XML Signature Syntax and Processing Version 1.1

W3C Working Draft 26 February 2009

This version:
http://www.w3.org/TR/2009/WD-xmldsig-core1-20090226/

Latest version:
http://www.w3.org/TR/xmldsig-core1/

Previous version:
http://www.w3.org/TR/2008/REC-xmldsig-core-20080610/

Latest XML Signature recommendation:
http://www.w3.org/TR/xmldsig-core/

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)
Thomas Roessler <tlr@w3.org> (2nd edition, 1.1)
Kelvin Yiu <kelviny@microsoft.com> (1.1)

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>

Contributors
See Acknowledgements

Copyright © 2009 The Internet Society & W3C® (MIT, ERCIM, Keio), All Rights Reserved. W3C liability, trademark and document use rules apply.

Abstract

This document specifies 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.

Status of this document

This section describes the status of this document at the time of its publication. Other documents may supersede this document. A list of current W3C publications and the latest revision of this technical report can be found in the W3C technical reports index at http://www.w3.org/TR/.
This is a First Public Working Draft of "XML Signature 1.1."

At the time of this publication, the most recent W3C Recommendation of XML Signature 1 is the 10 June 2008 XML Signature (Second Edition) Recommendation. A diff-marked version of this specification is available; it shows differences between the latest recommendation and this version of the specification.

Conformance-affecting changes against this previous recommendation mainly affect the set of mandatory to implement cryptographic algorithms, including Elliptic Curve DSA (and mark-up for corresponding key material), and additional hash algorithms. There is currently no consensus about the inclusion of the ECDSA algorithm as mandatory to implement, and the Working Group seeks early community input into what algorithms should be supported. Arguments for and against specific approaches are called out in an editorial note in section 6.1 Algorithm Identifiers and Implementation Requirements.

The Working Group is, in parallel to this work, developing requirements and designs for a more radically different version 2 of XML Signature.

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

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

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

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

Table of Contents

1. Introduction
   1. Editorial Conventions
   2. Design Philosophy
   3. Versions, Namespaces and Identifiers
   4. Acknowledgements
2. Signature Overview and Examples
   1. Simple Example (Signature, SignedInfo, Methods, and References)
      1. More on Reference
   2. Extended Example (Object and SignatureProperty)
   3. Extended Example (Object and Manifest)
3. Processing Rules
   1. Signature Generation
   2. Signature Validation
4. Core Signature Syntax
   1. The Signature element
2. **The SignatureValue Element**
3. **The SignedInfo Element**
   1. **The CanonicalizationMethod Element**
   2. **The SignatureMethod Element**
   3. **The Reference Element**
      1. **The URI Attribute**
      2. **The Reference Processing Model**
      3. **Same-Document URI-References**
      4. **The Transforms Element**
      5. **The DigestMethod Element**
      6. **The DigestValue Element**
   4. **The KeyInfo Element**
      1. **The KeyName Element**
      2. **The KeyValue Element**
         1. **The DSAKeyValue Element**
         2. **The RSAKeyValue Element**
         3. **The ECKeyValue Element**
            1. **Explicit Curve Parameters**
            2. **Compatibility with RFC 4050**
         3. **The RetrievalMethod Element**
      4. **The X509Data Element**
         1. **Distinguished Name Encoding Rules**
      5. **The PGPData Element**
      6. **The SPKIData Element**
      7. **The MgmtData Element**
   5. **The Object Element**
5. **Additional Signature Syntax**
   1. **The Manifest Element**
   2. **The SignatureProperties Element**
   3. **Processing Instructions**
   4. **Comments in dsig Elements**
6. **Algorithms**
   1. **Algorithm Identifiers and Implementation Requirements**
   2. **Message Digests**
   3. **Message Authentication Codes**
   4. **Signature Algorithms**
   5. **Canonicalization Algorithms**
      1. **Canonical XML 1.0**
      2. **Canonical XML 1.1**
      3. **Exclusive XML Canonicalization 1.0**
   6. **Transform Algorithms**
      1. **Canonicalization**
      2. **Base64**
      3. **XPath Filtering**
      4. **Enveloped Signature Transform**
      5. **XSLT Transform**
7. **XML Canonicalization and Syntax Constraint Considerations**
   1. **XML 1.0, Syntax Constraints, and Canonicalization**
   2. **DOM/SAX Processing and Canonicalization**
   3. **Namespace Context and Portable Signatures**
8. **Security Considerations**
   1. **Transforms**
      1. **Only What is Signed is Secure**
1.0 Introduction

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 Security Considerations (section 8).

1.1 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 Check the Security Model, section 8.3.)

This specification provides an XML Schema [XML-schema] and DTD [XML]. The schema definition is normative.

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

"they MUST only be used where it is actually required for interoperability 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-ns] is described as "REQUIRED."

1.2 Design Philosophy

The design philosophy and requirements of this specification are addressed in the XML-Signature Requirements document [XML-Signature-RD].

1.3 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:

<table>
<thead>
<tr>
<th>URI</th>
<th>namespace prefix</th>
<th>XML internal entity</th>
</tr>
</thead>
</table>

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] or Uniform Resource Identifiers [URI] defined by the relevant normative external specification.

