>
```

5.2.2 AES

Identifier:

http://www.w3.org/2001/04/xmlenc#aes128-cbc (REQUIRED)
http://www.w3.org/2001/04/xmlenc#aes192-cbc (OPTIONAL)
http://www.w3.org/2001/04/xmlenc#aes256-cbc (REQUIRED)

[AES] is used in the Cipher Block Chaining (CBC) mode with a 128 bit initialization vector (IV). The resulting cipher text is prefixed by the IV. If included in XML output, it is then base64 encoded. An example AES EncryptionMethod is as follows:

```
<EncryptionMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#aes128-cbc"/>
```

5.3 Stream Encryption Algorithms

Simple stream encryption algorithms generate, based on the key, a stream of bytes which are XORed with the plain text data bytes to produce the cipher text on encryption and with the cipher text bytes to produce plain text on decryption. They are normally used for the encryption of data and are specified by the value of the `Algorithm` attribute of the `EncryptionMethod` child of an `EncryptedData` element.

NOTE: It is critical that each simple stream encryption key (or key and initialization vector (IV) if an IV is also used) be used once only. If the same key (or key and IV) is ever used on two messages then, by XORing the two cipher texts, you can obtain the XOR of the two plain texts. This is usually very compromising.

No specific stream encryption algorithms are specified herein but this section is included to provide general guidelines.

Stream algorithms typically use the optional `KeySize` explicit parameter. In cases where the key size is not apparent from the algorithm URI or key source, as in the use of key agreement methods, this parameter sets the key size. If the size of the key to be used is apparent and disagrees with the `KeySize` parameter, an error MUST be returned. Implementation of any stream algorithms is optional. The schema for the KeySize parameter is as follows:

```
Schema Definition:

<simpleType name='KeySizeType'>
  <restriction base="integer"/>
</simpleType>
```

5.4 Key Transport

Key Transport algorithms are public key encryption algorithms especially specified for encrypting and decrypting keys. Their identifiers appear as `Algorithm` attributes to `EncryptionMethod` elements that are children of `EncryptedKey`. `EncryptedKey` is in turn the child of a `ds:KeyInfo` element. The type of key being transported, that is to say the algorithm in which it is planned to use the transported key, is given by the `Algorithm` attribute of the `EncryptionMethod` child of the `EncryptedData` or `EncryptedKey` parent of this `ds:KeyInfo` element.

(Key Transport algorithms may optionally be used to encrypt data in which case they appear directly as the `Algorithm` attribute of an `EncryptionMethod` child of an `EncryptedData` element. Because they use public key algorithms directly, Key Transport algorithms are not efficient for the transport of any amounts of data significantly larger than symmetric keys.)

The RSA v1.5 Key Transport algorithm given below are those used in conjunction with TRIPLEDES and the Cryptographic Message Syntax (CMS) of S/MIME [CMS-Algorithms]. The RSA v2 Key

Transport algorithm given below is that used in conjunction with AES and CMS [AES-WRAP].

5.4.1 RSA Version 1.5

Identifier:
http://www.w3.org/2001/04/xmlenc#rsa-1_5 (REQUIRED)

The RSAES-PKCS1-v1_5 algorithm, specified in RFC 2437 [PKCS1], takes no explicit parameters. An example of an RSA Version 1.5 `EncryptionMethod` element is:

```
<EncryptionMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
```

The `CipherValue` for such an encrypted key is the base64 [MIME] encoding of the octet string computed as per RFC 2437 [PKCS1, section 7.2.1: Encryption operation]. As specified in the EME-PKCS1-v1_5 function RFC 2437 [PKCS1, section 9.1.2.1], the value input to the key transport function is as follows:

```
CRYPT ( PAD ( KEY ))
```

where the padding is of the following special form:

```
02 | PS* | 00 | key
```

where "|" is concatenation, "02" and "00" are fixed octets of the corresponding hexadecimal value, PS is a string of strong pseudo-random octets [RANDOM] at least eight octets long, containing no zero octets, and long enough that the value of the quantity being CRYPTed is one octet shorter than the RSA modulus, and "key" is the key being transported. The key is 192 bits for TRIPLEDES and 128, 192, or 256 bits for AES. Support of this key transport algorithm for transporting 192 bit keys is MANDATORY to implement. Support of this algorithm for transporting other keys is OPTIONAL. RSA-OAEP is RECOMMENDED for the transport of AES keys.

The resulting base64 [MIME] string is the value of the child text node of the `CipherData` element, e.g.

```
<CipherData>
  <CipherValue>IWijxQjUrcXBYoCei4QxjWo9Kg8D3p9tlWoT4
   t0/gyTE96639In0FZFY2/rvP+/bMJ01EArmKZsR5VW3rwoPxw=
  </Ciphervalue>
</CipherData>
```

5.4.2 RSA-OAEP

Identifier:
http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p (REQUIRED)

The RSAES-OAEP-ENCRYPT algorithm, as specified in RFC 2437 [PKCS1], takes three parameters. The two user specified parameters are a MANDATORY message digest function and an OPTIONAL encoding octet string `OAEPparams`. The message digest function is indicated by the `Algorithm` attribute of a child `ds:DigestMethod` element and the mask generation function, the third parameter, is always MGF1 with SHA1 (mgf1SHA1Identifier). Both the message digest and mask generation functions are used in the EME-OAEP-ENCODE operation as part of RSAES-OAEP-ENCRYPT. The encoding octet string is the base64 decoding of the content of an optional `OAEPparams` child element . If no `OAEPparams` child is provided, a null string is used.

