XML Signature Syntax and Processing Version 2.0

W3C Working Group Note 23 July 2015

This version:
http://www.w3.org/TR/2015/NOTE-xmldsig-core2-20150723/
Latest published version:
http://www.w3.org/TR/xmldsig-core2/
Latest editor's draft:
http://www.w3.org/2008/xmlsec/Drafts/xmldsig-core-20/
Previous version:
http://www.w3.org/TR/2013/NOTE-xmldsig-core2-20130411/
Editors:
Donald Eastlake, d3e3e3@gmail.com
Joseph Reagle, reagle@mit.edu
David Solo, dsolo@alum.mit.edu
Frederick Hirsch, frederick.hirsch@nokia.com (2nd edition, 1.1, 2.0)
Thomas Roessler, tlr@w3.org (2nd edition, 1.1)
Kelvin Yiu, kelviny@microsoft.com (1.1)
Pratik Datta, pratik.datta@oracle.com (2.0)
Scott Cantor, cantor.2@osu.edu (2.0)
Authors:
Mark Bartel, mbartel@adobe.com
John Boyer, boyerj@ca.ibm.com
Barb Fox, bfox@Exchange.Microsoft.com
Brian LaMacchia, bal@microsoft.com
Ed Simon, edsimon@xmlsec.com

Abstract

This informative W3C Working Group Note describes XML digital signature processing rules and syntax. XML Signatures provide integrity, message authentication, and/or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.

XML Signature 2.0 includes a new Reference processing model designed to address additional requirements including performance, simplicity and streamability. This "2.0 mode" model is significantly different than the XML Signature 1.x model in that it explicitly defines selection, canonicalization and verification steps for data processing and disallows generic transforms. XML Signature 2.0 is designed to be backward compatible through the inclusion of a "Compatibility Mode" which enables the XML Signature 1.x model to be used where necessary.

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/.

Note: On 23 July 2015 this Working Group Note has been updated to include a reference to the XML Signature 2.0 schema in the XSD Schema section. A diff from the previous Note publication is available.

The following is the status from the previous WG Note publication:

Note: On 23 April 2013, the reference to the "Additional XML Security URIs" RFC was updated. The Director previously authorized the publication knowing that the reference would be updated in a near future.

The XML Security Working Group has agreed not to progress this XML Signature Syntax and Processing Version 2.0 specification further as a Recommendation track document, electing to publish it as an informative Working Group Note. The Working Group has not performed interop testing on the material in this document.

Since the last publication as a Candidate Recommendation the following changes in XML Signature 1.1 have been also incorporated into this specification:

- Removed the `OCSPResponse` element originally proposed to be part of XML Signature 1.1 for optional inclusion in the `X509Data` element.
- Changed the references and language related to the use of Elliptic Curve algorithms in line with the XML Security Patent Advisory Group report. In conjunction with these changes, removed warning notes related to the use of Elliptic Curve algorithms,
- Added algorithm identifiers and information related to additional OPTIONAL algorithms: `SHA-224`, `ECDSA-SHA224`, `RSAwithSHA224` and `HMAC-SHA224`,
- Updated the security considerations text related to key lengths for the DSA and RSA algorithms. Changed DSA 1024 bit verification from REQUIRED to MAY,
- Added the Exclusive C14N omits comments algorithm as REQUIRED to implement, reflecting existing practice, and
- Updated the `KeyInfoReference` implementation requirement to SHOULD instead of `RetrievalMethod`.
- Corrected minor errors in examples (e.g. `ECDSAKeyValue`),
- Updated the formatting of examples and schema samples,
- Clarified the text in the bullet for the **library of functions** in section B.7.3 XPath Filtering, in response to Last Call issue LC-2721.
- Referenced the XML Signature Best Practices Note [XMLDSIG-BESTPRACTICES] from the introduction

Additional changes for this publication include the following:

- Changing the status to W3C Working Group Note, updating the abstract, status section and title page material accordingly.

- Updating the references, including replacing RFC 4051 with RFC 6931 which updates it.

A diff showing changes since the previous Candidate Recommendation is available.

Additional information related to the IPR status of XML Signature 2.0 related to Elliptic Curve algorithms is available at http://www.w3.org/2011/02/09-xmlsec-status.html.

This document was published by the XML Security Working Group as a Working Group Note. If you wish to make comments regarding this document, please send them to public-xmlsec@w3.org (subscribe, archives). All comments are welcome.

Publication as a Working Group Note 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.

This document is governed by the 1 August 2014 W3C Process Document.

Table of Contents

1. Introduction
   - 1.1 XML Signature 2.0 and 1.x compatibility
   - 1.2 Editorial and Conformance Conventions
   - 1.3 Design Philosophy
   - 1.4 Versions Namespaces and Identifiers
   - 1.5 Acknowledgements
2. Signature Overview and Examples
   - 2.1 Simple XML Signature 2.0 Example
   - 2.2 Detailed XML Signature 2.0 Example Using Ids
   - 2.3 Detailed XML Signature 2.0 Example using XPath
3. Conformance
   - 3.1 Common Conformance Requirements
     - 3.1.1 General Algorithm Identifier and Implementation Requirements
   - 3.2 XML Signature 2.0 Conformance
     - 3.2.1 XML Signature 2.0 Algorithm Identifiers and Implementation Requirements
   - 3.3 Compatibility Mode Conformance
     - 3.3.1 Compatibility Mode Algorithm Identifiers and Implementation Requirements
4. Processing Rules
   - 4.1 Signature Generation
   - 4.2 Reference Generation
   - 4.3 Core Validation
   - 4.4 Reference Check
   - 4.5 Reference Validation
   - 4.6 Signature Validation
5. Core Signature Syntax
   - 5.1 The `ds:CryptoBinary` Simple Type
   - 5.2 The `Signature` element
   - 5.3 The `SignatureValue` Element
   - 5.4 The `SignedInfo` Element
     - 5.4.1 The `CanonicalizationMethod` Element
     - 5.4.2 The `SignatureMethod` Element
     - 5.4.3 The `DigestMethod` Element
     - 5.4.4 The `DigestValue` Element
6. Referencing Content
   - 6.1 The `Reference` Element
     - 6.1.1 The `URI` Attribute
   - 6.2 The `Transforms` Element
   - 6.3 The `dsig2:Selection` Element
     - 6.3.1 Subtrees with Optional Exclusions
   - 6.4 The `dsig2:Verifications` Element
7. The `KeyInfo` Element
   - 7.1 The `KeyName` Element
   - 7.2 The `KeyValue` Element
     - 7.2.1 The `DSAKeyValue` Element
     - 7.2.2 The `RSAKeyValue` Element
     - 7.2.3 The `dsig11:ECKeyValue` Element
       - 7.2.3.1 Explicit Curve Parameters
       - 7.2.3.2 Compatibility with RFC 4050
   - 7.3 The `RetrievalMethod` Element
   - 7.4 The `X509Data` Element
     - 7.4.1 Distinguished Name Encoding Rules
   - 7.5 The `PGPData` Element
   - 7.6 The `SPKIData` Element
   - 7.7 The `MgmtData` Element
   - 7.8 XML Encryption `EncryptedKey` and `DerivedKey` Elements
   - 7.9 The `dsig11:DEREncodedKeyValue` Element
   - 7.10 The `dsig11:KeyInfoReference` Element
8. The `Object` Element
9. Additional Signature Syntax
   - 9.1 The `Manifest` Element
   - 9.2 The `SignatureProperties` Element
   - 9.3 Processing Instructions in Signature Elements

9.4 Comments in Signature Elements
10. Algorithms
10.1 Message Digests
10.1.1 SHA-1
10.1.2 SHA-224
10.1.3 SHA-256
10.1.4 SHA-384
10.1.5 SHA-512
10.2 Message Authentication Codes
10.2.1 HMAC
10.3 Signature Algorithms
10.3.1 DSA
10.3.2 RSA (PKCS#1 v1.5)
10.3.3 ECDSA
10.4 Canonicalization Algorithms
10.4.1 Canonical XML 2.0
10.5 The `Transform` Algorithm
10.6 `dsig2:Selection` Algorithms
10.6.1 Selection of XML Documents or Fragments
10.6.1.1 The `dsig2:IncludedXPath` Element
10.6.1.2 The `dsig2:ExcludedXPath` Element
10.6.1.3 The `dsig2:ByteRange` Element
10.6.2 Selection of External Binary Data
10.6.3 Selection of Binary Data within XML
10.7 The `dsig2:Verification` Types
10.7.1 DigestDataLength
10.7.2 PositionAssertion
10.7.3 IDAttributes
11. XML Canonicalization and Syntax Constraint Considerations
11.1 XML 1.0 Syntax Constraints, and Canonicalization
11.2 DOM/SAX Processing and Canonicalization
12. Security Considerations
12.1 Transforms
12.1.1 Only What is Signed is Secure
12.1.2 Only What is "Seen" Should be Signed
12.1.3 "See" What is Signed
12.2 Check the Security Model
12.3 Algorithms, Key Lengths, Certificates, Etc.
13. Schema
13.1 XSD Schema
A. Definitions
B. Compatibility Mode
B.1 "Compatibility Mode" Examples
B.1.1 Simple Example in "Compatibility Mode"
B.1.2 More on `Reference`
B.1.3 Extended Example (`Object` and `SignatureProperty`)
B.1.4 Extended Example (`Object` and `Manifest`)
B.2 Compatibility Mode Processing
B.2.1 Reference Generation in "Compatibility Mode"
B.2.2 Reference check in "Compatibility Mode"
B.2.3 Signature Validation in "Compatibility Mode"
B.2.4 Reference Validation in "Compatibility Mode"
B.3 Use of `CanonicalizationMethod` in "Compatibility Mode"
B.4 The `URI` Attribute in "Compatibility Mode"
B.4.1 The "Compatibility Mode" Reference Processing Model
B.4.2 "Compatibility Mode" Same-Document URI-References
B.5 "Compatibility Mode" Transforms and Processing Model
B.6 "Compatibility Mode" Canonicalization Algorithms
B.6.1 Canonical XML 1.0
B.6.2 Canonical XML 1.1
B.6.3 Exclusive XML Canonicalization 1.0
B.7 "Compatibility Mode" `Transform` Algorithms
B.7.1 Canonicalization
B.7.2 Base64
B.7.3 XPath Filtering
B.7.4 Signature Transform
B.7.5 XSLT Transform
B.8 Namespace Context and Portable Signatures
C. References
C.1 Normative references
C.2 Informative references

1. Introduction

This section is non-normative.

This document specifies XML syntax and processing rules for creating and representing digital signatures. XML Signatures can be applied to any digital content (data object), including XML. An XML Signature may be applied to the content of one or more resources. Enveloped or enveloping signatures are over data within the same XML document as the signature; detached signatures are over data external to the signature element. More specifically, this specification defines an XML signature element type and an XML signature application; conformance requirements for each are specified by way of schema definitions and prose respectively. This specification also includes other useful types that identify methods for referencing collections of resources, algorithms, and keying and management information.

The XML Signature is a method of associating a key with referenced data (octets); it does not normatively specify how keys are associated with persons

or institutions, nor the meaning of the data being referenced and signed. Consequently, while this specification is an important component of secure XML applications, it itself is not sufficient to address all application security/trust concerns, particularly with respect to using signed XML (or other data formats) as a basis of human-to-human communication and agreement. Such an application must specify additional key, algorithm, processing and rendering requirements. For further information, please see section 12. Security Considerations.

XML Signature 2.0 includes a new Reference processing model designed to address additional requirements including performance, simplicity and streamability. This "2.0 mode" model is significantly different than the XML Signature 1.x model in that it explicitly defines selection, canonicalization and verification steps for data processing and disallows generic transforms. XML Signature 2.0 is designed to be backward compatible through the inclusion of a "Compatibility Mode" which enables the XML Signature 1.x model to be used where necessary.

The Working Group encourages implementers and developers to read *XML Signature Best Practices* [XMLDSIG-BESTPRACTICES]. It contains a number of best practices related to the use of XML Signature, including implementation considerations and practical ways of improving security.

1.1 XML Signature 2.0 and 1.x compatibility

This section is non-normative.

This specification defines XML Signature 2.0 which differs from XML Signature 1.x in some specific areas, in particular the use of various transform algorithms versus a fixed 2.0 transform that implies the use of Selection and Verification steps in conjunction with `ds:Reference` processing, the corresponding disuse of the `URI` `ds:Reference` attribute, the use of Canonical XML 2.0 [XML-C14N20] in place of other canonicalization algorithms, and updates to the required algorithms and other changes.

This specification defines a "Compatibility Mode" that supports an XML Signature 1.x mode of operation. Compliance and other aspects unique to "Compatibility Mode" are outlined in section B. Compatibility Mode.

The body of the document refers to the syntax and processing model for the new 2.0 mode of operation, referred to as "XML Signature 2.0" in the document. Use of the "Compatibility Mode" is noted explicitly when required.

1.2 Editorial and Conformance Conventions

For readability, brevity, and historic reasons this document uses the term "signature" to generally refer to digital authentication values of all types. Obviously, the term is also strictly used to refer to authentication values that are based on public keys and that provide signer authentication. When specifically discussing authentication values based on symmetric secret key codes we use the terms authenticators or authentication codes. (See section 12.2 Check the Security Model.)

This specification provides normative XML Schemas [XMLSCHEMA-1], [XMLSCHEMA-2]. The full normative grammar is defined by the XSD schemas and the normative text in this specification. The standalone XSD schema files are authoritative in case there is any disagreement between them and the XSD schema portions in this specification.

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].

> "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 key words 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 Namespaces in XML specification [XML-NAMES] is described as "REQUIRED."

1.3 Design Philosophy

The design philosophy and requirements of this specification are addressed in the original XML-Signature Requirements document [XMLDSIG-REQUIREMENTS], the XML Security 1.1 Requirements document [XMLSEC11-REQS], and the XML Security 2.0 Requirements document [XMLSEC2-REQS].

1.4 Versions Namespaces and Identifiers

This specification makes use of XML namespaces, and uses Uniform Resource Identifiers [URI] to identify resources, algorithms, and semantics.

Implementations of this specification MUST use the following XML namespace URIs:

URI	namespace prefix	XML internal entity
`http://www.w3.org/2000/09/xmldsig#`	*default namespace*, `ds:`, `dsig:`	`<!ENTITY dsig "http://www.w3.org/2000/09/xmldsig#">`
`http://www.w3.org/2009/xmldsig11#`	`dsig11:`	`<!ENTITY dsig11 "http://www.w3.org/2009/xmldsig11#">`
`http://www.w3.org/2010/xmldsig2#`	`dsig2:`	`<!ENTITY dsig2 "http://www.w3.org/2010/xmldsig2#">`

While implementations MUST support XML and XML namespaces, and while use of the above namespace URIs is REQUIRED, the namespace prefixes and entity declarations given are merely editorial conventions used in this document. Their use by implementations is OPTIONAL.

These namespace URIs are also used as the prefix for algorithm identifiers that are under control of this specification. For resources not under the control of this specification, we use the designated Uniform Resource Names [URN], [RFC3406] or Uniform Resource Identifiers [URI] defined by the relevant normative external specification.

The `http://www.w3.org/2000/09/xmldsig#` (`dsig:`) namespace was introduced in the first edition of this specification, and `http://www.w3.org/2009/xmldsig11#` (`dsig11:`) namespace was introduced in 1.1. This version does not coin any new elements or algorithm identifiers in those namespaces; instead, the `http://www.w3.org/2010/xmldsig2#` (`dsig2:`) namespace is used.

This specification uses algorithm identifiers in the namespace `http://www.w3.org/2001/04/xmldsig-more#` that were originally coined in [RFC6931]. RFC 6931 associates these identifiers with specific algorithms. Implementations of this specification MUST be fully interoperable with the algorithms specified in [RFC6931], but MAY compute the requisite values through any technique that leads to the same output.

Examples of items in various namespaces include:

`SignatureProperties` is identified and defined by the `disg:` namespace
`http://www.w3.org/2000/09/xmldsig#SignatureProperties`
`ECKeyValue` is identified and defined by the `dsig11:` namespace
`http://www.w3.org/2009/xmldsig11#ECKeyValue`
XSLT is identified and defined by an external URI
`http://www.w3.org/TR/1999/REC-xslt-19991116`
SHA1 is identified via this specification's namespace and defined via a normative reference [FIPS-180-3]
http://www.w3.org/2001/04/xmlenc#sha256
FIPS PUB 180-3. *Secure Hash Standard.* U.S. Department of Commerce/National Institute of Standards and Technology.
`Selection` is identified and defined by the `dsig2:` namespace
`http://www.w3.org/2010/xmldsig2#Selection`

No provision is made for an explicit version number in this syntax. If a future version of this specification requires explicit versioning of the document format, a different namespace will be used.

1.5 Acknowledgements

The contributions of the members of the XML Signature Working Group to the first edition specification are gratefully acknowledged: Mark Bartel, Adobe, was Accelio (Author); John Boyer, IBM (Author); Mariano P. Consens, University of Waterloo; John Cowan, Reuters Health; Donald Eastlake 3rd, Motorola; (Chair, Author/Editor); Barb Fox, Microsoft (Author); Christian Geuer-Pollmann, University Siegen; Tom Gindin, IBM; Phillip Hallam-Baker, VeriSign Inc; Richard Himes, US Courts; Merlin Hughes, Baltimore; Gregor Karlinger, IAIK TU Graz; Brian LaMacchia, Microsoft (Author); Peter Lipp, IAIK TU Graz; Joseph Reagle, NYU, was W3C (Chair, Author/Editor); Ed Simon, XMLsec (Author); David Solo, Citigroup (Author/Editor); Petteri Stenius, Capslock; Raghavan Srinivas, Sun; Kent Tamura, IBM; Winchel Todd Vincent III, GSU; Carl Wallace, Corsec Security, Inc.; Greg Whitehead, Signio Inc.

As are the first edition Last Call comments from the following:

- Dan Connolly, W3C
- Paul Biron, Kaiser Permanente, on behalf of the XML Schema WG.
- Martin J. Duerst, W3C; and Masahiro Sekiguchi, Fujitsu; on behalf of the Internationalization WG/IG.
- Jonathan Marsh, Microsoft, on behalf of the Extensible Stylesheet Language WG.

The following members of the XML Security Specification Maintenance Working Group contributed to the second edition: Juan Carlos Cruellas, Universitat Politècnica de Catalunya; Pratik Datta, Oracle Corporation; Phillip Hallam-Baker, VeriSign, Inc.; Frederick Hirsch, Nokia, (Chair, Editor); Konrad Lanz, Applied Information processing and Kommunications (IAIK); Hal Lockhart, BEA Systems, Inc.; Robert Miller, MITRE Corporation; Sean Mullan, Sun Microsystems, Inc.; Bruce Rich, IBM Corporation; Thomas Roessler, W3C/ERCIM, (Staff contact, Editor); Ed Simon, W3C Invited Expert; Greg Whitehead, HP.

Contributions for version 1.1 were received from the members of the XML Security Working Group: Scott Cantor, Juan Carlos Cruellas, Pratik Datta, Gerald Edgar, Ken Graf, Phillip Hallam-Baker, Brad Hill, Frederick Hirsch (Chair, Editor), Brian LaMacchia, Konrad Lanz, Hal Lockhart, Cynthia Martin, Rob Miller, Sean Mullan, Shivaram Mysore, Magnus Nyström, Bruce Rich, Thomas Roessler, Ed Simon, Chris Solc, John Wray, Kelvin Yiu.

2. Signature Overview and Examples

This section is non-normative.

This section provides an overview and examples of XML digital signature syntax. The specific processing is given in section 4. Processing Rules. The formal syntax is found in section 5. Core Signature Syntax and section 9. Additional Signature Syntax.

In this section, an informal representation and examples are used to describe the structure of the XML signature syntax. This representation and examples may omit attributes, details and potential features that are fully explained later.

XML Signatures are applied to arbitrary digital content (data objects) via an indirection. Data objects are digested, the resulting value is placed in an element (with other information) and that element is then digested and cryptographically signed. XML digital signatures are represented by the `Signature` element which has the following structure (where "?" denotes zero or one occurrence; "+" denotes one or more occurrences; and "*" denotes zero or more occurrences):

EXAMPLE 1

```
<Signature ID?>
  <SignedInfo>
    <CanonicalizationMethod />
    <SignatureMethod />
   (<Reference URI? >
     (<Transforms>)?
      <DigestMethod>
      <DigestValue>
    </Reference>)+
  </SignedInfo>
  <SignatureValue>
 (<KeyInfo>)?
 (<Object ID?>)*
</Signature>
```

Signatures are related to data objects via URIs [URI]. Within an XML document, signatures are related to local data objects via fragment identifiers. Such local data can be included within an enveloping signature or can enclose an enveloped signature. Detached signatures are over external network resources or local data objects that reside within the same XML document as sibling elements; in this case, the signature is neither enveloping (signature is parent) nor enveloped (signature is child). Since a `Signature` element (and its `Id` attribute value/name) may co-exist or be combined with other elements (and their IDs) within a single XML document, care should be taken in choosing names such that there are no subsequent collisions that violate the ID uniqueness validity constraint [XML10].

2.1 Simple XML Signature 2.0 Example

This section is non-normative.

This is the same example an as provided for the XML Signature 1.x, but for XML Signature 2.0. The only differences are in the `CanonicalizationMethod`

and `Reference` portions. The line numbers in this example match up with the line numbers in the "Compatibility Mode" example.

EXAMPLE 2

```
[s01] <Signature Id="MyFirstSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
[s02]   <SignedInfo>
[s03]   <CanonicalizationMethod Algorithm="http://www.w3.org/2010/xml-c14n2"/>
[s04]   <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
[s05]   <Reference>
[s06]     <Transforms>
[s07]       <Transform Algorithm="http://www.w3.org/2010/xmldsig2#transform">
[s07a]        <dsig2:Selection Algorithm="http://www.w3.org/2010/xmldsig2#xml"
 xmlns:dsig2="http://www.w3.org/2010/xmldsig2#"
 URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126">
>
[s07b]        </dsig2:Selection>
[s07c]        <CanonicalizationMethod Algorithm="http://www.w3.org/2010/xml-c14n2"/>
[s07d]      </Transform>
[s08]     </Transforms>
[s09]     <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[s10]     <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...</DigestValue>
[s11]   </Reference>
[s12] </SignedInfo>
[s13]   <SignatureValue>...</SignatureValue>
[s14]   <KeyInfo>
[s15a]    <KeyValue>
[s15b]      <DSAKeyValue>
[s15c]        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
[s15d]      </DSAKeyValue>
[s15e]    </KeyValue>
[s16]   </KeyInfo>
[s17] </Signature>
```

`[s03]` In XML Signature 2.0 the Canonicalization Method URI should be Canonical XML 2.0 (or a later version) and all the parameters for Canonical XML 2.0 should be present as subelements of this element [XML-C14N20].

`[s05-s08]` Note XML Signature 2.0 does not use various transforms, instead each reference object has two parts - a `dsig2:Selection` element to choose the data object to be signed, and a `Canonicalization` element to convert the data object to a canonicalized octet stream. To fit in these two elements, without breaking backwards compatibility with the 1.0 schema, these elements have been put inside a special `Transform` with URI `http://www.w3.org/2010/xmldsig2#transform`. In XML Signature 2.0 the `Transforms` element will contain only this particular fixed `Transform`.

`[s05]` In XML Signature 2.0, the `URI` attribute is omitted from the `Reference`. Instead it can be found in the `dsig2:Selection`.

`[s07a-s07b]` The `dsig2:Selection` element identifies the data object to be signed. This specification identifies only two types, "xml" and "binary", but user specified types are also possible. For example a new type "database-rows" could be defined to select rows from a database for signing. Usually a URI and a few other bits of information are used to identify the data object, but the URI is not required; for example, the "xml" type can identify a local document subset by using an XPath.

`[s07c]` The `CanonicalizationMethod` element provides the mechanism to convert the data object into a canonicalized octet stream. This specification addresses only canonicalization for xml data. Other forms of canonicalization can be defined - e.g. a scheme for signing mime attachments could define a canonicalization for mime headers and data. The output of the canonicalization is digested.

2.2 Detailed XML Signature 2.0 Example Using Ids

The followed detailed example shows XML Signature 2.0 in the context of Web Services Security [WS-SECURITY11], showing how the SOAP body can be referenced using an Id in XML Signature 2.0. This example shows more detail than the previous Simple XML Signature 2.0 Example.

Note: This example (and the next example using XPath) show the use of XML Signature 2.0 in the context of Web Services Security. This is illustrative of how a 2.0 signature could be substituted for an 1.x Signature, but has not been standardized in Web Services Security so should only be considered illustrative.

EXAMPLE 3

```
[ i01 ] <?xml version="1.0" encoding="UTF-8"?>
[ i02 ] <soap:Envelope
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
[ i03 ]   <soap:Header>
[ i04 ]     <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd">
[ i05 ]       <wsse:BinarySecurityToken wsu:Id="MyID"
[ i06 ] ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
[ i07 ] EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#Base64Binary">
[ i08 ]         MIIEZzCCA9CgAwIBAgIQEmtJZc0..
[ i09 ]       </wsse:BinarySecurityToken>
[ i10 ]       <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
[ i11 ]         <ds:SignedInfo>
[ i12 ]           <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2010/xml-c14n2"
[ i13 ] xmlns:c14n2="http://www.w3.org/2010/xml-c14n2">
[ i14 ]             <c14n2:IgnoreComments>true</c14n2:IgnoreComments>
[ i15 ]             <c14n2:TrimTextNodes>false</c14n2:TrimTextNodes>
[ i16 ]             <c14n2:PrefixRewrite>none</c14n2:PrefixRewrite>
[ i17 ]             <c14n2:QNameAware/>
[ i18 ]           </ds:CanonicalizationMethod>
[ i19 ]           <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
[ i20 ]           <ds:Reference>
[ i21 ]             <ds:Transforms>
[ i22 ]               <ds:Transform
Algorithm="http://www.w3.org/2010/xmldsig2#newTransformModel"
xmlns:dsig2="http://www.w3.org/2010/xmldsig2#">
[ i23 ]                 <dsig2:Selection Algorithm="http://www.w3.org/2010/xmldsig2#xml" URI="#MsgBody" />
[ i24 ]                 <dsig2:Canonicalization >
[ i25 ]                   <c14n2:IgnoreComments>true</c14n2:IgnoreComments>
[ i26 ]                   <c14n2:TrimTextNodes>true</c14n2:TrimTextNodes>
```

```
[ i27 ]                  <c14n2:PrefixRewrite>sequential</c14n2:PrefixRewrite>
[ i28 ]                  <c14n2:QNameAware/>
[ i29 ]                </dsig2:Canonicalization>
[ i30 ]                <dsig2:Verifications>
[ i31 ]                  <dsig2:Verification DigestDataLength="308"/>
[ i32 ]                </dsig2:Verifications>
[ i33 ]              </ds:Transform>
[ i34 ]            </ds:Transforms>
[ i35 ]            <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[ i36 ]            <ds:DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...</ds:DigestValue>
[ i37 ]          </ds:Reference>
[ i38 ]        </ds:SignedInfo>
[ i39 ]        <ds:SignatureValue>kdutrEsAEw56Sefgs34...</ds:SignatureValue>
[ i40 ]        <ds:KeyInfo>
[ i41 ]          <ds:KeyValue>
[ i42 ]            <wsse:SecurityTokenReference>
[ i43 ]              <wsse:Reference URI="#MyID"/>
[ i44 ]            </wsse:SecurityTokenReference>
[ i45 ]          </ds:KeyValue>
[ i46 ]        </ds:KeyInfo>
[ i47 ]      </ds:Signature>
[ i48 ]    </wsse:Security>
[ i49 ]  </soap:Header>
[ i50 ]  <soap:Body wsu:Id="MsgBody">
[ i51 ]    <ex:operation xmlns:ex="http://www.example.com/">
[ i52 ]      <ex:param1>42</ex:param1>
[ i53 ]      <ex:param2>43</ex:param2>
[ i54 ]    </ex:operation>
[ i55 ]  </soap:Body>
[ i56 ] </soap:Envelope>
```

[i05-i09] The `wsse:BinarySecurityToken` is a Web Services Security mechanism to convey key information needed for signature processing, in this case an X.509v3 certificate.

[i12-i18] This example shows explicit choices for parameters of the `ds:CanonicalizationMethod` rather than relying on implicit defaults. These canonicalization choices are for the canonicalization of `ds:SignedInfo` using Canonical XML 2.0 [XML-C14N20].

[i14] The `c14n2:IgnoreComments` parameter is set to `true`, the default, meaning that comments will be ignored.

[i15] The `c14n2:TrimTextNodes` parameter is set to `false`, so white space will be preserved.

[i16] The `c14n2:PrefixRewrite` parameter is set to `none`, the default, meaning that no prefixes will be rewritten.

[i17] The `c14n2:QNameAware` parameter is set to the empty set, the default, meaning that no QNames require special processing.

[i23] The `dsig2:Selection URI` parameter is set to `#MsgBody` meaning that the element with the corresponding Id (in this case `wsu:Id`) will be selected.

[i24-i29] The `dsig2:Canonicalization` element again has parameters set explicitly for `ds:Reference` canonicalization.

[i30-i33] This example uses the new ability in XML Signature 2.0 for a verifier to receive constraint information that can be used to verify correctness of the information received, to mitigate against attacks. The `dsig2:Verifications` element contains this verification information. In this case the length of the `ds:Reference` data that was digested is conveyed.

[i42-i44] Web Services Security uses its `SecurityTokenReference` mechanism to reference key information conveyed in tokens, such as an X.509 certificate. In this example this mechanism is used to reference the binary security token at using the `MyID` Id.

[i50] The `soapBody` has a `wsu:Id` attribute which is used by the `ds:Reference URI` attribute to reference the element.

2.3 Detailed XML Signature 2.0 Example using XPath

The followed detailed example shows use of XML Signature 2.0 in a Web Services Security example similar to the previous example using an Id reference, but here uses an XPath expression to help mitigate the possibility of wrapping attacks. In this case the `soap:Body` is signed, but the `ex:param2` is omitted from the signature. This could correspond to a case where the the first parameter is known to be invariant end-end while the second parameter might be expected to change as the SOAP message traverses SOAP intermediaries, so is omitted from the signature.

EXAMPLE 4

```
[ p01 ] <?xml version="1.0" encoding="UTF-8"?>
[ p02 ] <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/" xmlns:ex="http://www.example.com/">
[ p03 ]   <soap:Header>
[ p04 ]     <wsse:Security xmlns:wsse="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-secext-1.0.xsd"
[ p05 ] xmlns:wsu="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-wssecurity-utility-1.0.xsd">
[ p06 ]       <wsse:BinarySecurityToken wsu:Id="MyID"
[ p07 ] ValueType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#X509v3"
[ p08 ] EncodingType="http://docs.oasis-open.org/wss/2004/01/oasis-200401-wss-x509-token-profile-1.0#Base64Binary">
[ p09 ]         MIIEZzCCA9CgAwIBAgIQEmtJZc0..
[ p10 ]       </wsse:BinarySecurityToken>
[ p11 ]       <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
[ p12 ]         <ds:SignedInfo>
[ p13 ]           <ds:CanonicalizationMethod Algorithm="http://www.w3.org/2010/xml-c14n2"
[ p14 ] xmlns:c14n2="http://www.w3.org/2010/xml-c14n2">
[ p15 ]             <c14n2:IgnoreComments>true</c14n2:IgnoreComments>
[ p16 ]             <c14n2:TrimTextNodes>false</c14n2:TrimTextNodes>
[ p17 ]             <c14n2:PrefixRewrite>none</c14n2:PrefixRewrite>
[ p18 ]             <c14n2:QNameAware/>
[ p19 ]           </ds:CanonicalizationMethod>
[ p20 ]           <ds:SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
[ p21 ]           <ds:Reference>
[ p22 ]             <ds:Transforms>
[ p23 ]               <ds:Transform Algorithm="http://www.w3.org/2010/xmldsig2#newTransformModel" xmlns:dsig2="http://www.w3.org/2010/xmldsig2#">
[ p24 ]                 <dsig2:Selection Algorithm="http://www.w3.org/2010/xmldsig2#xml" URI="">
[ p25 ]                   <dsig2:IncludedXPath>/soap:Envelope/soap:Body[1]</dsig2:IncludedXPath>
[ p26 ]                   <dsig2:ExcludedXPath>
```

```
[ p27 ]                    /soap:Envelope/soap:Body[1]/ex:operation[1]/ex:param2[1]
[ p28 ]                  </dsig2:ExcludedXPath>
[ p29 ]                </dsig2:Selection>
[ p30 ]                <dsig2:Canonicalization >
[ p31 ]                  <c14n2:IgnoreComments>true</c14n2:IgnoreComments>
[ p32 ]                  <c14n2:TrimTextNodes>true</c14n2:TrimTextNodes>
[ p33 ]                  <c14n2:PrefixRewrite>sequential</c14n2:PrefixRewrite>
[ p34 ]                  <c14n2:QNameAware/>
[ p35 ]                </dsig2:Canonicalization>
[ p36 ]                <dsig2:Verifications>
[ p37 ]                  <dsig2:Verification DigestDataLength="169"/>
[ p38 ]                </dsig2:Verifications>
[ p39 ]              </ds:Transform>
[ p40 ]            </ds:Transforms>
[ p41 ]            <ds:DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[ p42 ]            <ds:DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...</ds:DigestValue>
[ p43 ]          </ds:Reference>
[ p44 ]        </ds:SignedInfo>
[ p45 ]        <ds:SignatureValue>kdutrEsAEw56Sefgs34...</ds:SignatureValue>
[ p46 ]        <ds:KeyInfo>
[ p47 ]          <ds:KeyValue>
[ p48 ]            <wsse:SecurityTokenReference>
[ p49 ]              <wsse:Reference URI="#MyID"/>
[ p50 ]            </wsse:SecurityTokenReference>
[ p51 ]          </ds:KeyValue>
[ p52 ]        </ds:KeyInfo>
[ p53 ]      </ds:Signature>
[ p54 ]    </wsse:Security>
[ p55 ]  </soap:Header>
[ p56 ]  <soap:Body>
[ p57 ]    <ex:operation>
[ p58 ]      <ex:param1>42</ex:param1>
[ p59 ]      <ex:param2>43</ex:param2>
[ p60 ]    </ex:operation>
[ p61 ]  </soap:Body>
[ p62 ] </soap:Envelope>
```

[p24] In this case the `URI` attribute of the `Reference` element is `""` as XPath is used rather than an Id based reference.

[p25] The `dsig2:IncludedXPath` element includes an XPath expression to reference the `soap:Body` element. Note that this expression is written to reference the specific `soap:Body` to mitigate wrapping attacks. The XPath expression is an XML Security 2.0 profile of XPath 1.0 [XMLDSIG-XPATH].

[p26] The `dsig2:ExcludedXPath` element specifies that the `ex:operation[1]/ex:param2[1]` child of the `soap:Body` not be included in the signature. The XPath expression specifies the exact instance to avoid wrapping attacks.

3. Conformance

This entire document is informative, published as a W3C Working Group Note. Thus this section should only be considered indicative as to how the material in this document could be interpreted.

An implementation that conforms to this specification MUST be conformant to XML Signature 2.0 mode, and MAY be conformant to XML Signature 1.1 Compatibility Mode.

3.1 Common Conformance Requirements

The following conformance requirements must be met by all implementations, including those in compatibility mode.

3.1.1 General Algorithm Identifier and Implementation Requirements

This section identifies algorithm conformance requirements applicable to both 2.0 and compatibility mode.

Algorithms are identified by URIs that appear as an attribute to the element that identifies the algorithms' role (`DigestMethod`, `Transform`, `SignatureMethod`, or `CanonicalizationMethod`). All algorithms used herein take parameters but in many cases the parameters are implicit. For example, a `SignatureMethod` is implicitly given two parameters: the keying info and the output of `CanonicalizationMethod`. Explicit additional parameters to an algorithm appear as content elements within the algorithm role element. Such parameter elements have a descriptive element name, which is frequently algorithm specific, and MUST be in the XML Signature namespace or an algorithm specific namespace.

This specification defines a set of algorithms, their URIs, and requirements for implementation. Requirements are specified over implementation, not over requirements for signature use. Furthermore, the mechanism is extensible; alternative algorithms may be used by signature applications.

Digest

Required

1. SHA1 (Use is DISCOURAGED; see SHA-1 Warning) http://www.w3.org/2000/09/xmldsig#sha1
2. SHA256 http://www.w3.org/2001/04/xmlenc#sha256

Optional

1. SHA224
   http://www.w3.org/2001/04/xmldsig-more#sha224
2. SHA384 http://www.w3.org/2001/04/xmldsig-more#sha384
3. SHA512 http://www.w3.org/2001/04/xmlenc#sha512

Encoding

Required

1. base64 (*note)
   http://www.w3.org/2000/09/xmldsig#base64

MAC

Required

1. HMAC-SHA1 (Use is DISCOURAGED; see SHA-1 Warning) http://www.w3.org/2000/09/xmldsig#hmac-sha1
2. HMAC-SHA256 http://www.w3.org/2001/04/xmldsig-more#hmac-sha256

Recommended

1. HMAC-SHA384 http://www.w3.org/2001/04/xmldsig-more#hmac-sha384
2. HMAC-SHA512 http://www.w3.org/2001/04/xmldsig-more#hmac-sha512

Optional

1. HMAC-SHA224
   http://www.w3.org/2001/04/xmldsig-more#hmac-sha224

Signature

Required

1. RSAwithSHA256 http://www.w3.org/2001/04/xmldsig-more#rsa-sha256 [RFC6931]
2. ECDSAwithSHA256 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256 [RFC6931]
3. DSAwithSHA1 (**signature verification**) http://www.w3.org/2000/09/xmldsig#dsa-sha1 [RFC6931]

Recommended

1. RSAwithSHA1 (**signature verification**; use for signature generation is DISCOURAGED; see SHA-1 Warning)
   http://www.w3.org/2000/09/xmldsig#rsa-sha1

Optional

1. RSAwithSHA224 http://www.w3.org/2001/04/xmldsig-more#rsa-sha224 [section 10.3.2 RSA (PKCS#1 v1.5)]
2. RSAwithSHA384 http://www.w3.org/2001/04/xmldsig-more#rsa-sha384 [section 10.3.2 RSA (PKCS#1 v1.5)]
3. RSAwithSHA512 http://www.w3.org/2001/04/xmldsig-more#rsa-sha512
4. ECDSAwithSHA1 (Use is DISCOURAGED; see SHA-1 Warning) http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha1 [section 10.3.3 ECDSA]
5. ECDSAwithSHA224
   http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha224 [section 10.3.3 ECDSA]
6. ECDSAwithSHA384 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384 [section 10.3.3 ECDSA]
7. ECDSAwithSHA512 http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512 [section 10.3.3 ECDSA]
8. DSAwithSHA1 (**signature generation**) http://www.w3.org/2000/09/xmldsig#dsa-sha1
9. DSAwithSHA256 http://www.w3.org/2009/xmldsig11#dsa-sha256

*note: Note that the same URI is used to identify base64 both in "encoding" context (e.g. within the `Object` element) as well as in "transform" context (when identifying a base64 transform).

3.2 XML Signature 2.0 Conformance

An implementation that conforms to this specification MUST support XML Signature 2.0 operation and conform to the following features when not operating in compatibility mode:

- MUST support the required steps of Signature generation, including the generation of Reference elements and the SignatureValue over SignedInfo as outlined in section 4.1 Signature Generation.
- MUST support the required steps of core validation as outlined in section 4.3 Core Validation.
- MUST support required XML Signature 2.0 Reference generation as outlined in section Not found 'sec-ReferenceGeneration-2.0'.
- MUST conform to the syntax as outlined in text of this specification
- MUST NOT have a `URI` attribute in a `Reference` element
- Every `Reference` element MUST have a single `Transforms` element and that element MUST contain exactly one `Transform` element with an `Algorithm` of `"http://www.w3.org/2010/xmldsig2#transform"`
- The result of processing each `Reference` MUST be an octet stream with the digest algorithm applied to the resulting data octets
- `RetrievalMethod` SHOULD NOT be used; `dsig11:KeyInfoReference` SHOULD be used instead.

3.2.1 XML Signature 2.0 Algorithm Identifiers and Implementation Requirements

This section identifies algorithms used with the XML digital signature specification. Entries contain the identifier to be used in `Signature` elements, a reference to the formal specification, and definitions, where applicable, for the representation of keys and the results of cryptographic operations.

Note that the algorithms required for 2.0 conformance are fewer than for compatibility mode, and that some algorithms required or optional are disallowed in 2.0.

Canonicalization

Required

1. Canonical XML 2.0

Transform

Required

1. XML Signature 2.0 Transform - http://www.w3.org/2010/xmldsig2#transform

Selection

Required

1. XML Documents or Fragments - http://www.w3.org/2010/xmldsig2#xml
2. External Binary Data - http://www.w3.org/2010/xmldsig2#binaryExternal
3. Selection of Binary Data within XML - http://www.w3.org/2010/xmldsig2#binaryfromBase64

Verification

Optional

1. DigestDataLength - http://www.w3.org/2010/xmldsig2#DigestDataLength
2. PositionAssertion - http://www.w3.org/2010/xmldsig2#PositionAssertion
3. IDAttributes - http://www.w3.org/2010/xmldsig2#IDAttributes

3.3 Compatibility Mode Conformance

An implementation that conforms to this specification MAY be conformant to Compatibility Mode. To conform to compatibility mode conformance with the following is required as well as conformance to common conformance requirements described in section 3.1 Common Conformance Requirements.

3.3.1 Compatibility Mode Algorithm Identifiers and Implementation Requirements

The following algorithm support is required for compatibility mode (in addition to those required for all modes).

Canonicalization

Required

1. Canonical XML 1.0 (omits comments) http://www.w3.org/TR/2001/REC-xml-c14n-20010315
2. Canonical XML 1.1 (omits comments) http://www.w3.org/2006/12/xml-c14n11
3. Exclusive XML Canonicalization 1.0 (omits comments) http://www.w3.org/2001/10/xml-exc-c14n#

Recommended

1. Canonical XML 1.0 with Comments http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
2. Canonical XML 1.1 with Comments http://www.w3.org/2006/12/xml-c14n11#WithComments
3. Exclusive XML Canonicalization 1.0 with Comments http://www.w3.org/2001/10/xml-exc-c14n#WithComments

Transform

Required

1. base64 (*note)
   http://www.w3.org/2000/09/xmldsig#base64
2. Enveloped Signature (**note)
   http://www.w3.org/2000/09/xmldsig#enveloped-signature

Recommended

1. XPath http://www.w3.org/TR/1999/REC-xpath-19991116
2. XPath Filter 2.0 http://www.w3.org/2002/06/xmldsig-filter2

Optional

1. XSLT http://www.w3.org/TR/1999/REC-xslt-19991116

**note: The Enveloped Signature transform removes the `Signature` element from the calculation of the signature when the signature is within the content that it is being signed. This MAY be implemented via the XPath specification specified in 6.6.4: Enveloped Signature Transform; it MUST have the same effect as that specified by the XPath Transform.

When using transforms, we RECOMMEND selecting the least expressive choice that still accomplishes the needs of the use case at hand: Use of XPath filter 2.0 is recommended over use of XPath filter. Use of XPath filter is recommended over use of XSLT.

Note: Implementation requirements for the XPath transform may be downgraded to OPTIONAL in a future version of this specification.

4. Processing Rules

The sections below describe the operations to be performed as part of signature generation and validation.

4.1 Signature Generation

The REQUIRED steps include the generation of `Reference` elements and the `SignatureValue` over `SignedInfo`.

1. Create `SignedInfo` element with `SignatureMethod`, `CanonicalizationMethod` and `Reference`(s).
2. Canonicalize and then calculate the `SignatureValue` over `SignedInfo` based on algorithms specified in `SignedInfo`. For XML Signature 2.0 signatures (i.e. not XML Signature 1.x or "Compatibility Mode" signatures), canonicalization in this step MUST use a canonicalization algorithm designated as compatible with XML Signature 2.0. This canonicalization algorithm SHOULD be the same as that used for Reference canonicalization.
3. Construct the `Signature` element that includes `SignedInfo`, `Object`(s) (if desired, encoding may be different than that used for signing), `KeyInfo` (if required), and `SignatureValue`.

   Note, if the `Signature` includes same-document references, [XML10] or [XMLSCHEMA-1] ,[XMLSCHEMA-2] validation of the document might introduce changes that break the signature. Consequently, applications should be careful to consistently process the document or refrain from using external contributions (e.g., defaults and entities).

4.2 Reference Generation

For each Reference:

1. Decide how to represent the data object as a `dsig2:Selection`.
2. Use `Canonicalization` to convert the data object into an octet stream. This is not required for binary data.
3. Calculate the digest value over the resulting data object.
4. Create a `Reference` element, including the `dsig2:Selection` element, `Canonicalization` element, the digest algorithm and the `DigestValue`. (Note, it is the canonical form of these references that are signed in section 4.1 Signature Generation and validated in section Not found 'sec-ReferenceCheck-2.0'.)

XML data objects MUST be canonicalized using Canonical XML 2.0 [XML-C14N20] or an alternative algorithm that is compliant with its interface.

4.3 Core Validation

The REQUIRED steps of core validation include

1. establishing trust in the signing key mentioned in the `KeyInfo`. (Note in some environments, the signing key is implicitly known, and `KeyInfo` is not used at all).
2. Checking each `Reference` to to see if the data object matches with the expected data object.
3. the cryptographic signature validation of the signature calculated over `SignedInfo`.
4. reference validation, the verification of the digest contained in each `Reference` in `SignedInfo`.

These steps are present in ascending order of complexity, which ensures that the verifier rejects invalid signatures as quickly as possible.

Note, there may be valid signatures that some signature applications are unable to validate. Reasons for this include failure to implement optional parts of this specification, inability or unwillingness to execute specified algorithms, or inability or unwillingness to dereference specified URIs (some URI schemes may cause undesirable side effects), etc.

Comparison of each value in reference and signature validation is over the numeric (e.g., integer) or decoded octet sequence of the value. Different implementations may produce different encoded digest and signature values when processing the same resources because of variances in their encoding, such as accidental white space. But if one uses numeric or octet comparison (choose one) on both the stated and computed values these problems are eliminated.

4.4 Reference Check

The absence of arbitrary transforms makes reference checking simpler in XML Signature 2.0. Implementations process the `dsig2:Selection` in each `Reference` to return a list of data objects that are included in the signature. For example each reference in a signature may point to a different part of the same document. The signature implementation should return all these parts (possibly as DOM elements) to the calling application, which can then compare them against its policy to make sure what was expected to be signed is actually signed.

4.5 Reference Validation

Reference Validation is very similar to that in XML Signature 1.x, except that `SignedInfo` need not be canonicalized, there are no arbitrary transforms to execute, and there is an optional `dsig2:Verifications` step.

For each `Reference` in `SignedInfo`:

1. Obtain the data object to be digested using the `dsig2:Selection`.
    1. *Optional*: If the selection relies on an ID-based reference, and there is a `dsig2:Verification` element with `Type="http://www.w3.org/2010/xmldsig2#IDAttributes"`, then its content may assist in obtaining the intended data object by identifying an ID attribute that the verifier may not otherwise recognize.
    2. *Optional*: If the selection relies on an ID-based reference, and there is a `dsig2:Verification` element with `Type="http://www.w3.org/2010/xmldsig2#PositionAssertion"`, then the verifier may confirm that the data object obtained is the same as that which would be obtained by resolving the XPath expression in the `PositionAssertion` attribute.
2. Perform the `Canonicalization` to compute an octet stream.
    1. *Optional*: If there is a `dsig2:Verification` element with `Type="http://www.w3.org/2010/xmldsig2#DigestDataLength"`, then verify that the length of the octet stream computed above is the same as the length specified in the `DigestDataLength` attribute.
3. Digest the resulting data object using the `DigestMethod` specified in its `Reference` specification. The canonicalization and digesting can be combined in one step for efficiency.
4. Compare the generated digest value against `DigestValue` in the `SignedInfo Reference`; if there is any mismatch, validation fails.

4.6 Signature Validation

Signature Validation in XML Signature 2.0 is very similar to XML Signature 1.x, except that `KeyInfo` cannot contain any transforms, and the canonicalization of `SignatureMethod` is not required. These are the steps.

1. Obtain the keying information from `KeyInfo` or from an external source.
2. Using the `CanonicalizationMethod`(which must be Canonical XML 2.0 or an alternative algorithm that is compliant with its interface) and use the result (and previously obtained `KeyInfo`) to confirm the `SignatureValue` over the `SignedInfo` element.