```
Schema Definition:
```

```
    <!-- use these element types as children of EncryptionMethod
         when used with RSA-OAEP -->
    <element name='OAEPparams' minOccurs='0' type='base64Binary'/>
    <element ref='ds:DigestMethod' minOccurs='0'/>
```

An example of an RSA-OAEP element is:

```
<EncryptionMethod
   Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-oaep-mgf1p">
  <OAEPparams> 9lWu3Q== </OAEPparams>
  <ds:DigestMethod
      Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
<EncryptionMethod>
```

The `CipherValue` for an RSA-OAEP encrypted key is the base64 [MIME] encoding of the octet string computed as per RFC 2437 [PKCS1, section 7.1.1: Encryption operation]. As described in the EME-OAEP-ENCODE function RFC 2437 [PKCS1, section 9.1.1.1], the value input to the key transport function is calculated using the message digest function and string specified in the `DigestMethod` and `OAEPparams` elements and using the mask generator function MGF1 (with SHA1) specified in RFC 2437. The desired output length for EME-OAEP-ENCODE is one byte shorter than the RSA modulus.

The transported key size is 192 bits for TRIPLEDES and 128, 192, or 256 bits for AES. Implementations MUST implement RSA-OAEP for the transport of 128 and 256 bit keys. They MAY implement RSA-OAEP for the transport of other keys.

5.5 Key Agreement

A Key Agreement algorithm provides for the derivation of a shared secret key based on a shared secret computed from certain types of compatible public keys from both the sender and the recipient. Information from the originator to determine the secret is indicated by an optional `OriginatorKeyInfo` parameter child of an `AgreementMethod` element while that associated with the recipient is indicated by an optional `RecipientKeyInfo`. A shared key is derived from this shared secret by a method determined by the Key Agreement algorithm.

Note: XML Encryption does not provide an on-line key agreement negotiation protocol. The `AgreementMethod` element can be used by the originator to identify the keys and computational procedure that were used to obtain a shared encryption key. The method used to obtain or select the keys or algorithm used for the agreement computation is beyond the scope of this specification.

The `AgreementMethod` element appears as the content of a `ds:KeyInfo` since, like other `ds:KeyInfo` children, it yields a key. This `ds:KeyInfo` is in turn a child of an `EncryptedData` or `EncryptedKey` element. The `Algorithm` attribute and `KeySize` child of the `EncryptionMethod` element under this `EncryptedData` or `EncryptedKey` element are implicit parameters to the key agreement computation. In cases where this `EncryptionMethod` algorithm URI is insufficient to determine the key length, a `KeySize` MUST have been included. In addition, the sender may place a `KA-Nonce` element under `AgreementMethod` to assure that different keying material is generated even for repeated agreements using the same sender and recipient public keys. For example:

```
<EncryptedData>
  <EncryptionMethod Algorithm="Example:Block/Alg"
    <KeySize>80</KeySize>
  </EncryptionMethod>
  <ds:KeyInfo xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <AgreementMethod Algorithm="example:Agreement/Algorithm">
      <KA-Nonce>Zm9v</KA-Nonce>
      <ds:DigestMethod
      Algorithm="http://www.w3.org/2001/04/xmlenc#sha1"/>
     <OriginatorKeyInfo>
        <ds:KeyValue>....</ds:KeyValue>
```

```
      </OriginatorKeyInfo>
      <RecipientKeyInfo>
        <ds:KeyValue>....</ds:KeyValue>
      </RecipientKeyInfo>
    </AgreementMethod>
  </ds:KeyInfo>
  <CipherData>...</CipherData>
</EncryptedData>
```

If the agreed key is being used to wrap a key, rather than data as above, then `AgreementMethod` would appear inside a `ds:KeyInfo` inside an `EncryptedKey` element.

The Schema for `AgreementMethod` is as follows:

```
Schema Definition:

<element name="AgreementMethod" type="xenc:AgreementMethodType"/>
<complexType name="AgreementMethodType" mixed="true">
  <sequence>
    <element name="KA-Nonce" minOccurs="0" type="base64Binary"/>
    <!-- <element ref="ds:DigestMethod" minOccurs="0"/> -->
    <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
    <element name="OriginatorKeyInfo" minOccurs="0"
             type="ds:KeyInfoType"/>
    <element name="RecipientKeyInfo" minOccurs="0"
             type="ds:KeyInfoType"/>
  </sequence>
  <attribute name="Algorithm" type="anyURI" use="required"/>
</complexType>
```

5.5.1 Diffie-Hellman Key Values

Identifier:
http://www.w3.org/2001/04/xmlenc#DHKeyValue (OPTIONAL)

Diffie-Hellman keys can appear directly within `KeyValue` elements or be obtained by `ds:RetrievalMethod` fetches as well as appearing in certificates and the like. The above identifier can be used as the value of the `Type` attribute of `Reference` or `ds:RetrievalMethod` elements.

As specified in [ESDH], a DH public key consists of up to six quantities, two large primes p and q, a "generator" g, the public key, and validation parameters "seed" and "pgenCounter". These relate as follows: The public key = (g**x mod p) where x is the corresponding private key; p = j*q + 1 where j >= 2. "seed" and "pgenCounter" are optional and can be used to determine if the Diffie-Hellman key has been generated in conformance with the algorithm specified in [ESDH]. Because the primes and generator can be safely shared over many DH keys, they may be known from the application environment and are optional. The schema for a `DHKeyValue` is as follows:

```
 Schema:

<element name="DHKeyValue" type="xenc:DHKeyValueType"/>
<complexType name="DHKeyValueType">
   <sequence>
      <sequence minOccurs="0">
         <element name="P" type="ds:CryptoBinary"/>
         <element name="Q" type="ds:CryptoBinary"/>
         <element name="Generator"type="ds:CryptoBinary"/>
      </sequence>
      <element name="Public" type="ds:CryptoBinary"/>
      <sequence minOccurs="0">
         <element name="seed" type="ds:CryptoBinary"/>
         <element name="pgenCounter" type="ds:CryptoBinary"/>
      </sequence>
   </sequence>
</complexType>
```

5.5.2 Diffie-Hellman Key Agreement

Identifier:
http://www.w3.org/2001/04/xmlenc#dh (OPTIONAL)

The Diffie-Hellman (DH) key agreement protocol [ESDH] involves the derivation of shared secret information based on compatible DH keys from the sender and recipient. Two DH public keys are compatible if they have the same prime and generator. If, for the second one, `Y = g**y mod p`, then the two parties can calculate the shared secret `ZZ = ( g**(x*y) mod p )` even though each knows only their own private key and the other party's public key. Leading zero bytes MUST be maintained in `ZZ` so it will be the same length, in bytes, as `p`. The size of `p` MUST be at least 512 bits and `g` at least 160 bits. There are numerous other complex security considerations in the selection of `g`, `p`, and a random `x` as described in [ESDH].

Diffie-Hellman key agreement is optional to implement. An example of a DH `AgreementMethod` element is as follows:

```
<AgreementMethod
    Algorithm="http://www.w3.org/2001/04/xmlenc#dh"
    ds:xmlns="http://www.w3.org/2000/09/xmldsig#">
  <KA-Nonce>Zm9v</KA-Nonce>
  <ds:DigestMethod
   Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
  <OriginatorKeyInfo>
    <ds:X509Data><ds:X509Certificate>
      ...
    </ds:X509Certificate></ds:X509Data>
  </OriginatorKeyInfo>
  <RecipientKeyInfo><ds:KeyValue>
    ...
  </ds:KeyValue></RecipientKeyInfo>
</AgreementMethod>
```

Assume the Diffie-Hellman shared secret is the octet sequence `ZZ`. The shared keying material needed will then be calculated as follows:

```
Keying Material = KM(1) | KM(2) | ...
```

where "|" is byte stream concatenation and

```
KM(counter) = DigestAlg ( ZZ | counter | EncryptionAlg |
              KA-Nonce | KeySize )
```

`DigestAlg`
: The message digest algorithm specified by the `DigestMethod` child of `AgreementMethod`.

`EncryptionAlg`
: The URI of the encryption algorithm, including possible key wrap algorithms, in which the derived keying material is to be used ("Example:Block/Alg" in the example above), not the URI of the agreement algorithm. This is the value of the `Algorithm` attribute of the `EncryptionMethod` child of the `EncryptedData` or `EncryptedKey` grandparent of `AgreementMethod`.

`KA-Nonce`
: The base64 decoding the content of the `KA-Nonce` child of `AgreementMethod`, if present. If the `KA-Nonce` element is absent, it is null.

`Counter`
: A one byte counter starting at one and incrementing by one. It is expressed as two hex digits where letters A through F are in upper case.

`KeySize`
: The size in bits of the key to be derived from the shared secret as the UTF-8 string for the

corresponding decimal integer with only digits in the string and no leading zeros. For some algorithms the key size is inherent in the URI. For others, such as most stream ciphers, it must be explicitly provided.

For example, the initial `(KM(1))` calculation for the `EncryptionMethod` of the Key Agreement example (section 5.5) would be as follows, where the binary one byte counter value of 1 is represented by the two character UTF-8 sequence `01`, `ZZ` is the shared secret, and "`foo`" is the base64 decoding of "`Zm9v`".

```
SHA-1 ( ZZ01Example:Block/Algfoo80 )
```

Assuming that `ZZ` is `0xDEADBEEF`, that would be

```
SHA-1( 0xDEADBEEF30314578616D706C653A426C6F636B2F416C67666F6F3830 )
```

whose value is

```
0x534C9B8C4ABDCB50038B42015A181711068B08C1
```

Each application of `DigestAlg` for successive values of `Counter` will produce some additional number of bytes of keying material. From the concatenated string of one or more `KM`'s, enough leading bytes are taken to meet the need for an actual key and the remainder discarded. For example, if `DigestAlg` is SHA-1 which produces 20 octets of hash, then for 128 bit AES the first 16 bytes from `KM(1)` would be taken and the remaining 4 bytes discarded. For 256 bit AES, all of `KM(1)` suffixed with the first 12 bytes of KM(2) would be taken and the remaining 8 bytes of `KM(2)` discarded.

5.5.3 Elliptic Curve Diffie-Hellman (ECDH) Key Values

Identifier:
http://www.w3.org/2009/xmldsig11#ECKeyValue (RECOMMENDED)

ECDH has identical public key parameters as ECDSA and can be represented with the ECPublicKey element [XMLDSIG11]. Note that if the curve parameters are explicitly stated using the ECParameters element, then the Cofactor element MUST be included.

As with Diffie-Hellman keys, Elliptic Curve Key Values can appear directly within `KeyValue` elements or be obtained by `ds:RetrievalMethod` fetches as well as appearing in certificates and the like. The above identifier can be used as the value of the `Type` attribute of `Reference` or `ds:RetrievalMethod` elements.

5.5.4 Elliptic Curve Diffie-Hellman (ECDH) Key Agreement (Ephemeral-Static Mode)

Identifier:
http://www.w3.org/2009/xmlenc11#ECDH-ES (REQUIRED)

ECDH is the elliptic curve analogue to the Diffie-Hellman key agreement algorithm. Details of the ECDH primitive can be found in section 5.7.1.2 of NIST SP 800-56A [SP800-56A]. When ECDH is used in Ephemeral-Static (ES) mode, the recipient has a static key pair, but the sender generates a ephemeral key pair for each message. The same ephemeral key may be used when there are multiple recipients that use the same curve parameters.

The shared key material is calculated from the Diffie-Hellman shared secret using a key derivation function (KDF). While applications may define other KDFs, compliant implementations MUST implement the Concatenation KDF specified in section 5.8.1 of SP800-56A [SP800-56A]. Parameters for the Concatenation KDF are represented by the SP80056AConcatKDF element. An

example of a DH `AgreementMethod` element is as follows:

```
<AgreementMethod
    Algorithm="http://www.w3.org/2009/xmlenc11#ECDH-ES"
    xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
    xmlns:ds11="http://www.w3.org/2009/xmldsig11"
    xmlns:xenc="http://www.w3.org/2001/04/xmlenc#"
    xmlns:xenc11="http://www.w3.org/2009/xmlenc11#">

  <xenc11:SP80056AConcatKDF>
    <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
    <xenc11:OtherInfo AlgorithmID="0" PartyUInfo="" PartyVInfo=""/>
  </xenc11:SP80056AConcatKDF>
  <OriginatorKeyInfo>
    <ds:KeyValue>
      <ds11:ECPublicKey>
        ...
      </ds11:ECPublicKey>
    </ds:KeyValue>
  </OriginatorKeyInfo>
  <RecipientKeyInfo>
    <ds:X509Data>
      <ds:X509IssuerSerial>
        ...
      </ds:X509IssuerSerial>
    </ds:X509Data>
  </RecipientKeyInfo>
</AgreementMethod>
```

The SP80056AConcatKDF element consists of 2 subelements:

1. The DigestMethod element identifies the digest algorithm used by the KDF. Compliant implementations MUST support SHA-256. Support for SHA-1 and SHA-384 are RECOMMENDED.
2. The OtherInfo element identities the subfields of OtherInfo as defined by SP800-56A. This element is REQUIRED for applications that comply with SP800-56A, but is OPTIONAL otherwise. When the output of the KDF is used with either the Triple DES Key Wrap or the AES Key Wrap algorithms, the AlgorithmID attribute shall contain the value "00".

The schema for the SP80056AConcatKDF element is as follows:

```
Schema Definition:
  <element name="SP80056AConcatKDF" type="xenc11:SP80056AConcatKDFType"/>
  <complexType name="SP80056AConcatKDFType">
    <sequence>
      <element ref="ds:DigestMethod"/>
      <element name="OtherInfo" type="xenc11:OtherInfoType" minOccurs="0"/>
    </sequence>
  </complexType>

  <complexType name="OtherInfoType">
    <attribute name="AlgorithmID" type="hexBinary"/>
    <attribute name="PartyUInfo" type="hexBinary"/>
    <attribute name="PartyVInfo" type="hexBinary"/>
    <attribute name="SuppPubInfo" type="hexBinary" use="optional"/>
  </complexType>
```

```
DTD Definition:
  TBD
```

5.6 Symmetric Key Wrap

Symmetric Key Wrap algorithms are shared secret key encryption algorithms especially specified for encrypting and decrypting symmetric keys. Their identifiers appear as `Algorithm` attribute values to

`EncryptionMethod` elements that are children of `EncryptedKey` which is in turn a child of `ds:KeyInfo` which is in turn a child of `EncryptedData` or another `EncryptedKey`. The type of the key being wrapped is indicated by the `Algorithm` attribute of `EncryptionMethod` child of the parent of the `ds:KeyInfo` grandparent of the `EncryptionMethod` specifying the symmetric key wrap algorithm.

5.6.1 CMS Key Checksum

Some key wrap algorithms make use of a key checksum as defined in CMS [CMS-Wrap]. The algorithm that provides an integrity check value for the key being wrapped is:

1. Compute the 20 octet SHA-1 hash on the key being wrapped.
2. Use the first 8 octets of this hash as the checksum value.

5.6.2 CMS Triple DES Key Wrap

Identifiers and Requirements:

http://www.w3.org/2001/04/xmlenc#kw-tripledes (REQUIRED)

XML Encryption implementations MUST support TRIPLEDES wrapping of 168 bit keys and may optionally support TRIPLEDES wrapping of other keys.

An example of a TRIPLEDES Key Wrap `EncryptionMethod` element is as follows:

```
<EncryptionMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#kw-tripledes"/>
```

The following algorithm wraps (encrypts) a key (the wrapped key, WK) under a TRIPLEDES key-encryption-key (KEK) as adopted from [CMS-Algorithms]:

1. Represent the key being wrapped as an octet sequence. If it is a TRIPLEDES key, this is 24 octets (192 bits) with odd parity bit as the bottom bit of each octet.
2. Compute the CMS key checksum (section 5.6.1) call this CKS.
3. Let `WKCKS = WK || CKS`, where || is concatenation.
4. Generate 8 random octets [RANDOM] and call this IV.
5. Encrypt WKCKS in CBC mode using KEK as the key and IV as the initialization vector. Call the results TEMP1.
6. Let `TEMP2 = IV || TEMP1`.
7. Reverse the order of the octets in `TEMP2` and call the result `TEMP3`.
8. Encrypt `TEMP3` in CBC mode using the `KEK` and an initialization vector of `0x4adda22c79e82105`. The resulting cipher text is the desired result. It is 40 octets long if a 168 bit key is being wrapped.

The following algorithm unwraps (decrypts) a key as adopted from [CMS-Algorithms]:

1. Check if the length of the cipher text is reasonable given the key type. It must be 40 bytes for a 168 bit key and either 32, 40, or 48 bytes for a 128, 192, or 256 bit key. If the length is not supported or inconsistent with the algorithm for which the key is intended, return error.
2. Decrypt the cipher text with TRIPLEDES in CBC mode using the `KEK` and an initialization vector (IV) of `0x4adda22c79e82105`. Call the output `TEMP3`.
3. Reverse the order of the octets in TEMP3 and call the result `TEMP2`.
4. Decompose TEMP2 into IV, the first 8 octets, and `TEMP1`, the remaining octets.
5. Decrypt TEMP1 using TRIPLEDES in CBC mode using the `KEK` and the IV found in the previous step. Call the result `WKCKS`.
6. Decompose `WKCKS`. `CKS` is the last 8 octets and `WK`, the wrapped key, are those octets before the `CKS`.
7. Calculate a CMS key checksum (section 5.6.1) over the `WK` and compare with the `CKS` extracted

in the above step. If they are not equal, return error.
8. WK is the wrapped key, now extracted for use in data decryption.

5.6.3 AES KeyWrap

Identifiers and Requirements:

http://www.w3.org/2001/04/xmlenc#kw-aes128 (REQUIRED)
http://www.w3.org/2001/04/xmlenc#kw-aes192 (OPTIONAL)
http://www.w3.org/2001/04/xmlenc#kw-aes256 (REQUIRED)

Implementation of AES key wrap is described below, as suggested by NIST. It provides for confidentiality and integrity. This algorithm is defined only for inputs which are a multiple of 64 bits. The information wrapped need not actually be a key. The algorithm is the same whatever the size of the AES key used in wrapping, called the key encrypting key or KEK. The implementation requirements are indicated below.

128 bit AES Key Encrypting Key
Implementation of wrapping 128 bit keys REQUIRED.
Wrapping of other key sizes OPTIONAL.

192 bit AES Key Encrypting Key
All support OPTIONAL.

256 bit AES Key Encrypting Key
Implementation of wrapping 256 bit keys REQUIRED.
Wrapping of other key sizes OPTIONAL.

Assume that the data to be wrapped consists of N 64-bit data blocks denoted P(1), P(2), P(3) ... P(N). The result of wrapping will be N+1 64-bit blocks denoted C(0), C(1), C(2), ... C(N). The key encrypting key is represented by K. Assume integers i, j, and t and intermediate 64-bit register A, 128-bit register B, and array of 64-bit quantities R(1) through R(N).

"|" represents concatenation so x|y, where x and y and 64-bit quantities, is the 128-bit quantity with x in the most significant bits and y in the least significant bits. AES(K)enc(x) is the operation of AES encrypting the 128-bit quantity x under the key K. AES(K)dec(x) is the corresponding decryption operation. XOR(x,y) is the bitwise exclusive or of x and y. MSB(x) and LSB(y) are the most significant 64 bits and least significant 64 bits of x and y respectively.

If N is 1, a single AES operation is performed for wrap or unwrap. If N>1, then 6*N AES operations are performed for wrap or unwrap.

The key wrap algorithm is as follows:

1. If N is 1:
   - B=AES(K)enc(0xA6A6A6A6A6A6A6A6|P(1))
   - C(0)=MSB(B)
   - C(1)=LSB(B)

   If N>1, perform the following steps:
2. Initialize variables:
   - Set A to 0xA6A6A6A6A6A6A6A6
   - For i=1 to N,
     R(i)=P(i)
3. Calculate intermediate values:
   - For j=0 to 5,
     - For i=1 to N,
       t= i + j*N
       B=AES(K)enc(A|R(i))
       A=XOR(t,MSB(B))
       R(i)=LSB(B)

4. Output the results:
    - Set `C(0)=A`
    - For `i=1` to `N`,
      `C(i)=R(i)`

The key unwrap algorithm is as follows:

1. If `N` is `1`:
    - `B=AES(K)dec(C(0)|C(1))`
    - `P(1)=LSB(B)`
    - If `MSB(B)` is `0xA6A6A6A6A6A6A6A6`, return success. Otherwise, return an integrity check failure error.

   If `N`>1, perform the following steps:
2. Initialize the variables:
    - `A=C(0)`
    - For `i=1` to `N`,
      `R(i)=C(i)`
3. Calculate intermediate values:
    - For `j=5` to `0`,
        - For `i=N` to `1`,
          `t= i + j*N`
          `B=AES(K)dec(XOR(t,A)|R(i))`
          `A=MSB(B)`
          `R(i)=LSB(B)`
4. Output the results:
    - For `i=1` to `N`,
      `P(i)=R(i)`
    - If `A` is `0xA6A6A6A6A6A6A6A6`, return success. Otherwise, return an integrity check failure error.

For example, wrapping the data `0x00112233445566778899AABBCCDDEEFF` with the `KEK` `0x000102030405060708090A0B0C0D0E0F` produces the ciphertext of `0x1FA68B0A8112B447`, `0xAEF34BD8FB5A7B82`, `0x9D3E862371D2CFE5`.

5.7 Message Digest

Message digest algorithms can be used in `AgreementMethod` as part of the key derivation, within RSA-OAEP encryption as a hash function, and in connection with the HMAC message authentication code method as described in [XML-DSIG].)

5.7.1 SHA1

Identifier:
http://www.w3.org/2000/09/xmldsig#sha1 (REQUIRED)

The SHA-1 algorithm [SHA] takes no explicit parameters. An example of an SHA-1 `DigestMethod` element is:

```
<DigestMethod
 Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
```

A SHA-1 digest is a 160-bit string. The content of the `DigestValue` element shall be the base64 encoding of this bit string viewed as a 20-octet octet stream. For example, the `DigestValue` element for the message digest:

```
A9993E36 4706816A BA3E2571 7850C26C 9CD0D89D
```

from Appendix A of the SHA-1 standard would be:

```
<DigestValue>qZk+NkcGgWq6PiVxeFDCbJzQ2J0=</DigestValue>
```

5.7.2 SHA256

Identifier:
http://www.w3.org/2001/04/xmlenc#sha256 (RECOMMENDED)

The SHA-256 algorithm [SHA] takes no explicit parameters. An example of an SHA-256 `DigestMethod` element is:

```
<DigestMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
```

A SHA-256 digest is a 256-bit string. The content of the `DigestValue` element shall be the base64 encoding of this bit string viewed as a 32-octet octet stream.

5.7.3 SHA512

Identifier:
http://www.w3.org/2001/04/xmlenc#sha512 (OPTIONAL)

The SHA-512 algorithm [SHA] takes no explicit parameters. An example of an SHA-512 `DigestMethod` element is:

```
<DigestMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#sha512"/>
```

A SHA-512 digest is a 512-bit string. The content of the `DigestValue` element shall be the base64 encoding of this bit string viewed as a 64-octet octet stream.

5.7.4 RIPEMD-160

Identifier:
http://www.w3.org/2001/04/xmlenc#ripemd160 (OPTIONAL)

The RIPEMD-160 algorithm [RIPEMD-160] takes no explicit parameters. An example of an RIPEMD-160 `DigestMethod` element is:

```
<DigestMethod
 Algorithm="http://www.w3.org/2001/04/xmlenc#ripemd160"/>
```

A RIPEMD-160 digest is a 160-bit string. The content of the `DigestValue` element shall be the base64 encoding of this bit string viewed as a 20-octet octet stream.

5.8 Message Authentication

Identifier:
http://www.w3.org/2000/09/xmldsig# (RECOMMENDED)

XML Signature [XML-DSIG] is OPTIONAL to implement for XML encryption applications. It is the recommended way to provide key based authentication.

5.9 Canonicalization

A Canonicalization of XML is a method of consistently serializing XML into an octet stream as is necessary prior to encrypting XML.

5.9.1 Inclusive Canonicalization

Identifiers:

http://www.w3.org/TR/2001/REC-xml-c14n-20010315 (OPTIONAL)
http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments (OPTIONAL)

Canonical XML [Canon] is a method of serializing XML which includes the in scope namespace and xml namespace attribute context from ancestors of the XML being serialized.

If XML is to be encrypted and then later decrypted into a different environment and it is desired to preserve namespace prefix bindings and the value of attributes in the "xml" namespace of its original environment, then the canonical XML with comments version of the XML should be the serialization that is encrypted.

5.9.2 Exclusive Canonicalization

Identifiers:

http://www.w3.org/2001/10/xml-exc-c14n# (OPTIONAL)
http://www.w3.org/2001/10/xml-exc-c14n#WithComments (OPTIONAL)

Exclusive XML Canonicalization [Exclusive] serializes XML in such a way as to include to the minimum extent practical the namespace prefix binding and xml namespace attribute context inherited from ancestor elements.

It is the recommended method where the outer context of a fragment which was signed and then encrypted may be changed. Otherwise the validation of the signature over the fragment may fail because the canonicalization by signature validation may include unnecessary namespaces into the fragment.

6 Security Considerations

6.1 Relationship to XML Digital Signatures

The application of both encryption and digital signatures over portions of an XML document can make subsequent decryption and signature verification difficult. In particular, when verifying a signature one must know whether the signature was computed over the encrypted or unencrypted form of elements.

A separate, but important, issue is introducing cryptographic vulnerabilities when combining digital signatures and encryption over a common XML element. Hal Finney has suggested that encrypting digitally signed data, while leaving the digital signature in the clear, may allow plaintext guessing attacks. This vulnerability can be mitigated by using secure hashes and the nonces in the text being processed.

In accordance with the requirements document [EncReq] the interaction of encryption and signing is an application issue and out of scope of the specification. However, we make the following recommendations:

1. When data is encrypted, any digest or signature over that data should be encrypted. This

satisfies the first issue in that only those signatures that can be seen can be validated. It also addresses the possibility of a plaintext guessing vulnerability, though it may not be possible to identify (or even know of) all the signatures over a given piece of data.
2. Employ the "decrypt-except" signature transform [XML-DSIG-Decrypt]. It works as follows: during signature transform processing, if you encounter a decrypt transform, decrypt all encrypted content in the document except for those excepted by an enumerated set of references.

Additionally, while the following warnings pertain to incorrect inferences by the user about the authenticity of information encrypted, applications should discourage user misapprehension by communicating clearly which information has integrity, or is authenticated, confidential, or non-repudiable when multiple processes (e.g., signature and encryption) and algorithms (e.g., symmetric and asymmetric) are used:

1. When an encrypted envelope contains a signature, the signature does not necessarily protect the authenticity or integrity of the ciphertext [Davis] .
2. While the signature secures plaintext it only covers that which is signed, recipients of encrypted messages must not infer integrity or authenticity of other unsigned information (e.g., headers) within the encrypted envelope, see [XML-DSIG, 8.1.1 Only What is Signed is Secure].

6.2 Information Revealed

Where a symmetric key is shared amongst multiple recipients, that symmetric key should *only* be used for the data intended for *all* recipients; even if one recipient is not directed to information intended (exclusively) for another in the same symmetric key, the information might be discovered and decrypted.

Additionally, application designers should be careful not to reveal any information in parameters or algorithm identifiers (e.g., information in a URI) that weakens the encryption.

6.3 Nonce and IV (Initialization Value or Vector)

An undesirable characteristic of many encryption algorithms and/or their modes is that the same plaintext when encrypted with the same key has the same resulting ciphertext. While this is unsurprising, it invites various attacks which are mitigated by including an arbitrary and non-repeating (under a given key) data with the plaintext prior to encryption. In encryption chaining modes this data is the first to be encrypted and is consequently called the IV (initialization value or vector).

Different algorithms and modes have further requirements on the characteristic of this information (e.g., randomness and secrecy) that affect the features (e.g., confidentiality and integrity) and their resistance to attack.

Given that XML data is redundant (e.g., Unicode encodings and repeated tags) and that attackers may know the data's structure (e.g., DTDs and schemas) encryption algorithms must be carefully implemented and used in this regard.