5. Core Signature Syntax

The general structure of an XML Signature is described in section 2. Signature Overview and Examples. This section provides detailed syntax of the core signature features. Features described in this section are mandatory to implement unless otherwise indicated. The syntax is defined via an XML Schema [XMLSCHEMA-1][XMLSCHEMA-2] with the following XML preamble, declaration, and internal entity.

```
Schema Definition:

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

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        targetNamespace="http://www.w3.org/2000/09/xmldsig#"
        version="0.1" elementFormDefault="qualified">
```

Additional markup defined in version 1.1 of this specification uses the `dsig11:` namespace. The syntax is defined in an XML schema with the following preamble:

Schema Definition:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE schema PUBLIC "-//W3C//DTD XMLSchema 200102//EN"
[
<!ENTITY dsig 'http://www.w3.org/2000/09/xmldsig#'>
<!ENTITY dsig11 'http://www.w3.org/2009/xmldsig11#'>
<!ENTITY % p ''>
<!ENTITY % s ''>
]>

<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
        xmlns:dsig11="http://www.w3.org/2009/xmldsig11#"
        targetNamespace="http://www.w3.org/2009/xmldsig11#"
        version="0.1" elementFormDefault="qualified">
```

Finally, markup defined by version 2.0 of this specification uses the `dsig2:` namespace. The syntax is defined in an XML schema with the following preamble:

Notwithstanding the presence of a mixed content model (via mixed="true" declarations) in the definitions of various elements that follow, use of mixed content in conjunction with any elements defined by this specification is NOT RECOMMENDED.

When these elements are used in conjunction with XML Signature 2.0 signatures, mixed content MUST NOT be used.

5.1 The `ds:CryptoBinary` Simple Type

This specification defines the `ds:CryptoBinary` simple type for representing arbitrary-length integers (e.g. "bignums") in XML as octet strings. The integer value is first converted to a "big endian" bitstring. The bitstring is then padded with leading zero bits so that the total number of bits == 0 mod 8 (so that there are an integral number of octets). If the bitstring contains entire leading octets that are zero, these are removed (so the high-order octet is always non-zero). This octet string is then base64 [RFC2045] encoded. (The conversion from integer to octet string is equivalent to IEEE 1363's I2OSP [IEEE1363] with minimal length).

This type is used by "bignum" values such as `RSAKeyValue` and `DSAKeyValue`. If a value can be of type `base64Binary` or `ds:CryptoBinary` they are defined as `base64Binary`. For example, if the signature algorithm is RSA or DSA then `SignatureValue` represents a bignum and would be `ds:CryptoBinary`. However, if HMAC-SHA1 is the signature algorithm then `SignatureValue` could have leading zero octets that must be preserved. Thus `SignatureValue` is generically defined as of type `base64Binary`.

Schema Definition:

```
<simpleType name="CryptoBinary">
  <restriction base="base64Binary" />
</simpleType>
```

5.2 The `Signature` element

The `Signature` element is the root element of an XML Signature. Implementation MUST generate laxly schema valid [XMLSCHEMA-1][XMLSCHEMA-2] `Signature` elements as specified by the following schema:

Schema Definition:

```
<element name="Signature" type="ds:SignatureType"/>

<complexType name="SignatureType">
  <sequence>
    <element ref="ds:SignedInfo"/>
    <element ref="ds:SignatureValue"/>
    <element ref="ds:KeyInfo" minOccurs="0"/>
    <element ref="ds:Object" minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

5.3 The `SignatureValue` Element

The `SignatureValue` element contains the actual value of the digital signature; it is always encoded using base64 [RFC2045].

Schema Definition:

```
<element name="SignatureValue" type="ds:SignatureValueType" />

<complexType name="SignatureValueType">
  <simpleContent>
    <extension base="base64Binary">
      <attribute name="Id" type="ID" use="optional"/>
    </extension>
  </simpleContent>
</complexType>
```

5.4 The `SignedInfo` Element

The structure of `SignedInfo` includes a canonicalization algorithm, a signature algorithm, and one or more references. Given the importance of reference processing, this is described separately in section 6. Referencing Content.

The `SignedInfo` element may contain an optional ID attribute allowing it to be referenced by other signatures and objects.

`SignedInfo` does not include explicit signature or digest properties (such as calculation time, cryptographic device serial number, etc.). If an application needs to associate properties with the signature or digest, it may include such information in a `SignatureProperties` element within an `Object` element.

Schema Definition:

```
<element name="SignedInfo" type="ds:SignedInfoType"/>
```

```
<complexType name="SignedInfoType">
  <sequence>
    <element ref="ds:CanonicalizationMethod"/>
    <element ref="ds:SignatureMethod"/>
    <element ref="ds:Reference" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

5.4.1 The `CanonicalizationMethod` Element

`CanonicalizationMethod` is a required element that specifies the canonicalization algorithm applied to the `SignedInfo` element prior to performing signature calculations. This element uses the general structure for algorithms described in section 3.2.1 XML Signature 2.0 Algorithm Identifiers and Implementation Requirements. Implementations MUST support the REQUIRED canonicalization algorithms.

Schema Definition:

```
<element name="CanonicalizationMethod" type="ds:CanonicalizationMethodType"/>

<complexType name="CanonicalizationMethodType" mixed="true">
  <sequence>
    <any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
    <!-- (0,unbounded) elements from (1,1) namespace -->
  </sequence>
  <attribute name="Algorithm" type="anyURI" use="required"/>
</complexType>
```

In XML Signature 2.0, the `SignedInfo` element is presented as a single subtree with no exclusions to the Canonicalization 2.0 [XML-C14N20] algorithm. Parameters to that algorithm are represented as subelements of the `Canonicalization` element.

XML Signature 2.0 signatures use the `CanonicalizationMethod` element to express the canonicalization of each `Reference`.

5.4.2 The `SignatureMethod` Element

`SignatureMethod` is a required element that specifies the algorithm used for signature generation and validation. This algorithm identifies all cryptographic functions involved in the signature operation (e.g. hashing, public key algorithms, MACs, padding, etc.). This element uses the general structure here for algorithms described in section 3.2.1 XML Signature 2.0 Algorithm Identifiers and Implementation Requirements. While there is a single identifier, that identifier may specify a format containing multiple distinct signature values.

Schema Definition:

```
<element name="SignatureMethod" type="ds:SignatureMethodType"/>

<complexType name="SignatureMethodType" mixed="true">
  <sequence>
    <element name="HMACOutputLength" minOccurs="0"
             type="ds:HMACOutputLengthType"/>
    <any namespace="##other" minOccurs="0" maxOccurs="unbounded"/>
    <!-- (0,unbounded) elements from (1,1) external namespace -->
  </sequence>
  <attribute name="Algorithm" type="anyURI" use="required"/>
</complexType>
```

The `ds:HMACOutputLength` parameter is used for HMAC [HMAC] algorithms. The parameter specifies a truncation length in bits. If this parameter is trusted without further verification, then this can lead to a security bypass [CVE-2009-0217]. Signatures MUST be deemed invalid if the truncation length is below the larger of (a) half the underlying hash algorithm's output length, and (b) 80 bits. Note that some implementations are known to not accept truncation lengths that are lower than the underlying hash algorithm's output length.

5.4.3 The `DigestMethod` Element

`DigestMethod` is a required element that identifies the digest algorithm to be applied to the signed object. This element uses the general structure here for algorithms specified in section 3.2.1 XML Signature 2.0 Algorithm Identifiers and Implementation Requirements.

For "Compatibility Mode" signatures, if the result of the URI dereference and application of `Transforms` is an XPath node-set (or sufficiently functional replacement implemented by the application) then it must be converted as described in section B.4.1 The "Compatibility Mode" Reference Processing Model. If the result of URI dereference and application of `Transforms` is an octet stream, then no conversion occurs (comments might be present if Canonical XML with Comments was specified in the `Transforms`). The digest algorithm is applied to the data octets of the resulting octet stream.

For XML Signature 2.0 signatures, the result of processing the `Reference` is an octet stream, and the digest algorithm is applied to the resulting data octets.

Schema Definition:

```
<element name="DigestMethod" type="ds:DigestMethodType"/>

<complexType name="DigestMethodType" mixed="true">
  <sequence>
    <any namespace="##other" processContents="lax"
         minOccurs="0" maxOccurs="unbounded"/>
  </sequence>
  <attribute name="Algorithm" type="anyURI" use="required"/>
</complexType>
```

5.4.4 The `DigestValue` Element

`DigestValue` is an element that contains the encoded value of the digest. The digest is always encoded using base64 [RFC2045].

Schema Definition:

```
<element name="DigestValue" type="ds:DigestValueType"/>

<simpleType name="DigestValueType">
```

```
    <restriction base="base64Binary"/>
  </simpleType>
```

6. Referencing Content

The XML Signature 2.0 specification is designed to support a new, simplified processing model while remaining backwardly-compatible with the older 1.x processing model through optional support of a "Compatibility Mode" defined in a separate section of this document, section B. Compatibility Mode.

A generic signature processor can determine the mode of a signature by examining the `Reference` element's attributes and the child element(s) of the `Transforms` element (if any). If the `URI` attributes is present, "Compatibility Mode" can be assumed. If the `URI` attribute is not present, **and** the `Transforms` element contains exactly one `Transform` element with an `Algorithm` of `"http://www.w3.org/2010/xmldsig2#transform"`, then XML Signature 2.0 processing can be assumed. Otherwise, "Compatibility Mode" is applied.

All the references of a signature SHOULD have the same mode; i.e. all XML Signature 2.0, or all "Compatibility Mode".

6.1 The `Reference` Element

`Reference` is an element that may occur one or more times. It specifies a digest algorithm and digest value, and optionally an identifier of the object being signed, the type of the object, and/or a list of transforms to be applied prior to digesting. The identification (URI) and transforms describe how the digested content (i.e., the input to the digest method) was created. The `Type` attribute facilitates the processing of referenced data. For example, while this specification makes no requirements over external data, an application may wish to signal that the referent is a `Manifest`. An optional ID attribute permits a `Reference` to be referenced from elsewhere.

```
Schema Definition:

<element name="Reference" type="ds:ReferenceType"/>

<complexType name="ReferenceType">
  <sequence>
    <element ref="ds:Transforms" minOccurs="0"/>
    <element ref="ds:DigestMethod"/>
    <element ref="ds:DigestValue"/>
  </sequence>
  <attribute name="Id" type="ID" use="optional"/>
  <attribute name="URI" type="anyURI" use="optional"/>
  <attribute name="Type" type="anyURI" use="optional"/>
</complexType>
```

6.1.1 The `URI` Attribute

The URI attribute MUST be omitted for XML Signature 2.0 signatures.

6.2 The `Transforms` Element

Each `Reference` MUST contain the `Transforms` element, and this MUST contain one and only one `Transform` element with an `Algorithm` of `"http://www.w3.org/2010/xmldsig2#transform"`. This signals the 2.0 syntax and processing (Compatibility mode transforms are described in section B.5 "Compatibility Mode" Transforms and Processing Model).

```
Schema Definition:

<element name="Transforms" type="ds:TransformsType"/>

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

<element name="Transform" type="ds:TransformType"/>

<complexType name="TransformType" mixed="true">
  <choice minOccurs="0" maxOccurs="unbounded">
    <any namespace="##other" processContents="lax"/>
    <!-- (1,1) elements from (0,unbounded) namespaces -->
    <element name="XPath" type="string"/>
  </choice>
  <attribute name="Algorithm" type="anyURI" use="required"/>
</complexType>
```

The semantics of the `Transform` element in XML Signature 2.0 is that its input is determined solely from within the `Transform` itself rather than via the surrounding `Reference`. The output is guaranteed to be an octet stream.

The detailed definition of the XML Signature 2.0 `Transform` algorithm definitions can be found in section Not found 'sec-Transforms-2.0'.

A difference from XML Signature 1.x (and the corresponding "Compatibility Mode") is that the use of extensible `Transform` algorithms is replaced with an extensible syntax for reference and selection processing. This construct is modeled as a fixed Transform, for compatibility with the original schema, and to ensure predictable failure modes for older implementations.

Legacy implementations should react to this as an undefined `Transform` and report failure in the fashion that is normal for them in such a case.

6.3 The `dsig2:Selection` Element

The `dsig2:Selection` element describes the data being signed for a "2.0 Mode" signature `Reference`. The content and processing model for this element depends on the value of the required `Algorithm` attribute, which identifies the selection algorithm/syntax in use. The required `URI` attribute and any child elements are passed to that algorithm as parameters to selection processing.

The `Algorithm` attribute is an extensibility point enabling application-specific content selection approaches. Each `Algorithm` must define the parameters expected, how they are expressed within the `dsig2:Selection` element, how to process the selection, what user-defined object the selection produces,

and what canonicalization algorithm(s) to allow for unambiguous conversation of the data into an octet stream.

The result of processing the `dsig2:Selection` element MUST be one of the following:

- one or more subtrees with optional exclusions (see Subtrees with Exclusions)
- an octet stream
- any user-defined object

In the first case, the current `Signature` node is implicitly added as an exclusion, and then a "2.0 Mode" canonicalization algorithm (one compatible with these inputs) MUST be applied to produce an octet stream for the digest algorithm. The contents of the sibling `CanonicalizationMethod` element, if present, will specify the algorithm to use, and supply any non-default parameters to that algorithm. If no sibling `CanonicalizationMethod` element is present, then the XML Canonicalization 2.0 Algorithm [XML-C14N20] MUST be applied with no non-default parameters.

For an octet stream, no further processing is applied, and the octets are supplied directly to the digest algorithm.

For a user-defined object (the result of a user-defined selection process), processing is subject to the definition of that process.

6.3.1 Subtrees with Optional Exclusions

Signatures in "2.0 Mode" do not deal with XML content to be signed in terms of an XPath nodeset. Instead, the following interface is used:

- An XML fragment to be signed is represented as one or more "inclusion" subtrees, and a set of zero or more "exclusions" consisting of subtrees and/or attribute nodes.
- Exclusions override inclusions; i.e., the selection contains all the nodes in the inclusion subtrees minus all the nodes in the exclusion subtrees.
- A "subtree" is the portion of an XML document consisting of all the descendants of a particular element node (inclusive), or the document root node. The subtree is identified by the element node/document root node.
- If, in the inclusion list, one subtree is included in another, the included one is effectively ignored (the two are simply unioned).
- Each subtree (except when the subtree is of a complete document) must be accompanied by the set of namespace declarations in scope (i.e., inherited from the ancestors of the subtree).

6.4 The `dsig2:Verifications` Element

`dsig2:Verifications` is an optional element containing information that aids in signature verification. It contains one or more `dsig2:Verification` elements identifying the type(s) of verification information available.

Use of the `dsig2:Verifications` element by validators is optional, even if the element is present. For example, validators may ignore a `dsig2:Verification` element of `Type "http://www.w3.org/2010/xmldsig2#PositionAssertion"`, and rely on ID-based referencing (with the risk of being vulnerable to signature wrapping attacks unless other steps are taken) for simplicity.

7. The `KeyInfo` Element

`KeyInfo` is an optional element that enables the recipient(s) to obtain the key needed to validate the signature. `KeyInfo` may contain keys, names, certificates and other public key management information, such as in-band key distribution or key agreement data. This specification defines a few simple types but applications may extend those types or all together replace them with their own key identification and exchange semantics using the XML namespace facility [XML-NAMES]. However, questions of trust of such key information (e.g., its authenticity or strength) are out of scope of this specification and left to the application. Details of the structure and usage of element children of `KeyInfo` other than simple types described in this specification are out of scope. For example, the definition of PKI certificate contents, certificate ordering, certificate revocation and CRL management are out of scope.

If `KeyInfo` is omitted, the recipient is expected to be able to identify the key based on application context. Multiple declarations within `KeyInfo` refer to the same key. While applications may define and use any mechanism they choose through inclusion of elements from a different namespace, compliant versions MUST implement `KeyValue` (section 7.2 The KeyValue Element) and SHOULD implement `KeyInfoReference` (section 7.10 The dsig11:KeyInfoReference Element). `KeyInfoReference` is preferred over use of `RetrievalMethod` as it avoids use of `Transform` child elements that introduce security risk and implementation challenges. Support for other children of `KeyInfo` is OPTIONAL.

The schema specification of many of `KeyInfo`'s children (e.g., `PGPData`, `SPKIData`, `X509Data`) permit their content to be extended/complemented with elements from another namespace. This may be done only if it is safe to ignore these extension elements while claiming support for the types defined in this specification. Otherwise, external elements, including *alternative* structures to those defined by this specification, MUST be a child of `KeyInfo`. For example, should a complete XML-PGP standard be defined, its root element MUST be a child of `KeyInfo`. (Of course, new structures from external namespaces can incorporate elements from the `dsig:` namespace via features of the type definition language. For instance, they can create a schema that permits, includes, imports, or derives new types based on `dsig:` elements.)

The following list summarizes the `KeyInfo` types that are allocated an identifier in the `dsig:` namespace; these can be used within the `RetrievalMethod Type` attribute to describe a remote `KeyInfo` structure.

- http://www.w3.org/2000/09/xmldsig#DSAKeyValue
- http://www.w3.org/2000/09/xmldsig#RSAKeyValue
- http://www.w3.org/2000/09/xmldsig#X509Data
- http://www.w3.org/2000/09/xmldsig#PGPData
- http://www.w3.org/2000/09/xmldsig#SPKIData
- http://www.w3.org/2000/09/xmldsig#MgmtData

The following list summarizes the additional `KeyInfo` types that are allocated an identifier in the `dsig11:` namespace.

- http://www.w3.org/2009/xmldsig11#ECKeyValue
- http://www.w3.org/2009/xmldsig11#DEREncodedKeyValue

In addition to the types above for which we define an XML structure, we specify one additional type to indicate a binary (ASN.1 DER) X.509 Certificate.

- http://www.w3.org/2000/09/xmldsig#rawX509Certificate

```
Schema Definition:

<element name="KeyInfo" type="ds:KeyInfoType"/>

<complexType name="KeyInfoType" mixed="true">
```

```
    <choice maxOccurs="unbounded">
      <element ref="ds:KeyName"/>
      <element ref="ds:KeyValue"/>
      <element ref="ds:RetrievalMethod"/>
      <element ref="ds:X509Data"/>
      <element ref="ds:PGPData"/>
      <element ref="ds:SPKIData"/>
      <element ref="ds:MgmtData"/>
      <!-- <element ref="dsig11:DEREncodedKeyValue"/> -->
      <!-- DEREncodedKeyValue (XMLDsig 1.1) will use the any element -->
      <!-- <element ref="dsig11:KeyInfoReference"/> -->
      <!-- KeyInfoReference (XMLDsig 1.1) will use the any element -->
      <!-- <element ref="xenc:EncryptedKey"/> -->
      <!-- EncryptedKey (XMLEnc) will use the any element -->
      <!-- <element ref="xenc:Agreement"/> -->
      <!-- Agreement (XMLEnc) will use the any element -->
      <any processContents="lax" namespace="##other"/>
      <!-- (1,1) elements from (0,unbounded) namespaces -->
    </choice>
    <attribute name="Id" type="ID" use="optional"/>
  </complexType>
```

7.1 The KeyName Element

The KeyName element contains a string value (in which white space is significant) which may be used by the signer to communicate a key identifier to the recipient. Typically, KeyName contains an identifier related to the key pair used to sign the message, but it may contain other protocol-related information that indirectly identifies a key pair. (Common uses of KeyName include simple string names for keys, a key index, a distinguished name (DN), an email address, etc.)

```
Schema Definition:

<element name="KeyName" type="string" />
```

7.2 The KeyValue Element

The KeyValue element contains a single public key that may be useful in validating the signature. Structured formats for defining DSA (REQUIRED), RSA (REQUIRED) and ECDSA (REQUIRED) public keys are defined in section 10.3 Signature Algorithms. The KeyValue element may include externally defined public keys values represented as PCDATA or element types from an external namespace.

```
Schema Definition:

<element name="KeyValue" type="ds:KeyValueType" />

<complexType name="KeyValueType" mixed="true">
  <choice>
    <element ref="ds:DSAKeyValue"/>
    <element ref="ds:RSAKeyValue"/>
    <!-- <element ref="dsig11:ECKeyValue"/> -->
    <!-- ECC keys (XMLDsig 1.1) will use the any element -->
    <any namespace="##other" processContents="lax"/>
  </choice>
</complexType>
```

7.2.1 The DSAKeyValue Element

Identifier

Type="http://www.w3.org/2000/09/xmldsig#DSAKeyValue" (this can be used within a RetrievalMethod or Reference element to identify the referent's type)

DSA keys and the DSA signature algorithm are specified in [FIPS-186-3]. DSA public key values can have the following fields:

P
: a prime modulus meeting the [FIPS-186-3] requirements

Q
: an integer in the range 2**159 < Q < 2**160 which is a prime divisor of P-1

G
: an integer with certain properties with respect to P and Q

Y
: G**X mod P (where X is part of the private key and not made public)

J
: (P - 1) / Q

seed
: a DSA prime generation seed

pgenCounter
: a DSA prime generation counter

Parameter J is available for inclusion solely for efficiency as it can be calculated from P and Q. Parameters seed and pgenCounter are used in the DSA prime number generation algorithm specified in [FIPS-186-3]. As such, they are optional but must either both be present or both be absent. This prime generation algorithm is designed to provide assurance that a weak prime is not being used and it yields a P and Q value. Parameters P, Q, and G can be public and common to a group of users. They might be known from application context. As such, they are optional but P and Q must either both appear or both be absent. If all of P, Q, seed, and pgenCounter are present, implementations are not required to check if they are consistent and are free to use either P and Q or seed and pgenCounter. All parameters are encoded as base64 [RFC2045] values.

Arbitrary-length integers (e.g. "bignums" such as RSA moduli) are represented in XML as octet strings as defined by the ds:CryptoBinary type.

```
Schema Definition:

<element name="DSAKeyValue" type="ds:DSAKeyValueType" />

<complexType name="DSAKeyValueType">
  <sequence>
    <sequence minOccurs="0">
```

```
      <element name="P" type="ds:CryptoBinary"/>
      <element name="Q" type="ds:CryptoBinary"/>
    </sequence>
    <element name="G" type="ds:CryptoBinary" minOccurs="0"/>
    <element name="Y" type="ds:CryptoBinary"/>
    <element name="J" type="ds:CryptoBinary" minOccurs="0"/>
    <sequence minOccurs="0">
      <element name="Seed" type="ds:CryptoBinary"/>
      <element name="PgenCounter" type="ds:CryptoBinary"/>
    </sequence>
  </sequence>
</complexType>
```

7.2.2 The `RSAKeyValue` Element

Identifier

Type="http://www.w3.org/2000/09/xmldsig#RSAKeyValue" (this can be used within a `RetrievalMethod` or `Reference` element to identify the referent's type)

RSA key values have two fields: Modulus and Exponent.

Arbitrary-length integers (e.g. "bignums" such as RSA moduli) are represented in XML as octet strings as defined by the `ds:CryptoBinary` type.

EXAMPLE 5

```
<RSAKeyValue>
  <Modulus>xA7SEU+e0yQH5rm9kbCDN9o3aPIo7HbP7tX6WOocLZAtNfyxSZDU16ksL6W
  jubafOqNEpcwR3RdFsT7bCqnXPBe5ELh5u4VEy19MzxkXRgrMvavzyBpVRgBUwUlV
  5foK5hhmbktQhyNdy/6LpQRhDUDsTvK+g9Ucj47es9AQJ3U=
  </Modulus>
  <Exponent>AQAB</Exponent>
</RSAKeyValue>
```

7.2.3 The `dsig11:ECKeyValue` Element

Identifier

Type="http://www.w3.org/2009/xmldsig11#ECKeyValue"
(this can be used within a `RetrievalMethod` or `Reference` element to identify the referent's type)

The `dsig11:ECKeyValue` element is defined in the http://www.w3.org/2009/xmldsig11# namespace.

EC public key values consists of two sub components: Domain parameters and `dsig11:PublicKey`.

EXAMPLE 6

```
<ECKeyValue xmlns="http://www.w3.org/2009/xmldsig11#">
  <NamedCurve URI="urn:oid:1.2.840.10045.3.1.7" />
  <PublicKey>
    vWccUP6Jp3pcaMCGIcAh3YOev4gaa2ukOANC7Ufg
    Cf8KDO7AtTOsGJK7/TA8IC3vZoCy9I5oPjRhyTBulBnj7Y
  </PublicKey>
</ECKeyValue>
```

Note - A line break has been added to the `dsig11:PublicKey` content to preserve printed page width.

Domain parameters can be encoded explicitly using the `dsig11:ECParameters` element or by reference using the `dsig11:NamedCurve` element. A named curve is specified through the `URI` attribute. For named curves that are identified by OIDs, such as those defined in [RFC3279] and [RFC4055], the OID SHOULD be encoded according to [URN-OID]. Conformant applications MUST support the `dsig11:NamedCurve` element and the 256-bit prime field curve as identified by the OID `1.2.840.10045.3.1.7`.

The `dsig11:PublicKey` element contains the base64 encoding of a binary representation of the x and y coordinates of the point. Its value is computed as follows:

1. Convert the elliptic curve point (x,y) to an octet string by first converting the field elements x and y to octet strings as specified in Section 6.2 of [ECC-ALGS] (note), and then prepend the concatenated result of the conversion with 0x04. Support for Elliptic-Curve-Point-to-Octet-String conversion without point compression is REQUIRED.
2. Base64 encode the octet string resulting from the conversion in Step 1.

Schema Definition:

```
<!-- targetNamespace="http://www.w3.org/2009/xmldsig11#" -->

<element name="ECKeyValue" type="dsig11:ECKeyValueType" />

<complexType name="ECKeyValueType">
  <sequence>
    <choice>
      <element name="ECParameters" type="dsig11:ECParametersType" />
      <element name="NamedCurve" type="dsig11:NamedCurveType" />
    </choice>
    <element name="PublicKey" type="dsig11:ECPointType" />
  </sequence>
  <attribute name="Id" type="ID" use="optional" />
</complexType>

<complexType name="NamedCurveType">
  <attribute name="URI" type="anyURI" use="required" />
</complexType>

<simpleType name="ECPointType">
  <restriction base="ds:CryptoBinary" />
</simpleType>
```

7.2.3.1 Explicit Curve Parameters

The dsig11:ECParameters element consists of the following subelements. Note these definitions are based on the those described in [RFC3279].

1. The dsig11:FieldID element identifies the finite field over which the elliptic curve is defined. Additional details on the structures for defining prime and characteristic two fields is provided below.
2. The dsig11:Curve element specifies the coefficients a and b of the elliptic curve E. Each coefficient is first converted from a field element to an octet string as specified in section 6.2 of [ECC-ALGS], then the resultant octet string is encoded in base64.
3. The dsig11:Base element specifies the base point P on the elliptic curve. The base point is represented as a value of type dsig11:ECPointType.
4. The dsig11:Order element specifies the order n of the base point and is encoded as a positiveInteger.
5. The dsig11:Cofactor element is an optional element that specifies the integer h = #E(Fq)/n. The cofactor is not required to support ECDSA, except in parameter validation. The cofactor MAY be included to support parameter validation for ECDSA keys. Parameter validation is not required by this specification. The cofactor is required in ECDH public key parameters.
6. The dsig11:ValidationData element is an optional element that specifies the hash algorithm used to generate the elliptic curve E and the base point G verifiably at random. It also specifies the seed that was used to generate the curve and the base point.

Schema Definition:

```
<!-- targetNamespace="http://www.w3.org/2009/xmldsig11#" -->

<complexType name="ECParametersType">
  <sequence>
    <element name="FieldID" type="dsig11:FieldIDType" />
    <element name="Curve" type="dsig11:CurveType" />
    <element name="Base" type="dsig11:ECPointType" />
    <element name="Order" type="ds:CryptoBinary" />
    <element name="CoFactor" type="integer" minOccurs="0" />
    <element name="ValidationData"
             type="dsig11:ECValidationDataType" minOccurs="0" />
  </sequence>
</complexType>

<complexType name="FieldIDType">
  <choice>
    <element ref="dsig11:Prime" />
    <element ref="dsig11:TnB" />
    <element ref="dsig11:PnB" />
    <element ref="dsig11:GnB" />
    <any namespace="##other" processContents="lax" />
  </choice>
</complexType>

<complexType name="CurveType">
  <sequence>
    <element name="A" type="ds:CryptoBinary" />
    <element name="B" type="ds:CryptoBinary" />
  </sequence>
</complexType>

<complexType name="ECValidationDataType">
  <sequence>
    <element name="seed" type="ds:CryptoBinary" />
  </sequence>
  <attribute name="hashAlgorithm" type="anyURI" use="required" />
</complexType>
```

dsig11:Prime fields are described by a single subelement dsig11:P, which represents the field size in bits. It is encoded as a positiveInteger.

Schema Definition:

```
<!-- targetNamespace="http://www.w3.org/2009/xmldsig11#" -->

<element name="Prime" type="dsig11:PrimeFieldParamsType" />

<complexType name="PrimeFieldParamsType">
  <sequence>
    <element name="P" type="ds:CryptoBinary" />
  </sequence>
</complexType>
```

Structures are defined for three types of characteristic two fields: gaussian normal basis, pentanomial basis and trinomial basis.

Schema Definition:

```
<!-- targetNamespace="http://www.w3.org/2009/xmldsig11#" -->

<element name="GnB" type="dsig11:CharTwoFieldParamsType" />

<complexType name="CharTwoFieldParamsType">
  <sequence>
    <element name="M" type="positiveInteger" />
  </sequence>
</complexType>

<element name="TnB" type="dsig11:TnBFieldParamsType" />

<complexType name="TnBFieldParamsType">
  <complexContent>
    <extension base="dsig11:CharTwoFieldParamsType">
      <sequence>
        <element name="K" type="positiveInteger" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

```
<element name="PnB" type="dsig11:PnBFieldParamsType" />

<complexType name="PnBFieldParamsType">
  <complexContent>
    <extension base="dsig11:CharTwoFieldParamsType">
      <sequence>
        <element name="K1" type="positiveInteger" />
        <element name="K2" type="positiveInteger" />
        <element name="K3" type="positiveInteger" />
      </sequence>
    </extension>
  </complexContent>
</complexType>
```

7.2.3.2 Compatibility with RFC 4050

Implementations that need to support the [RFC4050] format for ECDSA keys can avoid known interoperability problems with that specification by adhering to the following profile:

1. Avoid validating the `ECDSAKeyValue` element against the [RFC4050] schema. XML Schema validators may not support integer types with decimal data exceeding 18 decimal digits. [XMLSCHEMA-1][XMLSCHEMA-2].
2. Support only the `NamedCurve` element.
3. Support the 256-bit prime field curve, as identified by the URN `urn:oid:1.2.840.10045.3.1.7`.

The following is an example of a `ECDSAKeyValue` element that meets the profile described in this section.

EXAMPLE 7

```
<ECDSAKeyValue xmlns="http://www.w3.org/2001/04/xmldsig-more#">
  <DomainParameters>
    <NamedCurve URN="urn:oid:1.2.840.10045.3.1.7" />
  </DomainParameters>
  <PublicKey>
    <X Value="585110606538017443932491790464828333
              020493188426732615513405625862406434988 5" />
    <Y Value="102403352136827775240910267217779508 3
              590286425248815408780791198957641614349 36" />
  </PublicKey>
</ECDSAKeyValue>
```

Note - A line break has been added to the `X` and `Y Value` attribute values to preserve printed page width.

7.3 The `RetrievalMethod` Element

A `RetrievalMethod` element within `KeyInfo` is used to convey a reference to `KeyInfo` information that is stored at another location. For example, several signatures in a document might use a key verified by an X.509v3 certificate chain appearing once in the document or remotely outside the document; each signature's `KeyInfo` can reference this chain using a single `RetrievalMethod` element instead of including the entire chain with a sequence of `X509Certificate` elements.

`RetrievalMethod` uses the same syntax and dereferencing behavior as section B.4 The URI Attribute in "Compatibility Mode" and section B.4.1 The "Compatibility Mode" Reference Processing Model except that there are no `DigestMethod` or `DigestValue` child elements and presence of the `URI` attribute is mandatory.

`Type` is an optional identifier for the type of data retrieved after all transforms have been applied. The result of dereferencing a `RetrievalMethod` `Reference` for all `KeyInfo` types defined by this specification (section 7. The KeyInfo Element) with a corresponding XML structure is an XML element or document with that element as the root. The `rawX509Certificate` `KeyInfo` (for which there is no XML structure) returns a binary X509 certificate.

Note that when referencing one of the defined `KeyInfo` types within the same document, or some remote documents, at least one `Transform` is required to turn an ID-based reference to a `KeyInfo` element into a child element located inside it. This is due to the lack of an XML ID attribute on the defined `KeyInfo` types.

Transforms in `RetrievalMethod` are more attack prone, since they need to be evaluated in the first step of the signature validation, where the trust in the key has not yet been established, and the `SignedInfo` has not yet been verified. As noted in the [XMLDSIG-BESTPRACTICES] an attacker can easily causes a Denial of service, by adding a specially crafted transform in the `RetrievalMethod` without even bothering to have the key validate or the signature match.

Note: The `KeyInfoReference` element is preferred over use of `RetrievalMethod` as it avoids use of `Transform` child elements that introduce security risk and implementation challenges.

```
Schema Definition:

<element name="RetrievalMethod" type="ds:RetrievalMethodType" />

<complexType name="RetrievalMethodType">
  <sequence>
    <element ref="ds:Transforms" minOccurs="0" />
  </sequence>
  <attribute name="URI" type="anyURI" />
  <attribute name="Type" type="anyURI" use="optional" />
</complexType>
```

Note: The schema for the `URI` attribute of RetrievalMethod erroneously omitted the attribute: `use="required"`. However, this error only results in a more lax schema which permits all valid `RetrievalMethod` elements. Because the existing schema is embedded in many applications, which may include the schema in their signatures, the schema has not been corrected to be more restrictive.

7.4 The `X509Data` Element

Identifier

Type="http://www.w3.org/2000/09/xmldsig#X509Data "
(this can be used within a RetrievalMethod or Reference element to identify the referent's type)

An X509Data element within KeyInfo contains one or more identifiers of keys or X509 certificates (or certificates' identifiers or a revocation list). The content of X509Data is at least one element, from the following set of element types; any of these may appear together or more than once iff (if and only if) each instance describes or is related to the same certificate:

- The deprecated X509IssuerSerial element, which contains an X.509 issuer distinguished name/serial number pair. The distinguished name SHOULD be represented as a string that complies with section 3 of RFC4514 [LDAP-DN], to be generated according to the Distinguished Name Encoding Rules section below,
- The X509SubjectName element, which contains an X.509 subject distinguished name that SHOULD be represented as a string that complies with section 3 of RFC4514 [LDAP-DN], to be generated according to the Distinguished Name Encoding Rules section below,
- The X509SKI element, which contains the base64 encoded plain (i.e. non-DER-encoded) value of a X509 V.3 SubjectKeyIdentifier extension,
- The X509Certificate element, which contains a base64-encoded [X509V3] certificate, and
- The X509CRL element, which contains a base64-encoded certificate revocation list (CRL) [X509V3].
- The dsig11:X509Digest element contains a base64-encoded digest of a certificate. The digest algorithm URI is identified with a required Algorithm attribute. The input to the digest MUST be the raw octets that would be base64-encoded were the same certificate to appear in the X509Certificate element.
- Elements from an external namespace which accompanies/complements any of the elements above.

Any X509IssuerSerial, X509SKI, X509SubjectName, and dsig11:X509Digest elements that appear MUST refer to the certificate or certificates containing the validation key. All such elements that refer to a particular individual certificate MUST be grouped inside a single X509Data element and if the certificate to which they refer appears, it MUST also be in that X509Data element.

Any X509IssuerSerial, X509SKI, X509SubjectName, and dsig11:X509Digest elements that relate to the same key but different certificates MUST be grouped within a single KeyInfo but MAY occur in multiple X509Data elements.

Note that if X509Data child elements are used to identify a trusted certificate (rather than solely as an untrusted hint supplemented by validation by policy), the complete set of such elements that are intended to identify a certificate SHOULD be integrity protected, typically by signing an entire X509Data or KeyInfo element.

All certificates appearing in an X509Data element MUST relate to the validation key by either containing it or being part of a certification chain that terminates in a certificate containing the validation key.

No ordering is implied by the above constraints. The comments in the following instance demonstrate these constraints:

EXAMPLE 8

```
<KeyInfo>
  <X509Data> <!-- two pointers to certificate-A -->
    <X509IssuerSerial>
      <X509IssuerName>
        CN=TAMURA Kent, OU=TRL, O=IBM, L=Yamato-shi, ST=Kanagawa, C=JP
      </X509IssuerName>
      <X509SerialNumber>12345678</X509SerialNumber>
    </X509IssuerSerial>
    <X509SKI>31d97bd7</X509SKI>
  </X509Data>
  <X509Data><!-- single pointer to certificate-B -->
    <X509SubjectName>Subject of Certificate B</X509SubjectName>
  </X509Data>
  <X509Data> <!-- certificate chain -->
    <!--Signer cert, issuer CN=arbolCA,OU=FVT,O=IBM,C=US, serial 4-->
    <X509Certificate>MIICXTCCA..</X509Certificate>
    <!-- Intermediate cert subject CN=arbolCA,OU=FVT,O=IBM,C=US
         issuer CN=tootiseCA,OU=FVT,O=Bridgepoint,C=US -->
    <X509Certificate>MIICPzCCA...</X509Certificate>
    <!-- Root cert subject CN=tootiseCA,OU=FVT,O=Bridgepoint,C=US -->
    <X509Certificate>MIICSTCCA...</X509Certificate>
  </X509Data>
</KeyInfo>
```

Note, there is no direct provision for a PKCS#7 encoded "bag" of certificates or CRLs. However, a set of certificates and CRLs can occur within an X509Data element and multiple X509Data elements can occur in a KeyInfo. Whenever multiple certificates occur in an X509Data element, at least one such certificate must contain the public key which verifies the signature.

While in principle many certificate encodings are possible, it is RECOMMENDED that certificates appearing in an X509Certificate element be limited to an encoding of BER or its DER subset, allowing that within the certificate other content may be present. The use of other encodings may lead to interoperability issues. In any case, XML Signature implementations SHOULD NOT alter or re-encode certificates, as doing so could invalidate their signatures.

Deployments that expect to make use of the X509IssuerSerial element should be aware that many Certificate Authorities issue certificates with large, random serial numbers. XML Schema validators may not support integer types with decimal data exceeding 18 decimal digits [XML-schema]. Therefore such deployments should avoid schema-validating the X509IssuerSerial element, or make use of a local copy of the schema that adjusts the data type of the X509SerialNumber child element from "integer" to "string".

7.4.1 Distinguished Name Encoding Rules

To encode a distinguished name (X509IssuerSerial,X509SubjectName, and KeyName if appropriate), the encoding rules in section 2 of RFC 4514 [LDAP-DN] SHOULD be applied, except that the character escaping rules in section 2.4 of RFC 4514 [LDAP-DN] MAY be augmented as follows:

- Escape all occurrences of ASCII control characters (Unicode range \x00 - \x1f) by replacing them with "\" followed by a two digit hex number showing its Unicode number.
- Escape any trailing space characters (Unicode \x20) by replacing them with "\20", instead of using the escape sequence "\ ".

Since an XML document logically consists of characters, not octets, the resulting Unicode string is finally encoded according to the character encoding used for producing the physical representation of the XML document.

```
Schema Definition:

<element name="X509Data" type="ds:X509DataType"/>

<complexType name="X509DataType">
  <sequence maxOccurs="unbounded">
    <choice>
      <element name="X509IssuerSerial" type="ds:X509IssuerSerialType"/>
      <element name="X509SKI" type="base64Binary"/>
      <element name="X509SubjectName" type="string"/>
      <element name="X509Certificate" type="base64Binary"/>
      <element name="X509CRL" type="base64Binary"/>
      <!-- <element ref="dsig11:X509Digest"/> -->
      <!-- The X509Digest element (XMLDSig 1.1) will use the any element -->
      <any namespace="##other" processContents="lax"/>
    </choice>
  </sequence>
</complexType>

<complexType name="X509IssuerSerialType">
  <sequence>
    <element name="X509IssuerName" type="string"/>
    <element name="X509SerialNumber" type="integer"/>
  </sequence>
</complexType>

<!-- Note, this schema permits X509Data to be empty; this is
     precluded by the text in
     <a href="#sec-KeyInfo" class="sectionRef"></a> which states
     that at least one element from the dsig namespace should be present
     in the PGP, SPKI, and X509 structures. This is easily expressed for
     the other key types, but not for X509Data because of its rich
     structure. -->

<!-- targetNameSpace="http://www.w3.org/2009/xmldsig11#" -->

<element name="X509Digest" type="dsig11:X509DigestType"/>

<complexType name="X509DigestType">
  <simpleContent>
    <extension base="base64Binary">
      <attribute name="Algorithm" type="anyURI" use="required"/>
    </extension>
  </simpleContent>
</complexType>
```

7.5 The PGPData Element

Identifier

Type="http://www.w3.org/2000/09/xmldsig#PGPData " (this can be used within a RetrievalMethod or Reference element to identify the referent's type)

The PGPData element within KeyInfo is used to convey information related to PGP public key pairs and signatures on such keys. The PGPKeyID's value is a base64Binary sequence containing a standard PGP public key identifier as defined in [PGP] section 11.2]. The PGPKeyPacket contains a base64-encoded Key Material Packet as defined in [PGP] section 5.5]. These children element types can be complemented/extended by siblings from an external namespace within PGPData, or PGPData can be replaced all together with an alternative PGP XML structure as a child of KeyInfo. PGPData must contain one PGPKeyID and/or one PGPKeyPacket and 0 or more elements from an external namespace.

```
Schema Definition:

<element name="PGPData" type="ds:PGPDataType"/>

<complexType name="PGPDataType">
  <choice>
    <sequence>
      <element name="PGPKeyID" type="base64Binary"/>
      <element name="PGPKeyPacket" type="base64Binary" minOccurs="0"/>
      <any namespace="##other" processContents="lax" minOccurs="0"
           maxOccurs="unbounded"/>
    </sequence>
    <sequence>
      <element name="PGPKeyPacket" type="base64Binary"/>
      <any namespace="##other" processContents="lax" minOccurs="0"
           maxOccurs="unbounded"/>
    </sequence>
  </choice>
</complexType>
```

7.6 The SPKIData Element

Identifier

Type="http://www.w3.org/2000/09/xmldsig#SPKIData " (this can be used within a RetrievalMethod or Reference element to identify the referent's type)

The SPKIData element within KeyInfo is used to convey information related to SPKI public key pairs, certificates and other SPKI data. SPKISexp is the base64 encoding of a SPKI canonical S-expression. SPKIData must have at least one SPKISexp; SPKISexp can be complemented/extended by siblings from an external namespace within SPKIData, or SPKIData can be entirely replaced with an alternative SPKI XML structure as a child of KeyInfo.

```
Schema Definition:

<element name="SPKIData" type="ds:SPKIDataType"/>

<complexType name="SPKIDataType">
  <sequence maxOccurs="unbounded">
    <element name="SPKISexp" type="base64Binary"/>
    <any namespace="##other" processContents="lax" minOccurs="0"/>
  </sequence>
</complexType>
```

7.7 The MgmtData Element

Identifier
Type="http://www.w3.org/2000/09/xmldsig#MgmtData " (this can be used within a RetrievalMethod or Reference element to identify the referent's type)

The MgmtData element within KeyInfo is a string value used to convey in-band key distribution or agreement data. However, use of this element is NOT RECOMMENDED and SHOULD NOT be used. The section 7.8 XML Encryption EncryptedKey and DerivedKey Elements describes new KeyInfo types for conveying key information.

7.8 XML Encryption EncryptedKey and DerivedKey Elements

The <xenc:EncryptedKey> and <xenc:DerivedKey> elements defined in [XMLENC-CORE1] as children of ds:KeyInfo can be used to convey in-band encrypted or derived key material. In particular, the xenc:DerivedKey> element may be present when the key used in calculating a Message Authentication Code is derived from a shared secret.

7.9 The dsig11:DEREncodedKeyValue Element

Identifier
Type="http://www.w3.org/2009/xmldsig11#DEREncodedKeyValue"
(this can be used within a RetrievalMethod or Reference element to identify the referent's type)

The public key algorithm and value are DER-encoded in accordance with the value that would be used in the Subject Public Key Info field of an X.509 certificate, per section 4.1.2.7 of [RFC5280]. The DER-encoded value is then base64-encoded.

For the key value types supported in this specification, refer to the following for normative references on the format of Subject Public Key Info and the relevant OID values that identify the key/algorithm type:

RSA
See section 2.3.1 of [RFC3279]
DSA
See section 2.3.2 of [RFC3279]
EC
See section 2 of [RFC5480]

Specifications that define additional key types should provide such a normative reference for their own key types where possible.

```
Schema Definition:

<!-- targetNamespace="http://www.w3.org/2009/xmldsig11#" -->

<element name="DEREncodedKeyValue" type="dsig11:DEREncodedKeyValueType" />

<complexType name="DEREncodedKeyValueType">
  <simpleContent>
    <extension base="base64Binary">
      <attribute name="Id" type="ID" use="optional"/>
    </extension>
  </simpleContent>
</complexType>
```

Historical note: The dsig11:DEREncodedKeyValue element was added to XML Signature 1.1 in order to support certain interoperability scenarios where at least one of signer and/or verifier are not able to serialize keys in the XML formats described in section 7.2 The KeyValue Element above. The KeyValue element is to be used for "bare" XML key representations (not XML wrappings around other binary encodings like ASN.1 DER); for this reason the dsig11:DEREncodedKeyValue element is not a child of KeyValue. The dsig11:DEREncodedKeyValue element is also not a child of the X509Data element, as the keys represented by dsig11:DEREncodedKeyValue may not have X.509 certificates associated with them (a requirement for X509Data).

7.10 The dsig11:KeyInfoReference Element

A dsig11:KeyInfoReference element within KeyInfo is used to convey a reference to a KeyInfo element at another location in the same or different document. For example, several signatures in a document might use a key verified by an X.509v3 certificate chain appearing once in the document or remotely outside the document; each signature's KeyInfo can reference this chain using a single dsig11:KeyInfoReference element instead of including the entire chain with a sequence of X509Certificate elements repeated in multiple places.

dsig11:KeyInfoReference uses the same syntax and dereferencing behavior as Reference's URI (section B.4 The URI Attribute in "Compatibility Mode") and the Reference Processing Model (section B.4.1 The "Compatibility Mode" Reference Processing Model) except that there are no child elements and the presence of the URI attribute is mandatory.

The result of dereferencing a dsig11:KeyInfoReference MUST be a KeyInfo element, or an XML document with a KeyInfo element as the root.

Note: The KeyInfoReference element is a desirable alternative to the use of RetrievalMethod when the data being referred to is a KeyInfo element and the use of RetrievalMethod would require one or more Transform child elements, which introduce security risk and implementation challenges.

```
Schema Definition:

<!-- targetNamespace="http://www.w3.org/2009/xmldsig11#" -->

<element name="KeyInfoReference" type="dsig11:KeyInfoReferenceType"/>

<complexType name="KeyInfoReferenceType">
  <attribute name="URI" type="anyURI" use="required"/>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

8. The Object Element

Identifier
Type="http://www.w3.org/2000/09/xmldsig#Object"
(this can be used within a Reference element to identify the referent's type)

`Object` is an optional element that may occur one or more times. When present, this element may contain any data. The `Object` element may include optional MIME type, ID, and encoding attributes.

The `Object`'s `Encoding` attributed may be used to provide a URI that identifies the method by which the object is encoded (e.g., a binary file).

The `MimeType` attribute is an optional attribute which describes the data within the `Object` (independent of its encoding). This is a string with values defined by [RFC2045]. For example, if the `Object` contains base64 encoded PNG, the `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 by this specification. Applications that require normative type and encoding information for signature validation should rely on `Algorithm` in the `dsig2:Selection` element ("2.0 Mode") or specify `Transforms` with well defined resulting types and/or encodings ("Compatibility Mode").

The `Object`'s `Id` is commonly referenced from a `Reference` in `SignedInfo`, or `Manifest`. This element is typically used for enveloping signatures where the object being signed is to be included in the signature element. The digest is calculated over the entire `Object` element including start and end tags.

Note, if the application wishes to exclude the `<Object>` tags from the digest calculation the `Reference` must identify the actual data object using standard Referencing mechanisms. e.g.

- if the data object is a single XML subtree, then use an ID based reference to the data object.
- if the data object is multiple XML subtrees under the `<Object>` tag, then use an XPath Transform ("Compatibility Mode") or `dsig2:IncludedXPath` ("2.0 Mode") to refer to these nodes. Note in "2.0 Mode" it is not possible to refer to non-element nodes.
- if the data object is base64 text, then use a Base64 transform ("Compatibility Mode") or `dsig2:Selection` with a `Algorithm="http://www.w3.org/2010/xmldsig2#binaryfromBase64"` ("2.0 Mode")
- if the data is something else, then use a custom `Transform` ("Compatibility Mode") or `dsig2:Selection` ("2.0 Mode").

Exclusion of the object tags may be desired for cases where one wants the signature to remain valid if the data object is moved from inside a signature to outside the signature (or vice versa), or where the content of the Object is an encoding of an original binary document and it is desired to extract and decode so as to sign the original bitwise representation.

```
Schema Definition:

<element name="Object" type="ds:ObjectType" />

<complexType name="ObjectType" mixed="true">
  <sequence minOccurs="0" maxOccurs="unbounded">
    <any namespace="##any" processContents="lax" />
  </sequence>
  <attribute name="Id" type="ID" use="optional" />
  <attribute name="MimeType" type="string" use="optional" />
  <attribute name="Encoding" type="anyURI" use="optional" />
</complexType>
```

9. Additional Signature Syntax

This section describes the optional to implement `Manifest` and `SignatureProperties` elements and describes the handling of XML processing instructions and comments. With respect to the elements `Manifest` and `SignatureProperties` this section specifies syntax and little behavior -- it is left to the application. These elements can appear anywhere the parent's content model permits; the `Signature` content model only permits them within `Object`.

9.1 The `Manifest` Element

Identifier

`Type="http://www.w3.org/2000/09/xmldsig#Manifest"` (this can be used within a `Reference` element to identify the referent's type)

The `Manifest` element provides a list of `Reference`s. The difference from the list in `SignedInfo` is that it is application-defined which, if any, of the digests are actually checked against the objects referenced and what to do if the object is inaccessible or the digest compare fails. If a `Manifest` is pointed to from `SignedInfo`, the digest over the `Manifest` itself will be checked by the core signature validation behavior. The digests within such a `Manifest` are checked at the application's discretion. If a `Manifest` is referenced from another `Manifest`, even the overall digest of this two level deep `Manifest` might not be checked.

```
Schema Definition:

<element name="Manifest" type="ds:ManifestType" />

<complexType name="ManifestType">
  <sequence>
    <element ref="ds:Reference" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Id" type="ID" use="optional" />
</complexType>
```

9.2 The `SignatureProperties` Element

Identifier

`Type="http://www.w3.org/2000/09/xmldsig#SignatureProperties"` (this can be used within a `Reference` element to identify the referent's type)

Additional information items concerning the generation of the signature(s) can be placed in a `SignatureProperty` element (i.e., date/time stamp or the serial number of cryptographic hardware used in signature generation).

```
Schema Definition:

<element name="SignatureProperties" type="ds:SignaturePropertiesType" />

<complexType name="SignaturePropertiesType">
  <sequence>
    <element ref="ds:SignatureProperty" maxOccurs="unbounded" />
  </sequence>
  <attribute name="Id" type="ID" use="optional" />
</complexType>

<element name="SignatureProperty" type="ds:SignaturePropertyType" />
```

```
<complexType name="SignaturePropertyType" mixed="true">
  <choice maxOccurs="unbounded">
    <any namespace="##other" processContents="lax" />
    <!-- (1,1) elements from (1,unbounded) namespaces -->
  </choice>
  <attribute name="Target" type="anyURI" use="required" />
  <attribute name="Id" type="ID" use="optional" />
</complexType>
```

9.3 Processing Instructions in Signature Elements

No XML processing instructions (PIs) are used by this specification.

Note that PIs placed inside `SignedInfo` by an application will be signed unless the `CanonicalizationMethod` algorithm discards them. (This is true for any signed XML content.) All of the canonicalization algorithms identified within this specification retain PIs. When a PI is part of content that is signed (e.g., within `SignedInfo` or referenced XML documents) any change to the PI will obviously result in a signature failure.

9.4 Comments in Signature Elements

XML comments are not used by this specification.

Note that unless the `CanonicalizationMethod` removes comments within `SignedInfo` or any other referenced XML (which [XML-C14N] does), they will be signed. Consequently, if they are retained, a change to the comment will cause a signature failure. Similarly, the XML signature over any XML data will be sensitive to comment changes unless a comment-ignoring canonicalization/transform method, such as the Canonical XML [XML-C14N], is specified.

10. Algorithms

10.1 Message Digests

This specification defines several possible digest algorithms for the DigestMethod element, including REQUIRED algorithm SHA-256. Use of SHA-256 is strongly recommended over SHA-1 because recent advances in cryptanalysis (see e.g. [SHA-1-Analysis]) have cast doubt on the long-term collision resistance of SHA-1. Therefore, SHA-1 support is REQUIRED in this specification only for backwards-compatibility reasons.

Digest algorithms that are known not to be collision resistant SHOULD NOT be used in DigestMethod elements. For example, the MD5 message digest algorithm SHOULD NOT be used as specific collisions have been demonstrated for that algorithm.

10.1.1 SHA-1

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

> NOTE
> Use of SHA-256 is strongly recommended over SHA-1 because recent advances in cryptanalysis (see e.g. [SHA-1-Analysis], [SHA-1-Collisions]) have cast doubt on the long-term collision resistance of SHA-1.

The SHA-1 algorithm [FIPS-186-3] takes no explicit parameters. An example of an SHA-1 DigestAlg element is:

> EXAMPLE 9

```
<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:

> EXAMPLE 10

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

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

> EXAMPLE 11

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

10.1.2 SHA-224

Identifier:
http://www.w3.org/2001/04/xmldsig-more#sha224

The SHA-224 algorithm [FIPS-180-3] takes no explicit parameters. A SHA-224 digest is a 224-bit string. The content of the DigestValue element shall be the base64 encoding of this bit string viewed as a 28-octet octet stream.

10.1.3 SHA-256

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

The SHA-256 algorithm [FIPS-180-3] takes no explicit parameters. 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.

10.1.4 SHA-384

Identifier:
http://www.w3.org/2000/09/xmldsig#sha384

The SHA-384 algorithm [FIPS-180-3] takes no explicit parameters. A SHA-384 digest is a 384-bit string. The content of the DigestValue element shall be the base64 encoding of this bit string viewed as a 48-octet octet stream.

10.1.5 SHA-512

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

The SHA-512 algorithm [FIPS-180-3] takes no explicit parameters. 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.

10.2 Message Authentication Codes

MAC algorithms take two implicit parameters, their keying material determined from `KeyInfo` and the octet stream output by `CanonicalizationMethod`. MACs and signature algorithms are syntactically identical but a MAC implies a shared secret key.

10.2.1 HMAC

Identifier:
http://www.w3.org/2000/09/xmldsig#hmac-sha1
http://www.w3.org/2001/04/xmldsig-more#hmac-sha224
http://www.w3.org/2001/04/xmldsig-more#hmac-sha256
http://www.w3.org/2001/04/xmldsig-more#hmac-sha384
http://www.w3.org/2001/04/xmldsig-more#hmac-sha512

The HMAC algorithm (RFC2104 [HMAC]) takes the output (truncation) length in bits as a parameter; this specification REQUIRES that the truncation length be a multiple of 8 (i.e. fall on a byte boundary) because Base64 encoding operates on full bytes. If the truncation parameter is not specified then all the bits of the hash are output. Any signature with a truncation length that is less than half the output length of the underlying hash algorithm `MUST` be deemed invalid. An example of an HMAC `SignatureMethod` element:

EXAMPLE 12

```
<SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1">
  <HMACOutputLength>128</HMACOutputLength>
</SignatureMethod>
```

The output of the HMAC algorithm is ultimately the output (possibly truncated) of the chosen digest algorithm. This value shall be base64 encoded in the same straightforward fashion as the output of the digest algorithms. Example: the `SignatureValue` element for the HMAC-SHA1 digest

EXAMPLE 13

```
9294727A 3638BB1C 13F48EF8 158BFC9D
```

from the test vectors in [HMAC] would be

EXAMPLE 14

```
<SignatureValue>kpRyejY4uxwT9I74FYv8nQ==</SignatureValue>
```

```
Schema Definition:

<simpleType name="HMACOutputLengthType">
  <restriction base="integer" />
</simpleType>
```

10.3 Signature Algorithms

Signature algorithms take two implicit parameters, their keying material determined from `KeyInfo` and the octet stream output by `CanonicalizationMethod`. Signature and MAC algorithms are syntactically identical but a signature implies public key cryptography.

10.3.1 DSA

Identifier:
http://www.w3.org/2000/09/xmldsig#dsa-sha1
http://www.w3.org/2009/xmldsig11#dsa-sha256

The DSA family of algorithms is defined in FIPS 186-3 [FIPS-186-3]. FIPS 186-3 defines DSA in terms of two security parameters L and N where L = |p|, N = |q|, p is the prime modulus, q is a prime divisor of (p-1). FIPS 186-3 defines four valid pairs of (L, N); they are: (1024, 160), (2048, 224), (2048, 256) and (3072, 256). The pair (1024, 160) corresponds to the algorithm DSAwithSHA1, which is identified in this specification by the URI http://www.w3.org/2000/09/xmldsig#dsa-sha1. The pairs (2048, 256) and (3072, 256) correspond to the algorithm DSAwithSHA256, which is identified in this specification by the URI http://www.w3.org/2009/xmldsig11#dsa-sha256. This specification does not use the (2048, 224) instance of DSA (which corresponds to DSAwithSHA224).

DSA takes no explicit parameters; an example of a DSA `SignatureMethod` element is:

```
<SignatureMethod Algorithm="http://www.w3.org/2009/xmldsig11#dsa-sha256"/>
```

The output of the DSA algorithm consists of a pair of integers usually referred by the pair (r, s). The signature value consists of the base64 encoding of the concatenation of two octet-streams that respectively result from the octet-encoding of the values r and s in that order. Integer to octet-stream conversion must be done according to the I2OSP operation defined in the RFC 3447 [PKCS1] specification with a `l` parameter equal to 20. For example, the `SignatureValue` element for a DSA signature (`r`, `s`) with values specified in hexadecimal:

EXAMPLE 15

```
r = 8BAC1AB6 6410435C B7181F95 B16AB97C 92B341C0
s = 41E2345F 1F56DF24 58F426D1 55B4BA2D B6DCD8C8
```

from the example in Appendix 5 of the DSS standard would be

EXAMPLE 16

```
<SignatureValue>
i6watmQQQ1y3GB+VsWq5fJKzQcBB4jRfH1bfJFj0JtFVtLotttzYyA==</SignatureValue>
```

Security considerations regarding DSA key sizes

Per FIPS 186-3 [FIPS-186-3], the DSA security parameter L is defined to be 1024, 2048 or 3072 bits and the corresponding DSA q value is defined to be 160, 224/256 and 256 bits respectively.

NIST provides guidance on the use of keys of various strength for various time frames in special Publication SP 800-57 Part 1 [SP800-57]. Implementers should consult this publication for guidance on acceptable key lengths for applications, however 2048-bit public keys are the minimum recommended key length and 3072-bit keys are recommended for securing information beyond 2030. SP800-57 Part 1 states that DSA 1024-bit key sizes should not be used except to verify and honor signatures created using older legacy systems.

Since XML Signature 1.0 requires implementations to support DSA-based digital signatures, XML Signature 1.1 allows verifiers to verify DSA signatures for DSA keys of 1024 bits in order to validate existing signatures. XML Signature 2.0 maintains compatibility with XML Signature 1.1 for this functionality. XML Signature 2.0 implementations MAY but are NOT REQUIRED to support DSA-based signature generation. Given the short key size and SP800-57 guidelines, DSA with 1024-bit prime moduli SHOULD NOT be used to create signatures. DSA with 1024-bit prime moduli MAY be used to verify older legacy signatures, with an understanding of the associated risks. Important older signatures SHOULD be re-signed with stronger signatures.

10.3.2 RSA (PKCS#1 v1.5)

Identifier:

http://www.w3.org/2000/09/xmldsig#rsa-sha1
http://www.w3.org/2001/04/xmldsig-more#rsa-sha224
http://www.w3.org/2001/04/xmldsig-more#rsa-sha256
http://www.w3.org/2001/04/xmldsig-more#rsa-sha384
http://www.w3.org/2001/04/xmldsig-more#rsa-sha512

The expression "RSA algorithm" as used in this specification refers to the RSASSA-PKCS1-v1_5 algorithm described in RFC 3447 [PKCS1]. The RSA algorithm takes no explicit parameters. An example of an RSA SignatureMethod element is:

EXAMPLE 17

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

The `SignatureValue` content for an RSA signature is the base64 [RFC2045] encoding of the octet string computed as per RFC 3447 [PKCS1], section 8.2.1: Signature generation for the RSASSA-PKCS1-v1_5 signature scheme]. Computation of the signature will require concatenation of the hash value and a constant string determined by RFC 3447. Signature computation and verification does not require implementation of an ASN.1 parser.

The resulting base64 [RFC2045] string is the value of the child text node of the SignatureValue element, e.g.

EXAMPLE 18

```
<SignatureValue>
IWijxQjUrcXBYoCei4QxjWo9Kg8D3p9tlWoT4t0/gyTE96639In0FZFY2/rvP+/bMJ01EArmKZsR5VW3rwoPxw=
</SignatureValue>
```

Security considerations regarding RSA key sizes

NIST provides guidance on the use of keys of various strength for various time frames in special Publication SP 800-57 Part 1 [SP800-57]. Implementers should consult this publication for guidance on acceptable key lengths for applications, however 2048-bit public keys are the minimum recommended key length and 3072-bit keys are recommended for securing information beyond 2030.

All conforming implementations of XML Signature 2.0 MUST support RSA signature generation and verification with public keys at least 2048 bits in length. RSA public keys of 1024 bits or less SHOULD NOT be used to create new signatures but MAY be used to verify signatures created by older legacy systems. XML Signature 2.0 implementations MUST use at least 2048-bit keys for creating signatures, and SHOULD use at least 3072-bit keys for signatures that will be verified beyond 2030.

10.3.3 ECDSA

Identifiers:

http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha1
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha224
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256

http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384
http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512

The ECDSA algorithm [FIPS-186-3] takes no explicit parameters. An example of a ECDSA `SignatureMethod` element is:

EXAMPLE 19

```
<SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha256" />
```

The output of the ECDSA algorithm consists of a pair of integers usually referred by the pair (r, s). The signature value consists of the base64 encoding of the concatenation of two octet-streams that respectively result from the octet-encoding of the values r and s in that order. Integer to octet-stream conversion must be done according to the I2OSP operation defined in the RFC 3447 [PKCS1] specification with the `l` parameter equal to the size of the base point order of the curve in bytes (e.g. 32 for the P-256 curve and 66 for the P-521 curve).

This specification REQUIRES implementations to implement an algorithm that leads to the same results as ECDSA over the P-256 prime curve specified in Section D.2.3 of FIPS 186-3 [FIPS-186-3] (and using the SHA-256 hash algorithm), referred to as the ECDSAwithSHA256 signature algorithm [ECC-ALGS]. It is further RECOMMENDED that implementations also implement algorithms that lead to the same results as ECDSA over the P-384 and P-521 prime curves; these curves are defined in Sections D.2.4 and D.2.5 of FIPS 186-3, respectively [ECC-ALGS].

Note: As described in IETF RFC 6090, the Elliptic Curve DSA (ECDSA) and KT-I signature methods are mathematically and functionally equivalent for fields of characteristic greater than three. See IETF RFC 6090 Section 7.2 [ECC-ALGS].

10.4 Canonicalization Algorithms

The input to any canonicalization algorithm compatible with XML Signature 2.0 signatures is a set of document subtrees and exclusions in the form of subtrees or XML attributes. The actual representation of these inputs depends on the processing model and may be in terms of DOM nodes or representations suitable for streaming-based processing. The output is an octet stream.

Note: The input passed to "2.0 Mode" canonicalization algorithms MUST always exclude the current `Signature` element node (i.e., the `Signature` MUST be passed as one of the exclusion elements. This is equivalent to an implicit Enveloped Signature Transform in "Compatibility Mode", and has no effect for non-enveloped signatures.

This specification REQUIRES implementation of Canonical XML 2.0 [XML-C14N20]. Applications MAY support other canonicalization algorithms with the same input model (subtrees with exclusions). A `Reference` to non-XML data may not use canonicalization at all, or may use a custom canonicalization algorithm with this input model or a completely different one.

10.4.1 Canonical XML 2.0

Identifier for Canonical XML 2.0:
http://www.w3.org/2010/xml-c14n2

An example of a Canonical XML 2.0 element is:

There is no Canonical XML 2.0 `Transform`. Instead the same `CanonicalizationMethod` element is reused within the `dsig2:Selection` element for specifying canonicalization of referenced data,

The normative specification of Canonical XML 2.0 is [XML-C14N20].

10.5 The `Transform` Algorithm

In XML Signature 2.0, the `Transforms` element contains exactly one `Transform` element with an `Algorithm` of `"http://www.w3.org/2010/xmldsig2#transform"`. This transform encapsulates the process of selecting the content to sign, canonicalizing it, and attaching optional material that may aid the verifier.

This fixed `Transform` element consists of a single required `dsig2:Selection` element, followed by an optional `CanonicalizationMethod` element, and an optional `dsig2:Verifications` element.

10.6 `dsig2:Selection` Algorithms

10.6.1 Selection of XML Documents or Fragments

Identifier:
http://www.w3.org/2010/xmldsig2#xml

This `dsig2:Selection` algorithm allows the selection of XML documents or fragments.

The required `URI` attribute can be an external or same-document reference. External references are parsed into an XML document or event stream for the subsequent selection process to operate upon.

- Same-document references take the form of an empty value (e.g `URI=""`) or a fragment (e.g `URI="#foo"`). The former refers to the entire document, while the latter refers to a subtree rooted at the element with the "ID" contained in the fragment.
- External references may be complete external documents (e.g. `URI="http://example.com/bar.xml"`) or refer to fragments of external documents (e.g. `URI="http://example.com/bar.xml#chapter1"`).

The differences between the processing, and allowed syntax, of this `URI` attribute and that of a "Compatibility Mode" `Reference URI` are:

- Dereferencing a same-document reference does not result in a XPath node set.
- The `xpointer` syntax is not permitted.
- There is no comment node removal during the dereferencing process.

The `dsig2:IncludedXPath` MUST NOT be present, if the `URI` contains a fragment identifier. The `dsig2:ExcludedXPath` maybe present even if there is a fragment identifier. I.e the `dsig2:Selection` MUST have one of the following

- `URI` attribute with or without a fragment identifier.

- `URI` attribute with or without a fragment identifier, and one `dsig2:ExcludedXPath` parameter element.
- Non-fragment `URI` attribute and one `dsig2:IncludedXPath` parameter element.
- Non-fragment `URI` attribute, one `dsig2:IncludedXPath` parameter element and one `dsig2:ExcludedXPath` parameter element.

Note: When an `IncludedXPath` or `ExcludedXPath` selects an element node, it implies that the whole subtree rooted at that element is included or excluded.

Processing of the selection and parameters is as follows:

1. Remove the fragment part of the URI if present, and then dereference the URI into a XML document.
2. Do one of the following:
    - If there is a fragment identifier in the URI, search for an element with the ID in the fragment, and then add the element to the "inclusion" list.
    - OR If the `dsig2:IncludedXPath` element is present, evaluate this XPath at the root of document to select element node(s),then add them to the "inclusion" list.
    - OR If neither the fragment identifier or IncludedXPath is present, then add the document node to the "inclusion" list.
3. If the `dsig2:ExcludedXPath` is present, evaluate it at the root of the document to select element and or attribute nodes(s), then add them to the "exclusion list".
4. Add the current `Signature` element under computation/evaluation to the "exclusion list".

Initialize the XPath evaluation context for the `dsig2:IncludedXPath` element and the `dsig2:ExcludedXPath` as follows:

- A **context node** equal to the root of the document.
- A **context position**, initialized to 1.
- A **context size**, initialized to 1.
- A **library of functions** equal to the function set defined in [XMLDSIG-XPATH]. (Note: The XPath function `here()` defined in Xpath Filter Transform MUST NOT be used in this context)
- A set of variable bindings. No means for initializing these is defined. Thus, the set of variable bindings used when evaluating the XPath expression is empty, and use of a variable reference in the XPath expression results in an error.
- The set of namespace declarations in scope for the XPath expression.

The result of the selection process is a set of one or more element nodes, and a set of zero or more exclusions consisting of element and/or attribute nodes.

Note: In a "streaming mode" of evaluation, the XPath evaluation, the canonicalization and digesting need to happen in a pipeline. This is described in Section "2.1 Streaming for XPath Signatures" in [XMLDSIG-XPATH].

10.6.1.1 The `dsig2:IncludedXPath` Element

The `dsig2:IncludedXPath` element is used in conjunction with XML-based `dsig2:Selection` algorithms to specify the subtree(s) to include in a selection. The element contains an XPath 1.0 expression that is evaluated in the context of the root of the XML document.

For example, `"/Book/Chapter"` refers to the subtrees rooted by all `Chapter` child elements of all `Book` child elements of the document root.

The XPath 1.0 expression MUST evaluate only to element nodes, and MUST conform to the XML Signature Streaming Profile of XPath 1.0 [XMLDSIG-XPATH]. Implementations are not required to use a streaming XPath processor, but the expressions used MUST conform to the streaming profile to ensure compatibility with implementations that do use a streaming processor.

10.6.1.2 The `dsig2:ExcludedXPath` Element

The `dsig2:ExcludedXPath` element is used in conjunction with XML-based `dsig2:Selection` algorithms to specify subtree(s) and/or attributes to exclude from a selection. The element contains an XPath 1.0 expression that is evaluated in the context of the root of the XML document.

For example, `"/Book/Chapter"` refers to the subtrees rooted by all `Chapter` child elements of all `Book` child elements of the document root.

The XPath 1.0 expression MUST evaluate to element and/or attribute nodes, and MUST conform to the XML Signature Streaming Profile of XPath 1.0 [XMLDSIG-XPATH]. Implementations are not required to use a streaming XPath processor, but the expressions used MUST conform to the streaming profile to ensure compatibility with implementations that do use a streaming processor.

10.6.1.3 The `dsig2:ByteRange` Element

The `dsig2:ByteRange` element is used in conjunction with binary `dsig2:Selection` algorithms to specify byte range subsets of the originally selected octet stream to include.

The element value MUST conform to the Byte Ranges syntax described in section 14.35.1 of [HTTP11].

For example, element content of `0-20,220-270,320-` indicates that the first 21 bytes, then bytes 220 through 270, and finally bytes 320 through the rest of the stream are included.

10.6.2 Selection of External Binary Data

Identifier:
http://www.w3.org/2010/xmldsig2#binaryExternal

This `dsig2:Selection` algorithm allows the selection of external binary data.

The required `URI` attribute MUST be an external reference and the result of dereferencing it is treated as an octet stream.

The `dsig2:Selection` element MAY contain at most one `dsig2:ByteRange` parameter element to modify the selection result. If present, the range(s) indicated modify the resulting octet stream obtained from the `URI`. The implementation MAY incorporate the byte range into the dereferencing process as an optimization.

The final result of the selection process is an octet stream.

10.6.3 Selection of Binary Data within XML

Identifier:
http://www.w3.org/2010/xmldsig2#binaryfromBase64

This `dsig2:Selection` algorithm allows the selection of base64-encoded binary data from a Text node within an XML document.

The required `URI` attribute can be an external or same-document reference. External references are parsed into an XML document or event stream for the subsequent selection process to operate upon.

- Same-document references take the form of an empty value (e.g `URI=""`) or a fragment (e.g `URI="#foo"`). The former refers to the entire document, while the latter refers to a subtree rooted at the element with the "ID" contained in the fragment.
- External references may be complete external documents (e.g. `URI="http://example.com/bar.xml"`) or refer to fragments of external documents (e.g. `URI="http://example.com/bar.xml#chapter1"`).

The differences between the processing, and allowed syntax, of this `URI` attribute and that of a "Compatibility Mode" `Reference URI` are:

- Dereferencing a same-document reference does not result in a XPath node set.
- The `xpointer` syntax is not permitted.
- There is no comment node removal during the dereferencing process.

The `dsig2:Selection` element MAY contain at most one `dsig2:IncludedXPath` and at most one `dsig2:ByteRange` parameter element to modify the selection result. However `dsig2:IncludedXPath` MUST NOT be present, if the `URI` contains a fragment identifier.

Processing of the selection and parameters is as follows:

1. Remove the fragment part of the URI if present, and then dereference the URI into a XML document.
2. Do one of the following:
    - If there is a fragment identifier in the URI, search for an element with the ID in the fragment, and then select this element.
    - OR If the `IncludedXPath` element is present, evaluate this XPath at the root of document to select one element node. It is an error if the XPath returns more than one element node.
    - OR If neither the fragment identifier or IncludedXPath is present, then select the root element node of the document.
3. The selected element node MUST contain only Text node children, or an error results.
4. Coalesce the selected element's Text node children into a single string, and base64-decode the result to obtain an octet stream.
5. If a `dsig2:ByteRange` parameter is present, use these range(s) to modify the octet stream obtained in the previous step.

The final result of the selection process is an octet stream.

10.7 The `dsig2:Verification` Types

10.7.1 DigestDataLength

Identifier:
http://www.w3.org/2010/xmldsig2#DigestDataLength

The DigestDataLength `dsig2:Verification` type contains an integer that specifies the number of bytes that were digested for the containing `Reference`. This can be used for multiple purposes:

- to debug digest verification failures
- to indicate intentional signing of 0 bytes, such as if an XPath expression selects nothing
- to bypass the expensive digest calculation if during verification the length of the byte array containing the canonicalized bytes doesn't match the value found in the message

The non-negative integer value is carried within a `DigestDataLength` attribute inside the `dsig2:Verification` element.

10.7.2 PositionAssertion

Identifier:
http://www.w3.org/2010/xmldsig2#PositionAssertion

The PositionAssertion `dsig2:Verification` type is used to increase the resistance of ID-based referencing to signature wrapping attacks. It contains an XPath expression that must match the referenced content's position in the document. Thus, instead of "selecting" the referenced element via an XPath, its position is verified by one (which enables flexibility in the actual use of XPath by the signer or verifier). The actual selection process remains ID-based, which is simpler for many implementers.

The XPath expression is carried within a `PositionAssertion` attribute inside the `dsig2:Verification` element.

While using the PositionAssertion feature allows more flexibility in accomodating XPath-unaware signers and verifiers, applications SHOULD favor the use of XPath-based selection via the `dsig2:IncludedXPath` element over the use of this feature in most cases. Because verification of the PositionAssertion is formally optional, verifiers may become subject to positional wrapping attacks if they choose to ignore the assertion. This feature is appropriate mainly in applications in which knowledge of the verifier's support for the feature can be assured.

10.7.3 IDAttributes

Identifier:
http://www.w3.org/2010/xmldsig2#IDAttributes

The IDAttributes `dsig2:Verification` type is used in conjunction with ID-based references, to specify the ID attribute node name that the signer used. Ordinarily, ID attribute knowledge is imparted through a variety of normative and informal means, including DTDs, XML Schemas, use of xml:id, and application-specific content knowledge. A signer is not required to use this mechanism to identify ID attributes, but MAY do so to transfer its own ID knowledge to the verifier through the signature itself. Verifiers MAY incorporate this knowledge, or use more traditional means of recognizing ID attributes.

The `dsig2:Verification` element specifies exactly one ID attribute node. This MUST be the name of the node involved in the containing `Reference`.

The `dsig2:Verification` element MUST contain one of the following two child elements:

`dsig2:QualifiedAttr`
: Specifies a namespace-qualified ID attribute node, by means of `Name` and `NS` attributes.

`dsig2:UnqualifiedAttr`
: Specifies an unqualified ID attribute node, by means of a required `Name` attribute, and required `ParentName` and optional `ParentNS` attributes to identify the owning element.