For the Cipher Block Chaining (CBC) mode used by this specification, the IV must not be reused for any key and should be random, but it need not be secret. Additionally, under this mode an adversary modifying the IV can make a known change in the plain text after decryption. This attack can be avoided by securing the integrity of the plain text data, for example by signing it.

6.4 Denial of Service

This specification permits recursive processing. For example, the following scenario is possible: `EncryptedKey` **A** requires `EncryptedKey` **B** to be decrypted, which itself requires `EncryptedKey` **A**! Or,

an attacker might submit an `EncryptedData` for decryption that references network resources that are very large or continually redirected. Consequently, implementations should be able to restrict arbitrary recursion and the total amount of processing and networking resources a request can consume.

6.5 Unsafe Content

XML Encryption can be used to obscure, via encryption, content that applications (e.g., firewalls, virus detectors, etc.) consider unsafe (e.g., executable code, viruses, etc.). Consequently, such applications must consider encrypted content to be as unsafe as the unsafest content transported in its application context. Consequently, such applications may choose to (1) disallow such content, (2) require access to the decrypted form for inspection, or (3) ensure that arbitrary content can be safely processed by receiving applications.

7 Conformance

An implementation is conformant to this specification if it successfully generates syntax according to the schema definitions and satisfies all MUST/REQUIRED/SHALL requirements, including algorithm support and processing. Processing requirements are specified over the roles of ***decryptor***, ***encryptor***, and their calling ***application***.

8 XML Encryption Media Type

8.1 Introduction

XML Encryption Syntax and Processing [XML-Encryption] specifies a process for encrypting data and representing the result in XML. The data may be arbitrary data (including an XML document), an XML element, or XML element content. The result of encrypting data is an XML Encryption element which contains or references the cipher data.

The `application/xenc+xml` media type allows XML Encryption applications to identify encrypted documents. Additionally it allows applications cognizant of this media-type (even if they are not XML Encryption implementations) to note that the media type of the decrypted (original) object might be a type other than XML.

8.2 application/xenc+xml Registration

This is a media type registration as defined in Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures [MIME-REG]

MIME media type name: application

MIME subtype name: xenc+xml

Required parameters: none

Optional parameters: charset

> The allowable and recommended values for, and interpretation of the charset parameter are identical to those given for 'application/xml' in section 3.2 of RFC 3023 [XML-MT].

Encoding considerations:

> The encoding considerations are identical to those given for 'application/xml' in section 3.2 of RFC 3023 [XML-MT].

Security considerations:

> See the [XML-Encryption] Security Considerations section.

Interoperability considerations: none

Published specification: [XML-Encryption]

Applications which use this media type:

> XML Encryption is device-, platform-, and vendor-neutral and is supported by a range of Web applications.

Additional Information:

> Magic number(s): none
>
> > Although no byte sequences can be counted on to consistently identify XML Encryption documents, they will be XML documents in which the root element's `QName`'s `LocalPart` is `'EncryptedData'` or '`EncryptedKey`' with an associated namespace name of 'http://www.w3.org/2001/04/xmlenc#'. The `application/xenc+xml` type name MUST only be used for data objects in which the root element is from the XML Encryption namespace. XML documents which contain these element types in places other than the root element can be described using facilities such as [XML-schema].
>
> File extension(s): .xml
>
> Macintosh File Type Code(s): "TEXT"

Person & email address to contact for further information:

> Joseph Reagle <reagle@w3.org>
>
> XENC Working Group <xml-encryption@w3.org>

Intended usage: COMMON

Author/Change controller:

The XML Encryption specification is a work product of the World Wide Web Consortium (W3C) which has change control over the specification.

9 Schema and Valid Examples

Schema
xenc-schema.xsd
Example
enc-example.xml (not cryptographically valid but exercises much of the schema)

10 References

TRIPLEDES
ANSI X9.52: Triple Data Encryption Algorithm Modes of Operation. 1998.
AES
NIST FIPS 197: Advanced Encryption Standard (AES). November 2001.

http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf

AES-WRAP

RFC3394: Advanced Encryption Standard (AES) Key Wrap Algorithm. J. Schaad and R. Housley. Informational, September 2002.

CMS-Algorithms

RFC3370: Cryptographic Message Syntax (CMS) Algorithms. R. Housley. Informational, February 2002.
http://www.ietf.org/rfc/rfc3370.txt

CMS-Wrap

RFC3217: Triple-DES and RC2 Key Wrapping. R. Housley. Informational, December 2001.
http://www.ietf.org/rfc/rfc3217.txt

Davis

Defective Sign & Encrypt in S/MIME, PKCS#7, MOSS, PEM, PGP, and XML. D. Davis. USENIX Annual Technical Conference. 2001.
http://www.usenix.org/publications/library/proceedings/usenix01/davis.html

DES

NIST FIPS 46-3: Data Encryption Standard (DES). October 1999.
http://csrc.nist.gov/publications/fips/fips46-3/fips46-3.pdf

EncReq

XML Encryption Requirements. J. Reagle. W3C Note, March 2002.
http://www.w3.org/TR/2002/NOTE-xml-encryption-req-20020304

ESDH

RFC 2631: Diffie-Hellman Key Agreement Method. E. Rescorla. Standards Track, 1999.
http://www.ietf.org/rfc/rfc2631.txt
http://www.w3.org/TR/2002/CR-xml-exc-c14n-20020212

Glossary