> NOTE
>
> Without a DTD, there is technically no way to define IDness in an XML document. In practice, this typing was extended to documents validated by an XML Schema, and then to the creation of `xml:id`. Unfortunately, DTDs have mostly fallen out of use in many contexts, and schemas are expensive, rarely used in many runtime scenarios, and can't be relied on to be completely known by the verifier in the presence of extensible XML scenarios.
> `xml:id` has not yet seen wide adoption, mainly because a lot of the standards that needed it (SAML, WS-Security) were completed prior to its invention.
> The result is that applications that rely on ID-based references for signing have typically made insecure assumptions about the IDness of attributes based on their name (`ID`, `id`, `Id`, etc.), or have to provide APIs for applications to call before verification (which is also a problem in the face of extensibility). DOM Level 3, which is now fairly widely implemented, also provides the ability to identify attributes as an ID at runtime, although often without guaranteeing the uniqueness property.
> The IDAttributes verification type provides a deterministic way of defining an ID attribute used during signing, that is independent of DTD, XML Schema, DOM 3 or other application-specific mechanisms.

11. XML Canonicalization and Syntax Constraint Considerations

Digital signatures only work if the verification calculations are performed on exactly the same bits as the signing calculations. If the surface representation of the signed data can change between signing and verification, then some way to standardize the changeable aspect must be used before signing and verification. For example, even for simple ASCII text there are at least three widely used line ending sequences. If it is possible for signed text to be modified from one line ending convention to another between the time of signing and signature verification, then the line endings need to be canonicalized to a standard form before signing and verification or the signatures will break.

XML is subject to surface representation changes and to processing which discards some surface information. For this reason, XML digital signatures have a provision for indicating canonicalization methods in the signature so that a verifier can use the same canonicalization as the signer.

Throughout this specification we distinguish between the canonicalization of a `Signature` element and other signed XML data objects. It is possible for an isolated XML document to be treated as if it were binary data so that no changes can occur. In that case, the digest of the document will not change and it need not be canonicalized if it is signed and verified as such. However, XML that is read and processed using standard XML parsing and processing techniques is frequently changed such that some of its surface representation information is lost or modified. In particular, this will occur in many cases for the `Signature` and enclosed `SignedInfo` elements since they, and possibly an encompassing XML document, will be processed as XML.

Similarly, these considerations apply to `Manifest`, `Object`, and `SignatureProperties` elements if those elements have been digested, their `DigestValue` is to be checked, and they are being processed as XML.

The kinds of changes in XML that may need to be canonicalized can be divided into four categories. There are those related to the basic [XML10], as described in 7.1 below. There are those related to [DOM-LEVEL-1], [SAX], or similar processing as described in 7.2 below. Third, there is the possibility of coded character set conversion, such as between UTF-8 and UTF-16, both of which all [XML10] compliant processors are required to support, which is described in the paragraph immediately below. And, fourth, there are changes that related to namespace declaration and XML namespace attribute context as described in 7.3 below.

Any canonicalization algorithm should yield output in a specific fixed coded character set. All canonicalization algorithms identified in this document use UTF-8 (without a byte order mark (BOM)) and do not provide character normalization. We RECOMMEND that signature applications create XML content (`Signature` elements and their descendants/content) in Normalization Form C [NFC] and check that any XML being consumed is in that form as well; (if not, signatures may consequently fail to validate). Additionally, none of these algorithms provide data type normalization. Applications that normalize data types in varying formats (e.g., (true, false) or (1,0)) may not be able to validate each other's signatures.

11.1 XML 1.0 Syntax Constraints, and Canonicalization

XML 1.0 [XML10]] defines an interface where a conformant application reading XML is given certain information from that XML and not other information. In particular,

1. line endings are normalized to the single character #xA by dropping #xD characters if they are immediately followed by a #xA and replacing them with #xA in all other cases,
2. missing attributes declared to have default values are provided to the application as if present with the default value,
3. character references are replaced with the corresponding character,
4. entity references are replaced with the corresponding declared entity,
5. attribute values are normalized by
    1. replacing character and entity references as above,
    2. replacing occurrences of #x9, #xA, and #xD with #x20 (space) except that the sequence #xD#xA is replaced by a single space, and
    3. if the attribute is not declared to be CDATA, stripping all leading and trailing spaces and replacing all interior runs of spaces with a single space.

Note that items (2), (4), and (5.3) depend on the presence of a schema, DTD or similar declarations. The `Signature` element type is laxly schema valid [XMLSCHEMA-1][XMLSCHEMA-2], consequently external XML or even XML within the same document as the signature may be (only) well-formed or from another namespace (where permitted by the signature schema); the noted items may not be present. Thus, a signature with such content will only be verifiable by other signature applications if the following syntax constraints are observed when generating any signed material including the `SignedInfo` element:

1. attributes having default values be explicitly present,
2. all entity references (except "amp", "lt", "gt", "apos", "quot", and other character entities not representable in the encoding chosen) be expanded,
3. attribute value white space be normalized

11.2 DOM/SAX Processing and Canonicalization

In addition to the canonicalization and syntax constraints discussed above, many XML applications use the Document Object Model [DOM-LEVEL-1] or

the Simple API for XML [SAX]. DOM maps XML into a tree structure of nodes and typically assumes it will be used on an entire document with subsequent processing being done on this tree. SAX converts XML into a series of events such as a start tag, content, etc. In either case, many surface characteristics such as the ordering of attributes and insignificant white space within start/end tags is lost. In addition, namespace declarations are mapped over the nodes to which they apply, losing the namespace prefixes in the source text and, in most cases, losing where namespace declarations appeared in the original instance.

If an XML Signature is to be produced or verified on a system using the DOM or SAX processing, a canonical method is needed to serialize the relevant part of a DOM tree or sequence of SAX events. XML canonicalization specifications, such as [XML-C14N], are based only on information which is preserved by DOM and SAX. For an XML Signature to be verifiable by an implementation using DOM or SAX, not only must XML 1.0 syntax constraints given in the section 11.1 XML 1.0 Syntax Constraints, and Canonicalization be followed but an appropriate XML canonicalization MUST be specified so that the verifier can re-serialize DOM/SAX mediated input into the same octet stream that was signed.

12. Security Considerations

The XML Signature specification provides a very flexible digital signature mechanism. Implementers must give consideration to their application threat models and to the following factors. For additional security considerations in implementation and deployment of this specification, see [XMLDSIG-BESTPRACTICES].

12.1 Transforms

A requirement of this specification is to permit signatures to "apply to a part or totality of a XML document." (See [XMLDSIG-REQUIREMENTS], section 3.1.3].) The `Transforms` mechanism meets this requirement by permitting one to sign data derived from processing the content of the identified resource. For instance, applications that wish to sign a form, but permit users to enter limited field data without invalidating a previous signature on the form might use [XPATH] to exclude those portions the user needs to change. `Transforms` may be arbitrarily specified and may include encoding transforms, canonicalization instructions or even XSLT transformations. Three cautions are raised with respect to this feature in the following sections.

Note, core validation behavior does not confirm that the signed data was obtained by applying each step of the indicated transforms. (Though it does check that the digest of the resulting content matches that specified in the signature.) For example, some applications may be satisfied with verifying an XML signature over a cached copy of already transformed data. Other applications might require that content be freshly dereferenced and transformed.

12.1.1 Only What is Signed is Secure

First, obviously, signatures over a transformed document do not secure any information discarded by transforms: only what is signed is secure.

Note that the use of Canonical XML [XML-C14N] ensures that all internal entities and XML namespaces are expanded within the content being signed. All entities are replaced with their definitions and the canonical form explicitly represents the namespace that an element would otherwise inherit. Applications that do not canonicalize XML content (especially the `SignedInfo` element) SHOULD NOT use internal entities and SHOULD represent the namespace explicitly within the content being signed since they can not rely upon canonicalization to do this for them. Also, users concerned with the integrity of the element type definitions associated with the XML instance being signed may wish to sign those definitions as well (i.e., the schema, DTD, or natural language description associated with the namespace/identifier).

Second, an envelope containing signed information is not secured by the signature. For instance, when an encrypted envelope contains a signature, the signature does not protect the authenticity or integrity of unsigned envelope headers nor its ciphertext form, it only secures the plaintext actually signed.

12.1.2 Only What is "Seen" Should be Signed

Additionally, the signature secures any information introduced by the transform: only what is "seen" (that which is represented to the user via visual, auditory or other media) should be signed. If signing is intended to convey the judgment or consent of a user (an automated mechanism or person), then it is normally necessary to secure as exactly as practical the information that was presented to that user. Note that this can be accomplished by literally signing what was presented, such as the screen images shown a user. However, this may result in data which is difficult for subsequent software to manipulate. Instead, one can sign the data along with whatever filters, style sheets, client profile or other information that affects its presentation.

12.1.3 "See" What is Signed

Just as a user should only sign what he or she "sees," persons and automated mechanism that trust the validity of a transformed document on the basis of a valid signature should operate over the data that was transformed (including canonicalization) and signed, not the original pre-transformed data. This recommendation applies to transforms specified within the signature as well as those included as part of the document itself. For instance, if an XML document includes an embedded style sheet [XSLT] it is the transformed document that should be represented to the user and signed. To meet this recommendation where a document references an external style sheet, the content of that external resource should also be signed as via a signature `Reference` otherwise the content of that external content might change which alters the resulting document without invalidating the signature.

Some applications might operate over the original or intermediary data but should be extremely careful about potential weaknesses introduced between the original and transformed data. This is a trust decision about the character and meaning of the transforms that an application needs to make with caution. Consider a canonicalization algorithm that normalizes character case (lower to upper) or character composition ('e and accent' to 'accented-e'). An adversary could introduce changes that are normalized and consequently inconsequential to signature validity but material to a DOM processor. For instance, by changing the case of a character one might influence the result of an XPath selection. A serious risk is introduced if that change is normalized for signature validation but the processor operates over the original data and returns a different result than intended.

As a result:

- All documents operated upon and generated by signature applications MUST be in [NFC] (otherwise intermediate processors might unintentionally break the signature)
- Encoding normalizations SHOULD NOT be done as part of a signature transform, or (to state it another way) if normalization does occur, the application SHOULD always "see" (operate over) the normalized form.

12.2 Check the Security Model

This specification uses public key signatures and keyed hash authentication codes. These have substantially different security models. Furthermore, it permits user specified algorithms which may have other models.

With public key signatures, any number of parties can hold the public key and verify signatures while only the parties with the private key can create signatures. The number of holders of the private key should be minimized and preferably be one. Confidence by verifiers in the public key they are using and its binding to the entity or capabilities represented by the corresponding private key is an important issue, usually addressed by certificate or online

authority systems.

Keyed hash authentication codes, based on secret keys, are typically much more efficient in terms of the computational effort required but have the characteristic that all verifiers need to have possession of the same key as the signer. Thus any verifier can forge signatures.

This specification permits user provided signature algorithms and keying information designators. Such user provided algorithms may have different security models. For example, methods involving biometrics usually depend on a physical characteristic of the authorized user that can not be changed the way public or secret keys can be and may have other security model differences.

12.3 Algorithms, Key Lengths, Certificates, Etc.

The strength of a particular signature depends on all links in the security chain. This includes the signature and digest algorithms used, the strength of the key generation [RANDOM] and the size of the key, the security of key and certificate authentication and distribution mechanisms, certificate chain validation policy, protection of cryptographic processing from hostile observation and tampering, etc.

Care must be exercised by applications in executing the various algorithms that may be specified in an XML signature and in the processing of any "executable content" that might be provided to such algorithms as parameters, such as XSLT transforms. The algorithms specified in this document will usually be implemented via a trusted library but even there perverse parameters might cause unacceptable processing or memory demand. Even more care may be warranted with application defined algorithms.

The security of an overall system will also depend on the security and integrity of its operating procedures, its personnel, and on the administrative enforcement of those procedures. All the factors listed in this section are important to the overall security of a system; however, most are beyond the scope of this specification.

13. Schema

13.1 XSD Schema

XML Signature Core Schema Instance
xmldsig-core-schema.xsd
Valid XML schema instance based on [XMLSCHEMA-1][XMLSCHEMA-2].
XML Signature 1.1 Schema Instance
xmldsig11-schema.xsd
This schema document defines the additional elements defined in the 1.1 version of the XML Signature specification.
XML Signature 1.1 Schema Driver
xmldsig1-schema.xsd
This schema instance binds together the XML Signature Core Schema Instance and the XML Signature 1.1 Schema Instance
XML Signature 2.0 Schema Instance
xmldsig2-schema.xsd
This schema document defines the additional elements defined in this version of the XML Signature specification.

A. Definitions

This section is non-normative.

Authentication Code (Protected Checksum)
A value generated from the application of a shared key to a message via a cryptographic algorithm such that it has the properties of message authentication (and integrity) but not signer authentication. Equivalent to *protected checksum*, "A checksum that is computed for a data object by means that protect against active attacks that would attempt to change the checksum to make it match changes made to the data object." [RFC4949]
Authentication, Message
The property, given an authentication code/protected checksum, that tampering with both the data and checksum, so as to introduce changes while seemingly preserving integrity, are still detected. "A signature should identify what is signed, making it impracticable to falsify or alter either the signed matter or the signature without detection." [ABA-DSIG-GUIDELINES].
Authentication, Signer
The property that the identity of the signer is as claimed. "A signature should indicate who signed a document, message or record, and should be difficult for another person to produce without authorization." [ABA-DSIG-GUIDELINES] Note, signer authentication is an application decision (e.g., does the signing key actually correspond to a specific identity) that is supported by, but out of scope, of this specification.
Checksum
"A value that (a) is computed by a function that is dependent on the contents of a data object and (b) is stored or transmitted together with the object, for the purpose of detecting changes in the data." [RFC4949]
Core
The syntax and processing defined by this specification, including core validation. We use this term to distinguish other markup, processing, and applications semantics from our own.
Data Object (Content/Document)
The actual binary/octet data being operated on (transformed, digested, or signed) by an application -- frequently an HTTP entity [HTTP11]. Note that the proper noun `Object` designates a specific XML element. Occasionally we refer to a data object as a *document* or as a *resource's content*. The term *element content* is used to describe the data between XML start and end tags [XML10]. The term *XML document* is used to describe data objects which conform to the XML specification [XML10].
Integrity
"The property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner." [RFC4949] A simple checksum can provide integrity from incidental changes in the data; message authentication is similar but also protects against an active attack to alter the data whereby a change in the checksum is introduced so as to match the change in the data.
Object
An XML Signature element wherein arbitrary (non-core) data may be placed. An `Object` element is merely one type of digital data (or document) that can be signed via a `Reference`.
Resource
"A resource can be anything that has identity. Familiar examples include an electronic document, an image, a service (e.g., 'today's weather report for Los Angeles'), and a collection of other resources.... The resource is the conceptual mapping to an entity or set of entities, not necessarily the entity which corresponds to that mapping at any particular instance in time. Thus, a resource can remain constant even when its content---the entities to which it currently corresponds---changes over time, provided that the conceptual mapping is not changed in the process." [URI] In order to avoid a collision of the term *entity* within the URI and XML specifications, we use the term *data object*, *content* or *document* to refer to the actual bits/octets being operated upon.

Signature
: Formally speaking, a value generated from the application of a private key to a message via a cryptographic algorithm such that it has the properties of integrity, message authentication and/or signer authentication. (However, we sometimes use the term signature generically such that it encompasses Authentication Code values as well, but we are careful to make the distinction when the property of signer authentication is relevant to the exposition.) A signature may be (non-exclusively) described as detached, enveloping, or enveloped.

Signature, Application
: An application that implements the MANDATORY (REQUIRED/MUST) portions of this specification; these conformance requirements are over application behavior, the structure of the `Signature` element type and its children (including `SignatureValue`) and the specified algorithms.

Signature, Detached
: The signature is over content external to the `Signature` element, and can be identified via a `URI` or transform. Consequently, the signature is "detached" from the content it signs. This definition typically applies to separate data objects, but it also includes the instance where the `Signature` and data object reside within the same XML document but are sibling elements.

Signature, Enveloping
: The signature is over content found within an `Object` element of the signature itself. The `Object` (or its content) is identified via a `Reference` (via a `URI` fragment identifier or transform).

Signature, Enveloped
: The signature is over the XML content that contains the signature as an element. The content provides the root XML document element. Obviously, enveloped signatures must take care not to include their own value in the calculation of the `SignatureValue`.

Transform
: The processing of a data from its source to its derived form. Typical transforms include XML Canonicalization, XPath, and XSLT.

Validation, Core
: The core processing requirements of this specification requiring signature validation and `SignedInfo` reference validation.

Validation, Reference
: The hash value of the identified and transformed content, specified by `Reference`, matches its specified `DigestValue`.

Validation, Signature
: The `SignatureValue` matches the result of processing `SignedInfo` with `CanonicalizationMethod` and `SignatureMethod` as specified in section 4.3 Core Validation.

Validation, Trust/Application
: The application determines that the semantics associated with a signature are valid. For example, an application may validate the time stamps or the integrity of the signer key -- though this behavior is external to this core specification.

XML namespace URI
: This refers to the namespace name [XML-NAMES].

B. Compatibility Mode

Use of the "Compatibility Mode" described in this section enables the XML Signature 1.x model to be used where necessary, to enable backward compatibility.

B.1 "Compatibility Mode" Examples

The following examples are for a detached signature of the content of the HTML4 in XML specification.

B.1.1 Simple Example in "Compatibility Mode"

This example uses "Compatibility Mode".

EXAMPLE 20

```
[s01] <Signature Id="MyFirstSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
[s02]   <SignedInfo>
[s03]   <CanonicalizationMethod Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
[s04]   <SignatureMethod Algorithm="http://www.w3.org/2001/04/xmldsig-more#rsa-sha256"/>
[s05]   <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
[s06]     <Transforms>
[s07]       <Transform Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
[s08]     </Transforms>
[s09]     <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[s10]     <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...</DigestValue>
[s11]   </Reference>
[s12] </SignedInfo>
[s13]   <SignatureValue>...</SignatureValue>
[s14]   <KeyInfo>
[s15a]    <KeyValue>
[s15b]      <DSAKeyValue>
[s15c]        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
[s15d]      </DSAKeyValue>
[s15e]    </KeyValue>
[s16]   </KeyInfo>
[s17] </Signature>
```

`[s02-12]` The required `SignedInfo` element is the information that is actually signed. Core validation of `SignedInfo` consists of two mandatory processes: validation of the signature over `SignedInfo` and validation of each `Reference` digest within `SignedInfo`. Note that the algorithms used in calculating the `SignatureValue` are also included in the signed information while the `SignatureValue` element is outside `SignedInfo`.

`[s03]` The `CanonicalizationMethod` is the algorithm that is used to canonicalize the `SignedInfo` element before it is digested as part of the signature operation. Note that this example is not in canonical form. (None of the examples in this specification are in canonical form.)

`[s04]` The `SignatureMethod` is the algorithm that is used to convert the canonicalized `SignedInfo` into the `SignatureValue`. It is a combination of a digest algorithm and a key dependent algorithm and possibly other algorithms such as padding, for example RSA-SHA1. The algorithm names are signed to resist attacks based on substituting a weaker algorithm. To promote application interoperability we specify a set of signature algorithms that MUST be implemented, though their use is at the discretion of the signature creator. We specify additional algorithms as RECOMMENDED or OPTIONAL for implementation; the design also permits arbitrary user specified algorithms.

`[s05-11]` Each `Reference` element includes the digest method and resulting digest value calculated over the identified data object. It also may include transformations that produced the input to the digest operation. A data object is signed by computing its digest value and a signature over that value. The signature is later checked via reference and signature validation.

[s14-16] KeyInfo indicates the key to be used to validate the signature. Possible forms for identification include certificates, key names, and key agreement algorithms and information -- we define only a few. KeyInfo is optional for two reasons. First, the signer may not wish to reveal key information to all document processing parties. Second, the information may be known within the application's context and need not be represented explicitly. Since KeyInfo is outside of SignedInfo, if the signer wishes to bind the keying information to the signature, a Reference can easily identify and include the KeyInfo as part of the signature. Use of KeyInfo is optional, however note that senders and receivers must agree on how it will be used through a mechanism out of scope for this specification.

B.1.2 More on Reference

These section explaining the lines [s05] to [s11] of the previous example. This signature is in "compatibility mode".

EXAMPLE 21

```
[s05]   <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
[s06]     <Transforms>
[s07]       <Transform Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
[s08]     </Transforms>
[s09]     <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[s10]     <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...</DigestValue>
[s11]   </Reference>
```

[s05] The optional URI attribute of Reference identifies the data object to be signed. This attribute may be omitted on at most one Reference in a Signature. (This limitation is imposed in order to ensure that references and objects may be matched unambiguously.)

[s05-08] This identification, along with the transforms, is a description provided by the signer on how they obtained the signed data object in the form it was digested (i.e. the digested content). The verifier may obtain the digested content in another method so long as the digest verifies. In particular, the verifier may obtain the content from a different location such as a local store than that specified in the URI.

[s06-08] Transforms is an optional ordered list of processing steps that were applied to the resource's content before it was digested. Transforms can include operations such as canonicalization, encoding/decoding (including compression/inflation), XSLT, XPath, XML schema validation, or XInclude. XPath transforms permit the signer to derive an XML document that omits portions of the source document. Consequently those excluded portions can change without affecting signature validity. For example, if the resource being signed encloses the signature itself, such a transform must be used to exclude the signature value from its own computation. If no Transforms element is present, the resource's content is digested directly. While the Working Group has specified mandatory (and optional) canonicalization and decoding algorithms, user specified transforms are permitted.

[s09-10] DigestMethod is the algorithm applied to the data after Transforms is applied (if specified) to yield the DigestValue. The signing of the DigestValue is what binds the content of a resource to the signer's key.

B.1.3 Extended Example (Object and SignatureProperty)

This specification does not address mechanisms for making statements or assertions. Instead, this document defines what it means for something to be signed by an XML Signature (integrity, message authentication, and/or signer authentication). Applications that wish to represent other semantics must rely upon other technologies, such as [XML10], [RDF-PRIMER]. For instance, an application might use a foo:assuredby attribute within its own markup to reference a Signature element. Consequently, it's the application that must understand and know how to make trust decisions given the validity of the signature and the meaning of assuredby syntax. We also define a SignatureProperties element type for the inclusion of assertions about the signature itself (e.g., signature semantics, the time of signing or the serial number of hardware used in cryptographic processes). Such assertions may be signed by including a Reference for the SignatureProperties in SignedInfo. While the signing application should be very careful about what it signs (it should understand what is in the SignatureProperty) a receiving application has no obligation to understand that semantic (though its parent trust engine may wish to). Any content about the signature generation may be located within the SignatureProperty element. The mandatory Target attribute references the Signature element to which the property applies.

Consider the preceding example (in "Compatibility Mode") with an additional reference to a local Object that includes a SignatureProperty element. (Such a signature would not only be detached [p02] but enveloping [p03].)

EXAMPLE 22

```
[   ]  <Signature Id="MySecondSignature" ...>
[p01]  <SignedInfo>
[   ]   ...
[p02]   <Reference URI="http://www.w3.org/TR/xml-stylesheet/">
[   ]   ...
[p03]   <Reference URI="#AMadeUpTimeStamp"
[p04]         Type="http://www.w3.org/2000/09/xmldsig#SignatureProperties">
[p05]    <Transforms>
[p06]      <Transform Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
[p07]    </Transforms>
[p08]    <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[p09]    <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...</DigestValue>
[p10]   </Reference>
[p11]  </SignedInfo>
[p12]  ...
[p13]  <Object>
[p14]   <SignatureProperties>
[p15]     <SignatureProperty Id="AMadeUpTimeStamp" Target="#MySecondSignature">
[p16]        <timestamp xmlns="http://www.ietf.org/rfcXXXX.txt">
[p17]          <date>19990914</date>
[p18]          <time>14:34:34:34</time>
[p19]        </timestamp>
[p20]     </SignatureProperty>
[p21]   </SignatureProperties>
[p22]  </Object>
[p23]</Signature>
```

[p04] The optional Type attribute of Reference provides information about the resource identified by the URI. In particular, it can indicate that it is an Object, SignatureProperty, or Manifest element. This can be used by applications to initiate special processing of some Reference elements. References to an XML data element within an Object element SHOULD identify the actual element pointed to. Where the element content is not XML (perhaps it is binary or encoded data) the reference should identify the Object and the Reference Type, if given, SHOULD indicate Object. Note that Type is advisory and no action

based on it or checking of its correctness is required by core behavior.

[p13] `Object` is an optional element for including data objects within the signature element or elsewhere. The `Object` can be optionally typed and/or encoded.

[p14-21] Signature properties, such as time of signing, can be optionally signed by identifying them from within a `Reference`. (These properties are traditionally called signature "attributes" although that term has no relationship to the XML term "attribute".)

This is the same example in "2.0 Mode". Only the `Reference` content is different.

EXAMPLE 23

```
[   ]  ...
[p03]   <Reference>
[p04]
[p05]    <Transforms>
[p06]      <Transform Algorithm="http://www.w3.org/2010/xmldsig2#transform">
[s06a]       <dsig2:Selection type="http://www.w3.org/2010/xmldsig2#xml" xmlns:dsig2="http://www.w3.org/2010/xmldsig2#"
URI="#AMadeUpTimeStamp"
>
[p06b]       </dsig2:Selection>
[p06c]       <CanonicalizationMethod Algorithm="http://www.w3.org/2010/xml-c14n2"/>
[p06d]     </Transform>
[p07]    </Transforms>
[p08]    <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[p09]    <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...</DigestValue>
[p10]   </Reference>
[   ]   ...
```

B.1.4 Extended Example (`Object` and `Manifest`)

The `Manifest` element is provided to meet additional requirements not directly addressed by the mandatory parts of this specification. Two requirements and the way the `Manifest` satisfies them follow.

First, applications frequently need to efficiently sign multiple data objects even where the signature operation itself is an expensive public key signature. This requirement can be met by including multiple `Reference` elements within `SignedInfo` since the inclusion of each digest secures the data digested. However, some applications may not want the core validation behavior associated with this approach because it requires every `Reference` within `SignedInfo` to undergo reference validation -- the `DigestValue` elements are checked. These applications may wish to reserve reference validation decision logic to themselves. For example, an application might receive a signature valid `SignedInfo` element that includes three `Reference` elements. If a single `Reference` fails (the identified data object when digested does not yield the specified `DigestValue`) the signature would fail core validation. However, the application may wish to treat the signature over the two valid `Reference` elements as valid or take different actions depending on which fails. To accomplish this, `SignedInfo` would reference a `Manifest` element that contains one or more `Reference` elements (with the same structure as those in `SignedInfo`). Then, reference validation of the `Manifest` is under application control.

Second, consider an application where many signatures (using different keys) are applied to a large number of documents. An inefficient solution is to have a separate signature (per key) repeatedly applied to a large `SignedInfo` element (with many `Reference`s); this is wasteful and redundant. A more efficient solution is to include many references in a single `Manifest` that is then referenced from multiple `Signature` elements.

The example (in "Compatibility Mode") below includes a `Reference` that signs a `Manifest` found within the `Object` element.

EXAMPLE 24

```
[   ] ...
[m01]   <Reference URI="#MyFirstManifest"
[m02]     Type="http://www.w3.org/2000/09/xmldsig#Manifest">
[m03]     <Transforms>
[m04]       <Transform Algorithm="http://www.w3.org/2006/12/xml-c14n11"/>
[m05]     </Transforms>
[m06]     <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[m07]     <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...=</DigestValue>
[m08]   </Reference>
[   ] ...
[m09] <Object>
[m10]   <Manifest Id="MyFirstManifest">
[m11]     <Reference>
[m12]     ...
[m13]     </Reference>
[m14]     <Reference>
[m15]     ...
[m16]     </Reference>
[m17]   </Manifest>
[m18] </Object>
```

Here is the modified `Reference` in "2.0 Mode"

EXAMPLE 25

```
[m01]   <Reference
[m02]     Type="http://www.w3.org/2000/09/xmldsig#Manifest">
[m03]     <Transforms>
[m04]      <Transform Algorithm="http://www.w3.org/2010/xmldsig2#transform">
[m04a]       <dsig2:Selection type="http://www.w3.org/2010/xmldsig2#xml" xmlns:dsig2="http://www.w3.org/2010/xmldsig2#"
URI="#MyFirstManifest">
[m04b]       </dsig2:Selection>
[m04c]       <CanonicalizationMethod Algorithm="http://www.w3.org/2010/xml-c14n2"/>
[m04d]     </Transform>
[m05]     </Transforms>
[m06]     <DigestMethod Algorithm="http://www.w3.org/2001/04/xmlenc#sha256"/>
[m07]     <DigestValue>dGhpcyBpcyBub3QgYSBzaWduYXR1cmUK...=</DigestValue>
[m08]   </Reference>
```

B.2 Compatibility Mode Processing

B.2.1 Reference Generation in "Compatibility Mode"

For each data object being signed:

1. Apply the `Transforms`, as determined by the application, to the data object.
2. Calculate the digest value over the resulting data object.
3. Create a `Reference` element, including the (optional) identification of the data object, any (optional) transform elements, the digest algorithm and the `DigestValue`. (Note, it is the canonical form of these references that are signed in 3.1.3 and validated in 3.2.1 .)

The Reference Processing Model (section B.4.1 The "Compatibility Mode" Reference Processing Model) requires use of Canonical XML 1.0 [XML-C14N] as default processing behavior when a transformation is expecting an octet-stream, but the data object resulting from URI dereferencing or from the previous transformation in the list of `Transform` elements is a node-set. We RECOMMEND that, when generating signatures, signature applications do not rely on this default behavior, but explicitly identify the transformation that is applied to perform this mapping. In cases in which inclusive canonicalization is desired, we RECOMMEND that Canonical XML 1.1 [XML-C14N11] be used.

B.2.2 Reference check in "Compatibility Mode"

It is very important to check that the `Reference` actually includes the data that is expected to be signed. The [XMLDSIG-BESTPRACTICES] document describes a number of attacks, where what is apparently being signed is not actually signed.

One way to check the reference is to allow only certain combinations of transforms. For example [SAML2-CORE] and [EBXML-MSG] follow this approach.

Another option is for XML Signature libraries to return the pre-digest data to the application, so that application can inspect it to verify what is actually signed. This too may not be enough, for example in a Web Services scenario, if the reference is pointing to a soap:Body, it is not sufficient to just check the name of the "soap:Body" element, as it can lead to wrapping attacks [MCINTOSH-WRAP];Instead the application should check if this soap:Body is in the correct position, i.e. as a child of the top level soap:Envelope.

B.2.3 Signature Validation in "Compatibility Mode"

1. Obtain the keying information from `KeyInfo` or from an external source.
2. Obtain the canonical form of the `SignatureMethod` using the `CanonicalizationMethod` and use the result (and previously obtained `KeyInfo`) to confirm the `SignatureValue` over the `SignedInfo` element.

Note, `KeyInfo` (or some transformed version thereof) may be signed via a `Reference` element. Transformation and validation of this reference (3.2.1) is orthogonal to Signature Validation which uses the `KeyInfo` as parsed.

Additionally, the `SignatureMethod` URI may have been altered by the canonicalization of `SignedInfo` (e.g., absolutization of relative URIs) and it is the canonical form that MUST be used. However, the required canonicalization [XML-C14N] of this specification does not change URIs.

B.2.4 Reference Validation in "Compatibility Mode"

1. Canonicalize the `SignedInfo` element based on the `CanonicalizationMethod` in `SignedInfo`.
2. For each `Reference` in `SignedInfo`:
    1. Obtain the data object to be digested. (For example, the signature application may dereference the `URI` and execute `Transforms` provided by the signer in the `Reference` element, or it may obtain the content through other means such as a local cache.)
    2. Digest the resulting data object using the `DigestMethod` specified in its `Reference` specification.
    3. Compare the generated digest value against `DigestValue` in the `SignedInfo Reference`; if there is any mismatch, validation fails.

Note, `SignedInfo` is canonicalized in step 1. The application must ensure that the CanonicalizationMethod has no dangerous side effects, such as rewriting URIs, (see `CanonicalizationMethod` Note in section B.3 Use of CanonicalizationMethod in "Compatibility Mode") and that it "Sees What is Signed", which is the canonical form (see section 12.1.3 "See" What is Signed).

Note, After a `Signature` element has been created during Signature Generation for a signature with a same document reference, an implementation can serialize the XML content with variations in that serialization. This means that Reference Validation needs to canonicalize the XML document before digesting in step 1 to avoid issues related to variations in serialization.

B.3 Use of `CanonicalizationMethod` in "Compatibility Mode"

Alternatives to the REQUIRED section B.6 "Compatibility Mode" Canonicalization Algorithms, such as section B.6.1 Canonical XML 1.0 or a minimal canonicalization (such as CRLF and charset normalization), may be explicitly specified but are NOT REQUIRED. Consequently, their use may not interoperate with other applications that do not support the specified algorithm (see section 11. XML Canonicalization and Syntax Constraint Considerations. Security issues may also arise in the treatment of entity processing and comments if non-XML aware canonicalization algorithms are not properly constrained (see section 12.1.2 Only What is "Seen" Should be Signed).

The way in which the `SignedInfo` element is presented to the canonicalization method is dependent on that method. The following applies to algorithms which process XML as nodes or characters:

- XML based canonicalization implementations MUST be provided with an [XPATH] node-set originally formed from the document containing the `SignedInfo` and currently indicating the `SignedInfo`, its descendants, and the attribute and namespace nodes of `SignedInfo` and its descendant elements.
- Text based canonicalization algorithms (such as CRLF and charset normalization) should be provided with the UTF-8 octets that represent the well-formed `SignedInfo` element, from the first character to the last character of the XML representation, inclusive. This includes the entire text of the start and end tags of the `SignedInfo` element as well as all descendant markup and character data (i.e., the text) between those tags. Use of text based canonicalization of `SignedInfo` is NOT RECOMMENDED.

We recommend applications that implement a text-based instead of XML-based canonicalization -- such as resource constrained apps -- generate canonicalized XML as their output serialization so as to mitigate interoperability and security concerns. For instance, such an implementation SHOULD (at least) generate standalone XML instances [XML10].

NOTE: The signature application must exercise great care in accepting and executing an arbitrary `CanonicalizationMethod`. For example, the canonicalization method could rewrite the URIs of the `Reference`s being validated. Or, the method could significantly transform `SignedInfo` so that validation would always succeed (i.e., converting it to a trivial signature with a known key over trivial data). Since `CanonicalizationMethod` is inside `SignedInfo`, in the resulting canonical form it could erase itself from `SignedInfo` or modify the `SignedInfo` element so that it appears that a different canonicalization function was used! Thus a `Signature` which appears to authenticate the desired data with the desired key, `DigestMethod`, and `SignatureMethod`, can be meaningless if a capricious `CanonicalizationMethod` is used.

B.4 The `URI` Attribute in "Compatibility Mode"

If the `URI` attribute is omitted for a "Compatibility Mode" signature, then the receiving application is expected to know the identity of the object. For example, a lightweight data protocol might omit this attribute given the identity of the object is part of the application context.

In "Compatibility Mode", at most one `Reference` element without a `URI` attribute may be present in any particular `SignedInfo`, or `Manifest`.

The `URI` attribute identifies a data object using a URI-Reference [URI].

The mapping from this attribute's value to a URI reference MUST be performed as specified in section 3.2.17 of [XMLSCHEMA-2]. Additionally: Some existing implementations are known to verify the value of the URI attribute against the grammar in [URI]. It is therefore safest to perform any necessary escaping while generating the URI attribute.

We RECOMMEND XML Signature applications be able to dereference URIs in the HTTP scheme. Dereferencing a URI in the HTTP scheme MUST comply with the Status Code Definitions of [HTTP11] (e.g., 302, 305 and 307 redirects are followed to obtain the entity-body of a 200 status code response). Applications should also be cognizant of the fact that protocol parameter and state information, (such as HTTP cookies, HTML device profiles or content negotiation), may affect the content yielded by dereferencing a URI.

If a resource is identified by more than one URI, the most specific should be used (e.g. http://www.w3.org/2000/06/interop-pressrelease.html.en instead of http://www.w3.org/2000/06/interop-pressrelease). (See section 4.3 Core Validation for further information on reference processing.)

The optional Type attribute contains information about the type of object being signed after all `ds:Reference` transforms have been applied. This is represented as a URI. For example:

```
Type="http://www.w3.org/2000/09/xmldsig#Object"
Type="http://www.w3.org/2000/09/xmldsig#Manifest"
```

The `Type` attribute applies to the item being pointed at, not its contents. For example, a reference that results in the digesting of an `Object` element containing a `SignatureProperties` element is still of type `#Object`. The `Type` attribute is advisory. No validation of the type information is required by this specification.

B.4.1 The "Compatibility Mode" Reference Processing Model

Note: XPath is RECOMMENDED. Signature applications need not conform to [XPATH] specification in order to conform to this specification. However, the XPath data model, definitions (e.g., node-sets) and syntax are used within this document in order to describe functionality for those that want to process XML-as-XML (instead of octets) as part of signature generation. For those that want to use these features, a conformant [XPATH] implementation is one way to implement these features, but it is not required. Such applications could use a sufficiently functional replacement to a node-set and implement only those XPath expression behaviors REQUIRED by this specification. However, for simplicity we generally will use XPath terminology without including this qualification on every point. Requirements over "XPath node-sets" can include a node-set functional equivalent. Requirements over XPath processing can include application behaviors that are equivalent to the corresponding XPath behavior.

The data-type of the result of URI dereferencing or subsequent Transforms is either an octet stream or an XPath node-set.

The `Transforms` specified in this document are defined with respect to the input they require. The following is the default signature application behavior:

- If the data object is an octet stream and the next transform requires a node-set, the signature application MUST attempt to parse the octets yielding the required node-set via [XML10] well-formed processing.
- If the data object is a node-set and the next transform requires octets, the signature application MUST attempt to convert the node-set to an octet stream using Canonical XML [XML-C14N].

Users may specify alternative transforms that override these defaults in transitions between transforms that expect different inputs. The final octet stream contains the data octets being secured. The digest algorithm specified by `DigestMethod` is then applied to these data octets, resulting in the `DigestValue`.

Note: The section B.2.1 Reference Generation in "Compatibility Mode" includes further restrictions on the reliance upon defined default transformations when applications generate signatures.

In this specification, a 'same-document' reference is defined as a URI-Reference that consists of a hash sign ('#') followed by a fragment or alternatively consists of an empty URI [URI].

Unless the URI-Reference is such a 'same-document' reference, the result of dereferencing the URI-Reference MUST be an octet stream. In particular, an XML document identified by URI is not parsed by the signature application unless the URI is a same-document reference or unless a transform that requires XML parsing is applied. (See section 6.2 The Transforms Element.)

When a fragment is preceded by an absolute or relative URI in the URI-Reference, the meaning of the fragment is defined by the resource's MIME type [RFC2045]. Even for XML documents, URI dereferencing (including the fragment processing) might be done for the signature application by a proxy. Therefore, reference validation might fail if fragment processing is not performed in a standard way (as defined in the following section for same-document references). Consequently, we RECOMMEND in this case that the `URI` attribute not include fragment identifiers and that such processing be specified as an additional XPath Transform or XPath Filter 2 Transform [XMLDSIG-XPATH-FILTER2].

When a fragment is not preceded by a URI in the URI-Reference, XML Signature applications MUST support the null URI and shortname XPointer [XPTR-FRAMEWORK]. We RECOMMEND support for the same-document XPointers '`#xpointer(/)`' and '`#xpointer(id('ID'))`' if the application also intends to support any canonicalization that preserves comments. (Otherwise `URI="#foo"` will automatically remove comments before the canonicalization can even be invoked due to the processing defined in section B.4.2 "Compatibility Mode" Same-Document URI-References.) All other support for XPointers is OPTIONAL, especially all support for shortname and other XPointers in external resources since the application may not have control over how the fragment is generated (leading to interoperability problems and validation failures).

'`#xpointer(/)`' MUST be interpreted to identify the root node [XPATH] of the document that contains the `URI` attribute.

'#xpointer(id('ID'))' MUST be interpreted to identify the element node identified by '#element(ID)' [XPTR-ELEMENT] when evaluated with respect to the document that contains the URI attribute.

The original edition of this specification [XMLDSIG-CORE] referenced the XPointer Candidate Recommendation [XPTR-XPOINTER-CR2001] and some implementations support it optionally. That Candidate Recommendation has been superseded by the [XPTR-FRAMEWORK], [XPTR-XMLNS] and [XPTR-ELEMENT] Recommendations, and -- at the time of this edition -- the [XPTR-XPOINTER] Working Draft. Therefore, the use of the xpointer() scheme [XPTR-XPOINTER] beyond the usage discussed in this section is discouraged.

The following examples demonstrate what the URI attribute identifies and how it is dereferenced:

URI="http://example.com/bar.xml"
: Identifies the octets that represent the external resource 'http://example.com/bar.xml', that is probably an XML document given its file extension.

URI="http://example.com/bar.xml#chapter1"
: Identifies the element with ID attribute value 'chapter1' of the external XML resource 'http://example.com/bar.xml', provided as an octet stream. Again, for the sake of interoperability, the element identified as 'chapter1' should be obtained using an XPath transform rather than a URI fragment (shortname XPointer resolution in external resources is not REQUIRED in this specification).

URI=""
: Identifies the node-set (minus any comment nodes) of the XML resource containing the signature

URI="#chapter1"
: Identifies a node-set containing the element with ID attribute value 'chapter1' of the XML resource containing the signature. XML Signature (and its applications) modify this node-set to include the element plus all descendants including namespaces and attributes -- but not comments.

B.4.2 "Compatibility Mode" Same-Document URI-References

Dereferencing a same-document reference MUST result in an XPath node-set suitable for use by Canonical XML [XML-C14N]. Specifically, dereferencing a null URI (URI="") MUST result in an XPath node-set that includes every non-comment node of the XML document containing the URI attribute. In a fragment URI, the characters after the number sign ('#') character conform to the XPointer syntax [XPTR-FRAMEWORK]. When processing an XPointer, the application MUST behave as if the XPointer was evaluated with respect to the XML document containing the URI attribute . The application MUST behave as if the result of XPointer processing [XPTR-FRAMEWORK] were a node-set derived from the resultant subresource as follows:

1. include XPath nodes having full or partial content within the subresource
2. replace the root node with its children (if it is in the node-set)
3. replace any element node **E** with **E** plus all descendants of **E** (text, comment, PI, element) and all namespace and attribute nodes of **E** and its descendant elements.
4. if the URI has no fragment identifier or the fragment identifier is a shortname XPointer, then delete all comment nodes

The second to last replacement is necessary because XPointer typically indicates a subtree of an XML document's parse tree using just the element node at the root of the subtree, whereas Canonical XML treats a node-set as a set of nodes in which absence of descendant nodes results in absence of their representative text from the canonical form.

The last step is performed for null URIs and shortname XPointers. It is necessary because when [XML-C14N] or [XML-C14N11] is passed a node-set, it processes the node-set as is: with or without comments. Only when it is called with an octet stream does it invoke its own XPath expressions (default or without comments). Therefore to retain the default behavior of stripping comments when passed a node-set, they are removed in the last step if the URI is not a scheme-based XPointer. To retain comments while selecting an element by an identifier *ID*, use the following scheme-based XPointer: URI='#xpointer(id('ID'))'. To retain comments while selecting the entire document, use the following scheme-based XPointer: URI='#xpointer(/)'.

The interpretation of these XPointers is defined in section B.4.1 The "Compatibility Mode" Reference Processing Model.

B.5 "Compatibility Mode" Transforms and Processing Model

If the optional Transforms element is present and contains exactly one Transform element with an Algorithm of "http://www.w3.org/2010/xmldsig2#transform" then 2.0 processing is performed as described in section 6.2 The Transforms Element otherwise compatibility mode transform processing is performed as described here.

The optional Transforms element contains an ordered list of Transform elements; these describe how the signer obtained the data object that was digested. Each Transform consists of an Algorithm attribute and content parameters, if any, appropriate for the given algorithm. The Algorithm attribute value specifies the name of the algorithm to be performed, and the Transform content provides additional data to govern the algorithm's processing of the transform input.

The Transforms element is optional and its presence indicates that the signer is not signing the native (original) document but the resulting (transformed) document. (See section 12.1.1 Only What is Signed is Secure).

The output of each Transform serves as input to the next Transform. The input to the first Transform is the result of dereferencing the URI attribute of the Reference element. The output from the last Transform is the input for the DigestMethod algorithm.

As described in section B.4.1 The "Compatibility Mode" Reference Processing Model, some transforms take an XPath node-set as input, while others require an octet stream. If the actual input matches the input needs of the transform, then the transform operates on the unaltered input. If the transform input requirement differs from the format of the actual input, then the input must be converted.

Some Transforms may require explicit MIME type, charset (IANA registered "character set"), or other such information concerning the data they are receiving from an earlier Transform or the source data, although no Transform algorithm specified in this document needs such explicit information. Such data characteristics are provided as parameters to the Transform algorithm and should be described in the specification for the algorithm.

Examples of transforms include but are not limited to base64 decoding [RFC2045], canonicalization [XML-C14N], XPath filtering [XPATH], and XSLT [XSLT]. The generic definition of the Transform element also allows application-specific transform algorithms. For example, the transform could be a decompression routine given by a Java class appearing as a base64 encoded parameter to a Java Transform algorithm. However, applications should refrain from using application-specific transforms if they wish their signatures to be verifiable outside of their application domain. section B.7 "Compatibility Mode" Transform Algorithms defines the list of standard "Compatibility Mode" transformations.

B.6 "Compatibility Mode" Canonicalization Algorithms

If canonicalization is performed over octets, the canonicalization algorithms take two implicit parameters: the content and its charset. The charset is derived according to the rules of the transport protocols and media types (e.g, [XML-MEDIA-TYPES] defines the media types for XML). This information is necessary to correctly sign and verify documents and often requires careful server side configuration.

Various canonicalization algorithms require conversion to [UTF-8]. The algorithms below understand at least [UTF-8] and [UTF-16] as input encodings. We RECOMMEND that externally specified algorithms do the same. Knowledge of other encodings is OPTIONAL.

Various canonicalization algorithms transcode from a non-Unicode encoding to Unicode. The output of these algorithms will be in NFC [NFC]. This is because the XML processor used to prepare the XPath data model input is required (by the Data Model) to use Normalization Form C when converting an XML document to the UCS character domain from any encoding that is not UCS-based.

We RECOMMEND that externally specified canonicalization algorithms do the same. (Note, there can be ambiguities in converting existing charsets to Unicode, for an example see the XML Japanese Profile Note [XML-Japanese].)

This specification REQUIRES implementation of Canonical XML 1.0 [XML-C14N], Canonical XML 1.1 [XML-C14N11]] and Exclusive XML Canonicalization [XML-EXC-C14N]. We RECOMMEND that applications that generate signatures choose Canonical XML 1.1 [XML-C14N11] when inclusive canonicalization is desired.

Note: Canonical XML 1.0 [XML-C14N] and Canonical XML 1.1 [XML-C14N11] specify a standard serialization of XML that, when applied to a subdocument, includes the subdocument's ancestor context including all of the namespace declarations and some attributes in the 'xml:' namespace. However, some applications require a method which, to the extent practical, excludes unused ancestor context from a canonicalized subdocument. The Exclusive XML Canonicalization Recommendation [XML-EXC-C14N] may be used to address requirements resulting from scenarios where a subdocument is moved between contexts.

B.6.1 Canonical XML 1.0

Identifier for REQUIRED Canonical XML 1.0 (omits comments):
http://www.w3.org/TR/2001/REC-xml-c14n-20010315
Identifier for Canonical XML 1.0 with Comments:
http://www.w3.org/TR/2001/REC-xml-c14n-20010315#WithComments
Input:
octet-stream, node-set
Output:
octet-stream

An example of an XML canonicalization element is:

EXAMPLE 26

```
<CanonicalizationMethod Algorithm="http://www.w3.org/TR/2001/REC-xml-c14n-20010315"/>
```

The normative specification of Canonical XML1.0 is [XML-C14N]. The algorithm is capable of taking as input either an octet stream or an XPath node-set (or sufficiently functional alternative). The algorithm produces an octet stream as output. Canonical XML is easily parameterized (via an additional URI) to omit or retain comments.

B.6.2 Canonical XML 1.1

Identifier for REQUIRED Canonical XML 1.1 (omits comments):
http://www.w3.org/2006/12/xml-c14n11
Identifier for Canonical XML 1.1 with Comments:
http://www.w3.org/2006/12/xml-c14n11#WithComments
Input:
octet-stream, node-set
Output:
octet-stream

The normative specification of Canonical XML 1.1 is [XML-C14N11]. The algorithm is capable of taking as input either an octet stream or an XPath node-set (or sufficiently functional alternative). The algorithm produces an octet stream as output. Canonical XML 1.1 is easily parameterized (via an additional URI) to omit or retain comments.

B.6.3 Exclusive XML Canonicalization 1.0

Identifier for REQUIRED Exclusive XML Canonicalization 1.0 (omits comments):
http://www.w3.org/2001/10/xml-exc-c14n#
Identifier for Exclusive XML Canonicalization 1.0 with Comments:
http://www.w3.org/2001/10/xml-exc-c14n#WithComments
Input:
octet-stream, node-set
Output:
octet-stream

The normative specification of Exclusive XML Canonicalization 1.0 is [XML-EXC-C14N]].

B.7 "Compatibility Mode" `Transform` Algorithms

A `Transform` algorithm has a single implicit parameter: an octet stream from the `Reference` or the output of an earlier `Transform`.

For implementation requirements, please see section 3.3 Compatibility Mode Conformance. Application developers are strongly encouraged to support all transforms that are listed as RECOMMENDED unless the application environment has resource constraints that would make such support impractical. Compliance with this recommendation will maximize application interoperability and libraries should be available to enable support of these transforms in applications without extensive development.

B.7.1 Canonicalization

Any canonicalization algorithm that can be used for `CanonicalizationMethod` (such as those in section B.6 "Compatibility Mode" Canonicalization Algorithms) can be used as a `Transform`.

B.7.2 Base64

Identifiers:
http://www.w3.org/2000/09/xmldsig#base64
Input:
octet-stream, node-set
Output:
octet-stream

The normative specification for base64 decoding transforms is [RFC2045]. The base64 `Transform` element has no content. The input is decoded by the algorithms. This transform is useful if an application needs to sign the raw data associated with the encoded content of an element.