RFC 2828: Internet Security Glossary. R Shirey. Informational, May 2000.
http://www.ietf.org/rfc/rfc2828.txt

HMAC

RFC 2104: HMAC: Keyed-Hashing for Message Authentication. H. Krawczyk, M. Bellare, and R. Canetti. Informational, February 1997.
http://www.ietf.org/rfc/rfc2104.txt

HTTP

RFC 2616: Hypertext Transfer Protocol -- HTTP/1.1. J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Standards Track, June 1999.
http://www.ietf.org/rfc/rfc2616.txt

KEYWORDS

RFC 2119: Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. Best Current Practice, March 1997.
http://www.ietf.org/rfc/rfc2119.txt

MD5

RFC 1321: The MD5 Message-Digest Algorithm. R. Rivest. Informational, April 1992.
http://www.ietf.org/rfc/rfc1321.txt

MIME

RFC 2045: Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies. N. Freed and N. Borenstein. Standards Track, November 1996.
http://www.ietf.org/rfc/rfc2045.txt

MIME-REG

RFC 2048: Multipurpose Internet Mail Extensions (MIME) Part Four: Registration Procedures. N. Freed, J. Klensin, and J. Postel. Best Current Practice, November 1996.
http://www.ietf.org/rfc/rfc2048.txt

NFC

TR15, Unicode Normalization Forms. M. Davis and M. Dürst. Revision 18: November 1999.
http://www.unicode.org/unicode/reports/tr15/tr15-18.html.

NFC-Corrigendum

Corrigendum #2: Yod with Hiriq Normalization.

http://www.unicode.org/versions/corrigendum2.html.

prop1

XML Encryption strawman proposal. E. Simon and B. LaMacchia. Aug 2000. http://lists.w3.org/Archives/Public/xml-encryption/2000Aug/0001.html

prop2

Another proposal of XML Encryption. T. Imamura. Aug 2000. http://lists.w3.org/Archives/Public/xml-encryption/2000Aug/0005.html

prop3

XML Encryption Syntax and Processing. B. Dillaway, B. Fox, T. Imamura, B. LaMacchia, H. Maruyama, J. Schaad, and E. Simon. December 2000. http://lists.w3.org/Archives/Public/xml-encryption/2000Dec/att-0024/01-XMLEncryption_v01.html

PKCS1

RFC 2437: PKCS #1: RSA Cryptography Specifications Version 2.0. B. Kaliski and J. Staddon. Informational, October 1998. http://www.ietf.org/rfc/rfc2437.txt

RANDOM

RFC 1750: Randomness Recommendations for Security. D. Eastlake, S. Crocker, and J. Schiller. Informational, December 1994. http://www.ietf.org/rfc/rfc1750.txt

RIPEMD-160

CryptoBytes, Volume 3, Number 2. The Cryptographic Hash Function RIPEMD-160. RSA Laboratories. Autumn 1997. ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto3n2.pdf http://www.esat.kuleuven.ac.be/~cosicart/pdf/AB-9601/AB-9601.pdf

SHA

FIPS PUB 180-2. *Secure Hash Standard.* U.S. Department of Commerce/National Institute of Standards and Technology. http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf

SP800-56A

NIST Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography (Revised). March 2007

Tobin

R. Tobin. Infoset for external entities, XML Core mailing list, 2000 [W3C Member Only]. http://lists.w3.org/Archives/Member/w3c-xml-core-wg/2000OctDec/0054

UTF-16

RFC 2781: UTF-16, an encoding of ISO 10646. P. Hoffman and F. Yergeau. Informational, February 2000. http://www.ietf.org/rfc/rfc2781.txt

UTF-8

RFC 2279: UTF-8, a transformation format of ISO 10646F. F. Yergeau. Standards Track, January 1998. http://www.ietf.org/rfc/rfc2279.txt

URI

RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. T. Berners-Lee, R. Fielding, and L. Masinter. Standards Track, August 1998. http://www.ietf.org/rfc/rfc2396.txt http://www.ietf.org/rfc/rfc1738.txt http://www.ietf.org/rfc/rfc2141.txt

RFC 2611: URN Namespace Definition Mechanisms. Best Current Practices. Daigle, D. van Gulik, R. Iannella, P. Falstrom. June 1999. http://www.ietf.org/rfc/rfc2611.txt

X509v3

ITU-T Recommendation X.509 version 3 (1997). "Information Technology - Open Systems Interconnection - The Directory Authentication Framework" ISO/IEC 9594-8:1997.

XML

Extensible Markup Language (XML) 1.0 (Second Edition). T. Bray, J. Paoli, C. M.

Sperberg-McQueen, and E. Maler. W3C Recommendation, October 2000.

XML-Base

XML Base. J. Marsh. W3C Recommendation, June 2001.
http://www.w3.org/TR/2001/REC-xmlbase-20010627/

XML-C14N

Canonical XML. J. Boyer. W3C Recommendation, March 2001.
http://www.w3.org/TR/2001/REC-xml-c14n-20010315
http://www.ietf.org/rfc/rfc3076.txt

XML-exc-C14N

Exclusive XML Canonicalization. J. Boyer, D. Eastlake, and J. Reagle. W3C Recommendation, July 2002.
http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/

XML-DSIG

XML-Signature Syntax and Processing. D. Eastlake, J. Reagle, and D. Solo. W3C Recommendation, February 2002.
http://www.w3.org/TR/xmldsig-core/

XMLDSIG11

XML Signature Syntax and Processing Version 1.1. D. Eastlake, J. Reagle, D. Solo, F. Hirsch, T. Roessler. K. Yiu. W3C Working Draft, February 2009.

XML-DSIG-Decrypt

Decryption Transform for XML Signature. M. Hughes, T. Imamura and H. Maruyama. W3C Recommendation, December 2002.
http://www.w3.org/TR/2002/REC-xmlenc-decrypt-20021210

XML-Encryption

XML Encryption Syntax and Processing. D. Eastlake and J. Reagle. W3C Candidate Recommendation, December 2002.
http://www.w3.org/TR/2002/CR-xmlenc-core-20020802/

XML-Infoset

XML Information Set. J. Cowan and R. Tobin. W3C Recommendation, October 2001
http://www.w3.org/TR/2001/REC-xml-infoset-20011024/

XML-MT

RFC 3023: XML Media Types. M. Murata, S. St. Laurent, and D. Kohn. Informational, January 2001.
http://www.ietf.org/rfc/rfc2376.txt

XML-NS

Namespaces in XML. T. Bray, D. Hollander, and A. Layman. W3C Recommendation, January 1999.
http://www.w3.org/TR/1999/REC-xml-names-19990114/

XML-schema

XML Schema Part 1: Structures D. Beech, M. Maloney, and N. Mendelsohn. W3C Recommendation, May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-1-20010502/
XML Schema Part 2: Datatypes. P. Biron and A. Malhotra. W3C Recommendation, May 2001.
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

XPath

XML Path Language (XPath) Version 1.0. J. Clark and S. DeRose. W3C Recommendation, October 1999.
http://www.w3.org/TR/1999/REC-xpath-19991116