This transform accepts either an octet-stream or a node-set as input. If an octet-string is given as input, then this octet-stream is processed directly. If an XPath node-set (or sufficiently functional alternative) is given as input, then it is converted to an octet stream by performing operations logically equivalent to 1) applying an XPath transform with expression `self::text()`, then 2) sorting the nodeset by document order, then concatenating the string-value of each of the nodes into one long string. Thus, if an XML element is identified by a shortname XPointer in the `Reference` URI, and its content consists solely of base64 encoded character data, then this transform automatically strips away the start and end tags of the identified element and any of its descendant elements as well as any descendant comments and processing instructions. The output of this transform is an octet stream.

B.7.3 XPath Filtering

Identifier:
http://www.w3.org/TR/1999/REC-xpath-19991116
Input:
octet-stream, node-set
Output:
node-set

The normative specification for XPath expression evaluation is [XPATH]. The XPath expression to be evaluated appears as the character content of a transform parameter child element named `XPath`.

The input required by this transform is an XPath node-set or an octet-stream. Note that if the actual input is an XPath node-set resulting from a null URI or shortname XPointer dereference, then comment nodes will have been omitted. If the actual input is an octet stream, then the application MUST convert the octet stream to an XPath node-set suitable for use by Canonical XML with Comments. (A subsequent application of the REQUIRED Canonical XML algorithm would strip away these comments.) In other words, the input node-set should be equivalent to the one that would be created by the following process:

1. Initialize an XPath evaluation context by setting the initial node equal to the input XML document's root node, and set the context position and size to 1.
2. Evaluate the XPath expression `(//. | //@* | //namespace::*)`

The evaluation of this expression includes all of the document's nodes (including comments) in the node-set representing the octet stream.

The transform output is always an XPath node-set. The XPath expression appearing in the `XPath` parameter is evaluated once for each node in the input node-set. The result is converted to a boolean. If the boolean is true, then the node is included in the output node-set. If the boolean is false, then the node is omitted from the output node-set.

Note: Even if the input node-set has had comments removed, the comment nodes still exist in the underlying parse tree and can separate text nodes. For example, the markup `<e>Hello, <!-- comment -->world!</e>` contains two text nodes. Therefore, the expression `self::text()[string()="Hello, world!"]` would fail. Should this problem arise in the application, it can be solved by either canonicalizing the document before the XPath transform to physically remove the comments or by matching the node based on the parent element's string value (e.g. by using the expression `self::text()[string(parent::e)="Hello, world!"]`).

The primary purpose of this transform is to ensure that only specifically defined changes to the input XML document are permitted after the signature is affixed. This is done by omitting precisely those nodes that are allowed to change once the signature is affixed, and including all other input nodes in the output. It is the responsibility of the XPath expression author to include all nodes whose change could affect the interpretation of the transform output in the application context.

Note that the XML-Signature XPath Filter 2.0 Recommendation [XMLDSIG-XPATH-FILTER2] may be used for this purpose. That recommendation defines an XPath transform that permits the easy specification of subtree selection and omission that can be efficiently implemented.

An important scenario would be a document requiring two enveloped signatures. Each signature must omit itself from its own digest calculations, but it is also necessary to exclude the second signature element from the digest calculations of the first signature so that adding the second signature does not break the first signature.

The XPath transform establishes the following evaluation context for each node of the input node-set:

- A **context node** equal to a node of the input node-set.
- A **context position**, initialized to 1.
- A **context size**, initialized to 1.
- A **library of functions** equal to the function set defined in [XPATH] augmented with a function named **here** to be treated as if part of the library (and not namespace prefixed).
- A set of variable bindings. No means for initializing these is defined. Thus, the set of variable bindings used when evaluating the XPath expression is empty, and use of a variable reference in the XPath expression results in an error.
- The set of namespace declarations in scope for the XPath expression.

As a result of the context node setting, the XPath expressions appearing in this transform will be quite similar to those used in used in [XSLT], except that the size and position are always 1 to reflect the fact that the transform is automatically visiting every node (in XSLT, one recursively calls the command `apply-templates` to visit the nodes of the input tree).

The function `here()` is defined as follows:

Function: *node-set* **here**()

The **here** function returns a node-set containing the attribute or processing instruction node or the parent element of the text node that directly bears the

XPath expression. This expression results in an error if the containing XPath expression does not appear in the same XML document against which the XPath expression is being evaluated.

As an example, consider creating an enveloped signature (a `Signature` element that is a descendant of an element being signed). Although the signed content should not be changed after signing, the elements within the `Signature` element are changing (e.g. the digest value must be put inside the `DigestValue` and the `SignatureValue` must be subsequently calculated). One way to prevent these changes from invalidating the digest value in `DigestValue` is to add an XPath `Transform` that omits all `Signature` elements and their descendants. For example,

EXAMPLE 27

```
<Document>
  ...
  <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      ...
      <Reference URI="">
        <Transforms>
          <Transform Algorithm="http://www.w3.org/TR/1999/REC-xpath-19991116">
            <XPath xmlns:dsig="&dsig;">
              not(ancestor-or-self::dsig:Signature)
            </XPath>
          </Transform>
        </Transforms>
        <DigestMethod Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
        <DigestValue></DigestValue>
      </Reference>
    </SignedInfo>
    <SignatureValue></SignatureValue>
  </Signature>
  ...
</Document>
```

Due to the null `Reference` URI in this example, the XPath transform input node-set contains all nodes in the entire parse tree starting at the root node (except the comment nodes). For each node in this node-set, the node is included in the output node-set except if the node or one of its ancestors has a tag of `Signature` that is in the namespace given by the replacement text for the entity `&dsig;`.

A more elegant solution uses the **here** function to omit only the `Signature` containing the XPath Transform, thus allowing enveloped signatures to sign other signatures. In the example above, use the `XPath` element:

EXAMPLE 28

```
<XPath xmlns:dsig="&dsig;">
count(ancestor-or-self::dsig:Signature |
here()/ancestor::dsig:Signature[1]) >
count(ancestor-or-self::dsig:Signature)</XPath>
```

Since the XPath equality operator converts node sets to string values before comparison, we must instead use the XPath union operator (|). For each node of the document, the predicate expression is true if and only if the node-set containing the node and its `Signature` element ancestors does not include the enveloped `Signature` element containing the XPath expression (the union does not produce a larger set if the enveloped `Signature` element is in the node-set given by `ancestor-or-self::Signature`).

B.7.4 Signature Transform

Identifier:
http://www.w3.org/2000/09/xmldsig#enveloped-signature
Input:
node-set
Output:
node-set

An enveloped signature transform ***T*** removes the whole `Signature` element containing ***T*** from the digest calculation of the `Reference` element containing ***T***. The entire string of characters used by an XML processor to match the `Signature` with the XML production `element` is removed. The output of the transform is equivalent to the output that would result from replacing ***T*** with an XPath transform containing the following `XPath` parameter element:

```
<XPath xmlns:dsig="&dsig;">
count(ancestor-or-self::dsig:Signature |
here()/ancestor::dsig:Signature[1]) >
count(ancestor-or-self::dsig:Signature)</XPath>
```

The input and output requirements of this transform are identical to those of the XPath transform, but may only be applied to a node-set from its parent XML document. Note that it is not necessary to use an XPath expression evaluator to create this transform. However, this transform MUST produce output in exactly the same manner as the XPath transform parameterized by the XPath expression above.

B.7.5 XSLT Transform

Identifier:
http://www.w3.org/TR/1999/REC-xslt-19991116
Input:
octet-stream
Output:
octet-stream

The normative specification for XSL Transformations is [XSLT]. Specification of a namespace-qualified stylesheet element, which MUST be the sole child of the `Transform` element, indicates that the specified style sheet should be used. Whether this instantiates in-line processing of local XSLT declarations within the resource is determined by the XSLT processing model; the ordered application of multiple stylesheet may require multiple `Transforms`. No special provision is made for the identification of a remote stylesheet at a given URI because it can be communicated via an `xsl:include` or `xsl:import`

within the `stylesheet` child of the `Transform`.

This transform requires an octet stream as input.

The output of this transform is an octet stream. The processing rules for the XSL style sheet [XSL10] or transform element are stated in the XSLT specification [XSLT].

We RECOMMEND that XSLT transform authors use an output method of `xml` for XML and HTML. As XSLT implementations do not produce consistent serializations of their output, we further RECOMMEND inserting a transform after the XSLT transform to canonicalize the output. These steps will help to ensure interoperability of the resulting signatures among applications that support the XSLT transform. Note that if the output is actually HTML, then the result of these steps is logically equivalent [XHTML10].

B.8 Namespace Context and Portable Signatures

In [XPATH] and consequently the Canonical XML data model an element has namespace nodes that correspond to those declarations within the element and its ancestors:

> "**Note:** An element ***E*** has namespace nodes that represent its namespace declarations *as well as* any namespace declarations made by its ancestors that have not been overridden in ***E***'s declarations, the default namespace if it is non-empty, and the declaration of the prefix `xml`." [XML-C14N]

When serializing a `Signature` element or signed XML data that's the child of other elements using these data models, that `Signature` element and its children may have in-scope namespaces inherited from its ancestral context. In addition, the Canonical XML and Canonical XML with Comments algorithms define special treatment for attributes in the XML namespace, which can cause them to be part of the canonicalized XML even if they were outside of the document subset. Simple inheritable attributes (i.e. attributes that have a value that requires at most a simple redeclaration such as xml:lang and xml:space) are inherited from nearest ancestor in which they are declared to the apex node of canonicalized XML unless they are already declared at that node. This may frustrate the intent of the signer to create a signature in one context which remains valid in another. For example, given a signature which is a child of `B` and a grandchild of `A`:

EXAMPLE 29

```
<A xmlns:n1="http://foo.example">
  <B xmlns:n2="http://bar.example">
    <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
      ...
      <Reference URI="#signme"/> ...
    </Signature>
    <C ID="signme" xmlns="http://baz.example" />
  </B>
</A>
```

when either the element `B` or the signed element `C` is moved into a [SOAP12-PART1] envelope for transport:

EXAMPLE 30

```
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/">
  ...
  <SOAP:Body>
    <B xmlns:n2="http://bar.example">
      <Signature xmlns="http://www.w3.org/2000/09/xmldsig#">
        ...
      </Signature>
      <C ID="signme" xmlns="http://baz.example" />
    </B>
  </SOAP:Body>
</SOAP:Envelope>
```

The canonical form of the signature in this context will contain new namespace declarations from the `SOAP:Envelope` context, invalidating the signature. Also, the canonical form will lack namespace declarations it may have originally had from element `A`'s context, also invalidating the signature. To avoid these problems, the application may:

1. Rely upon the enveloping application to properly divorce its body (the signature payload) from the context (the envelope) before the signature is validated. Or,
2. Use a canonicalization method that "repels/excludes" instead of "attracts" ancestor context. [XML-C14N] purposefully attracts such context.

C. References

Dated references below are to the latest known or appropriate edition of the referenced work. The referenced works may be subject to revision, and conformant implementations may follow, and are encouraged to investigate the appropriateness of following, some or all more recent editions or replacements of the works cited. It is in each case implementation-defined which editions are supported.

C.1 Normative references

[ECC-ALGS]
D. McGrew; K. Igoe; M. Salter. *RFC 6090: Fundamental Elliptic Curve Cryptography Algorithms*. February 2011. IETF Informational RFC. URL: http://www.rfc-editor.org/rfc/rfc6090.txt

[FIPS-180-3]
FIPS PUB 180-3 Secure Hash Standard. U.S. Department of Commerce/National Institute of Standards and Technology. URL: http://csrc.nist.gov/publications/fips/fips180-3/fips180-3_final.pdf

[FIPS-186-3]
FIPS PUB 186-3: Digital Signature Standard (DSS). June 2009. U.S. Department of Commerce/National Institute of Standards and Technology. URL: http://csrc.nist.gov/publications/fips/fips186-3/fips_186-3.pdf

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

[HTTP11]
R. Fielding et al. *Hypertext Transfer Protocol - HTTP/1.1*. June 1999. RFC 2616. URL: http://www.ietf.org/rfc/rfc2616.txt
[LDAP-DN]
K. Zeilenga. *Lightweight Directory Access Protocol : String Representation of Distinguished Names*. June 2006. IETF RFC 4514. URL: http://www.ietf.org/rfc/rfc4514.txt
[NFC]
M. Davis, Ken Whistler. *TR15, Unicode Normalization Forms.*. 17 September 2010, URL: http://www.unicode.org/reports/tr15/
[PGP]
J. Callas, L. Donnerhacke, H. Finney, D. Shaw, R. Thayer. *OpenPGP Message Format*. IETF RFC 4880. November 2007. URL: http://www.ietf.org/rfc/rfc4880.txt
[PKCS1]
J. Jonsson and B. Kaliski. *Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1.* RFC 3447 (Informational), February 2003. URL: http://www.ietf.org/rfc/rfc3447.txt
[RFC2045]
N. Freed and N. Borenstein. *Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.* November 1996. URL: http://www.ietf.org/rfc/rfc2045.txt
[RFC2119]
S. Bradner. *Key words for use in RFCs to Indicate Requirement Levels.* March 1997. Internet RFC 2119. URL: http://www.ietf.org/rfc/rfc2119.txt
[RFC3279]
W. Polk, R. Housley, L. Bassham. *Algorithmupdates and Identifiers for the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. April 2002. Internet RFC 3279. URL: http://www.ietf.org/rfc/rfc3279.txt
[RFC3406]
L. Daigle, D. van Gulik, R. Iannella, P. Faltstrom. *URN Namespace Definition Mechanisms.*. IETF RFC 3406 October 2002. URL: http://www.ietf.org/rfc/rfc3406.txt
[RFC4055]
J. Schaad, B. Kaliski, R. Housley. *Additional Algorithms and Identifiers for RSA Cryptography for use in the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*. June 2005. IETF RFC 4055. URL: http://www.ietf.org/rfc/rfc4055.txt
[RFC5280]
D. Cooper, et. al. *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* . IETF RFC 5280 May 2008. URL: http://www.ietf.org/rfc/rfc5280.txt
[RFC5480]
S. Turner, et. al. *Elliptic Curve Cryptography Subject Public Key Information.*. IETF RFC 5480 March 2009. URL: http://www.ietf.org/rfc/rfc5480.txt
[RFC6931]
D. Eastlake. *Additional XML Security Uniform Resource Identifiers (URIs)*. IETF RFC 6931. April 2013. URL: https://datatracker.ietf.org/doc/rfc6931/
[SP800-57]
Recommendation for Key Management – Part 1: General (Revision 3). SP800-57. July 2012. U.S. Department of Commerce/National Institute of Standards and Technology. URL: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf
[URI]
T. Berners-Lee; R. Fielding; L. Masinter. *Uniform Resource Identifiers (URI): generic syntax*. January 2005. RFC 3986. URL: http://www.ietf.org/rfc/rfc3986.txt
[URN]
R. Moats. *URN Syntax.* IETF RFC 2141. May 1997. URL: http://www.ietf.org/rfc/rfc2141.txt
[URN-OID]
M. Mealling. *A URN Namespace of Object Identifiers.* . IETF RFC 3061. February 2001. URL: http://www.ietf.org/rfc/rfc3061.txt
[UTF-8]
F. Yergeau. *UTF-8, a transformation format of ISO 10646*. IETF RFC 3629. November 2003. URL: http://www.ietf.org/rfc/rfc3629.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-C14N]
John Boyer. *Canonical XML Version 1.0*. 15 March 2001. W3C Recommendation. URL: http://www.w3.org/TR/2001/REC-xml-c14n-20010315
[XML-C14N11]
John Boyer; Glenn Marcy. *Canonical XML Version 1.1*. 2 May 2008. W3C Recommendation. URL: http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/
[XML-C14N20]
John Boyer; Glen Marcy; Pratik Datta; Frederick Hirsch. *Canonical XML Version 2.0*. 11 April 2013. W3C Working Group Note. URL: http://www.w3.org/TR/2013/NOTE-xml-c14n2-20130411/
[XML-EXC-C14N]
Donald E. Eastlake 3rd; Joseph Reagle; John Boyer. *Exclusive XML Canonicalization Version 1.0*. 18 July 2002. W3C Recommendation. URL: http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/
[XML-MEDIA-TYPES]
Ümit Yalçınalp; Anish Karmarkar. *Describing Media Content of Binary Data in XML*. 4 May 2005. W3C Note. URL: http://www.w3.org/TR/2005/NOTE-xml-media-types-20050504/
[XML-NAMES]
Richard Tobin et al. *Namespaces in XML 1.0 (Third Edition)*. 8 December 2009. W3C Recommendation. URL: http://www.w3.org/TR/2009/REC-xml-names-20091208/
[XML10]
C. M. Sperberg-McQueen et al. *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. 26 November 2008. W3C Recommendation. URL: http://www.w3.org/TR/2008/REC-xml-20081126/
[XMLDSIG-XPATH]
Pratik Datta; Frederick Hirsch; Meiko Jensen. *XML Signature Streaming Profile of XPath 1.0*. 11 April 2013. W3C Working Group Note. URL: http://www.w3.org/TR/2013/NOTE-xmldsig-xpath-20130411/
[XMLDSIG-XPATH-FILTER2]
Merlin Hughes; John Boyer; Joseph Reagle. *XML-Signature XPath Filter 2.0*. 8 November 2002. W3C Recommendation. URL: http://www.w3.org/TR/2002/REC-xmldsig-filter2-20021108/
[XMLENC-CORE1]
J. Reagle; D. Eastlake; F. Hirsch; T. Roessler. *XML Encryption Syntax and Processing Version 1.1*. 11 April 2013. W3C Recommendation. URL: http://www.w3.org/TR/2013/REC-xmlenc-core1-20130411/
[XMLSCHEMA-1]
Henry S. Thompson et al. *XML Schema Part 1: Structures Second Edition*. 28 October 2004. W3C Recommendation. URL: http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/
[XMLSCHEMA-2]
Paul V. Biron; Ashok Malhotra. *XML Schema Part 2: Datatypes Second Edition*. 28 October 2004. W3C Recommendation. URL:

http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/
[XPATH]
James Clark; Steven DeRose. *XML Path Language (XPath) Version 1.0*. 16 November 1999. W3C Recommendation. URL: http://www.w3.org/TR/1999/REC-xpath-19991116/
[XPTR-ELEMENT]
Norman Walsh et al. *XPointer element() Scheme*. 25 March 2003. W3C Recommendation. URL: http://www.w3.org/TR/2003/REC-xptr-element-20030325/
[XPTR-FRAMEWORK]
Paul Grosso et al. *XPointer Framework*. 25 March 2003. W3C Recommendation. URL: http://www.w3.org/TR/2003/REC-xptr-framework-20030325/
[XSL10]
Jeremy Richman et al. *Extensible Stylesheet Language (XSL) Version 1.0*. 15 October 2001. W3C Recommendation. URL: http://www.w3.org/TR/2001/REC-xsl-20011015/
[XSLT]
James Clark. *XSL Transformations (XSLT) Version 1.0*. 16 November 1999. W3C Recommendation. URL: http://www.w3.org/TR/1999/REC-xslt-19991116

C.2 Informative references

[ABA-DSIG-GUIDELINES]
Digital Signature Guidelines. 1 August 1996. Information Security Committee, American Bar Association. URL: http://www.americanbar.org/content/dam/aba/events/science_technology/2013/dsg_tutorial.authcheckdam.pdf
[CVE-2009-0217]
Common Vulnerabilities and Exposures List, CVE-2009-0217 URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0217
[DOM-LEVEL-1]
Vidur Apparao et al. *Document Object Model (DOM) Level 1*. 1 October 1998. W3C Recommendation. URL: http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/
[EBXML-MSG]
Ian Jones; Brian Gibb; David Fischer. *OASIS ebXML Message Service Specification* 1 April 2002. URL: https://www.oasis-open.org/committees/download.php/272/ebMS_v2_0.pdf
[IEEE1363]
IEEE 1363: Standard Specifications for Public Key Cryptography. August 2000. URL: http://grouper.ieee.org/groups/1363/
[MCINTOSH-WRAP]
Michael McIntosh; Paula Austel. *XML signature element wrapping attacks and countermeasures.* In Workshop on Secure Web Services, 2005
[RANDOM]
D. Eastlake, S. Crocker, J. Schiller. *Randomness Recommendations for Security.*. IETF RFC 4086. June 2005. URL: http://www.ietf.org/rfc/rfc4086.txt
[RDF-PRIMER]
Frank Manola; Eric Miller. *RDF Primer*. 10 February 2004. W3C Recommendation. URL: http://www.w3.org/TR/2004/REC-rdf-primer-20040210/
[RFC4050]
S. Blake-Wilson; G. Karlinger; T. Kobayashi; Y. Wang. *Using the Elliptic Curve Signature Algorithm (ECDSA) for XML Digital Signatures (RFC 4050)*. April 2005. RFC. URL: http://www.ietf.org/rfc/rfc4050.txt
[RFC4949]
R. Shirey. *Internet Security Glossary, Version 2.*. IETF RFC 4949. August 2007. URL: http://www.ietf.org/rfc/rfc4949.txt
[SAML2-CORE]
Scott Cantor; John Kemp; Rob Philpott; Eve Maler. *Assertions and Protocols for SAML V2.0* 15 March 2005. URL: http://docs.oasis-open.org/security/saml/v2.0/saml-core-2.0-os.pdf
[SAX]
D. Megginson, et al. *SAX: The Simple API for XML*. May 1998. URL: http://www.megginson.com/downloads/SAX/
[SHA-1-Analysis]
McDonald, C., Hawkes, P., and J. Pieprzyk. *SHA-1 collisions now 2^{52}*. EuroCrypt 2009 Rump session. URL: http://eurocrypt2009rump.cr.yp.to/837a0a8086fa6ca714249409ddfae43d.pdf
[SHA-1-Collisions]
X. Wang, Y.L. Yin, H. Yu. *Finding Collisions in the Full SHA-1*. In Shoup, V., editor, Advances in Cryptology - CRYPTO 2005, 25th Annual International Cryptology Conference, Santa Barbara, California, USA, August 14-18, 2005, Proceedings, volume 3621 of LNCS, pages 17–36. Springer, 2005. URL: http://people.csail.mit.edu/yiqun/SHA1AttackProceedingVersion.pdf (also published in http://www.springerlink.com/content/26vljj3xhc28ux5m/)
[SOAP12-PART1]
Noah Mendelsohn et al. *SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)*. 27 April 2007. W3C Recommendation. URL: http://www.w3.org/TR/2007/REC-soap12-part1-20070427/
[UTF-16]
P. Hoffman , F. Yergeau. *UTF-16, an encoding of ISO 10646.* IETF RFC 2781. February 2000. URL: http://www.ietf.org/rfc/rfc2781.txt
[WS-SECURITY11]
A. Nadalin, C. Kaler, R. Monzillo, P. Hallam-Baker. *Web Services Security: SOAP Message Security 1.1 (WS-Security 2004)*. OASIS Standard, 1 February 2006. URL: https://www.oasis-open.org/standards#wssv1.1
[XHTML10]
Steven Pemberton. *XHTML™ 1.0 The Extensible HyperText Markup Language (Second Edition)*. 1 August 2002. W3C Recommendation. URL: http://www.w3.org/TR/2002/REC-xhtml1-20020801/
[XML-Japanese]
M. Murata. *XML Japanese Profile (2nd Edition)*. March 2005. W3C Member Submission. URL: http://www.w3.org/Submission/2005/SUBM-japanese-xml-20050324/
[XMLDSIG-BESTPRACTICES]
Pratik Datta; Frederick Hirsch. *XML Signature Best Practices*. 11 April 2013. W3C Working Group Note. URL: http://www.w3.org/TR/2013/NOTE-xmldsig-bestpractices-20130411/
[XMLDSIG-CORE]
Joseph Reagle et al. *XML Signature Syntax and Processing (Second Edition)*. 10 June 2008. W3C Recommendation. URL: http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/
[XMLDSIG-REQUIREMENTS]
Joseph Reagle Jr. *XML-Signature Requirements*. 14 October 1999. W3C Working Draft. URL: http://www.w3.org/TR/1999/WD-xmldsig-requirements-19991014
[XMLSEC11-REQS]
Frederick Hirsch; Thomas Roessler. *XML Security 1.1 Requirements and Design Considerations*. 11 April 2013. W3C Working Group Note. URL: http://www.w3.org/TR/2013/NOTE-xmlsec-reqs-20130411/
[XMLSEC2-REQS]

Frederick Hirsch; Pratik Datta. *XML Security 2.0 Requirements and Design Considerations*. 11 April 2013. W3C Working Group Note. URL: http://www.w3.org/TR/2013/NOTE-xmlsec-reqs2-20130411/

[XPTR-XMLNS]
Jonathan Marsh et al. *XPointer xmlns() Scheme*. 25 March 2003. W3C Recommendation. URL: http://www.w3.org/TR/2003/REC-xptr-xmlns-20030325/

[XPTR-XPOINTER]
Ron Daniel Jr; Eve Maler; Steven DeRose. *XPointer xpointer() Scheme*. 19 December 2002. W3C Working Draft. URL: http://www.w3.org/TR/2002/WD-xptr-xpointer-20021219/

[XPTR-XPOINTER-CR2001]
Ron Daniel Jr; Eve Maler; Steven DeRose. *XPointer xpointer() Scheme*. September 2001. W3C Candidate Recommendation. URL: http://www.w3.org/TR/2001/CR-xptr-20010911/