For instance:

SignatureProperties is identified and defined by the disig: namespace
http://www.w3.org/2000/09/xmlmdsig#SignatureProperties

ECKeyValue is identified and defined by the dsig11: namespace
http://www.w3.org/2009/xmlmdsig11#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
http://www.w3.org/2001/04/xmlenc#sha256

The http://www.w3.org/2000/09/xmlmdsig# (dsig:) namespace was introduced in the first edition of this specification. This version does not coin any new elements or algorithm identifiers in that namesapce; instead, the dsig11: namespace is used.

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
1.4 Acknowledgements

The contributions of the following Working Group members to this 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 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 Kommunikations (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
Contributions for version 1.1 were received from the members of the XML Security Working Group:

| TBD. See public list of participants for now. |

2.0 Signature Overview and Examples

This section provides an overview and examples of XML digital signature syntax. The specific processing is given in Processing Rules (section 3). The formal syntax is found in Core Signature Syntax (section 4) and Additional Signature Syntax (section 5).

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):

```
<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 [XML].

2.1 Simple Example (Signature, SignedInfo, Methods, and Reference)s

The following example is a detached signature of the content of the HTML4 in XML specification.

```
  <Signature Id="MyFirstSignature" xmlns="http://www.w3.org/2000/09/xmldsig#">
    <SignedInfo>
      <CanonicalizationMethod/>
      <SignatureMethod/>
      (<Reference URI?>+
        <Transforms>)?
      <DigestMethod>
        <DigestValue>
      </Reference>+  
      </SignedInfo>
      <SignatureValue>
        (<KeyInfo>)?
        (<Object ID?>)*
      </Signature>
  </Signature>
```

http://www.w3.org/TR/xmldsig-core1/
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.

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, and all examples in this specification, are not in canonical form.

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.

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.

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.

2.1.1 More on Reference
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.)

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.

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.

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 a resources content to the signer's key.

2.2 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 [XML, RDF]. 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 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].)
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 its correctness is required by core behavior.

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

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").

### 2.3 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 References); 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 below includes a Reference that signs a Manifest found within the Object element.

```xml
[ m01 ]  ...  
[ m02 ]   <Reference URI="#MyFirstManifest"
[ m03 ]     Type="http://www.w3.org/2000/09/xmldsig#Manifest">
[ 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 ]     </DigestValue>
[   ]  ...  
[ m09 ] <Object>
[ m10 ]   <Manifest Id="MyFirstManifest">
[ m11 ]     <Reference>
[ m12 ]       ...  
[ m13 ]     </Reference>
[ m14 ]     <Reference>
[ m15 ]       ...  
[ m16 ]     </Reference>
[ m17 ]     </Manifest>
[ m18 ] </Object>
```

### 3.0 Processing Rules

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

#### 3.1 Core Generation

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

##### 3.1.1 Reference Generation

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 3.1.2 and validated in 3.2.1.)

The Reference Processing Model (section 4.3.3.2) 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.

3.1.2 Signature Generation

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.
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, [XML] or [XML-schema] 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).

3.2 Core Validation

The REQUIRED steps of core validation include (1) reference validation, the verification of the digest contained in each Reference in SignedInfo, and (2) the cryptographic signature validation of the signature calculated over SignedInfo.

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 values in reference and signature validation are 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.

3.2.1 Reference Validation

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 affects, such as rewriting URIs, (see CanonicalizationMethod (section 4.3)) and that it Sees What is Signed, which is the canonical form.

3.2.2 Signature Validation

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.

4.0 Core Signature Syntax

The general structure of an XML signature is described in Signature Overview (section 2). 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 DTDs and [XML-Schema] with the following XML preamble, declaration, and internal entity.

Schema Definition:

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE schema PUBLIC "-//W3C//DTD XMLSchema 200102//EN" "http://www.w3.org/2001/XMLSchema.dtd"
[<!ATTLIST schema xmlns: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">
```

DTD:

```xml
<!--
The following entity declarations enable external/flexible content in the Signature content model.

#PCDATA emulates schema:string; when combined with element types it emulates schema mixed="true".

%foo.ANY permits the user to include their own element types from
```
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:

```xml
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE schema
 PUBLIC "//W3C//DTD XMLSchema 200102//EN" "http://www.w3.org/2001/XMLSchema.dtd"
[
  <!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">
```

### 4.0.1 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 \(\equiv 0 \mod 8\) (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 [MIME] encoded. (The conversion from integer to octet string is equivalent to IEEE 1363's I2OSP [1363] 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 could 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:**

```xml
<simpleType name="CryptoBinary">
  <restriction base="base64Binary">
  </restriction>
</simpleType>
```
4.1 The **Signature** element

The **Signature** element is the root element of an XML Signature. Implementation MUST generate [laxly schema valid](http://www.w3.org/2000/09/xmldsig#) **Signature** elements as specified by the following schema:

```xml
<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>
```

**DTD:**

```xml
<!ELEMENT Signature (SignedInfo, SignatureValue, KeyInfo?, Object*)>
<!ATTLIST Signature
  xmlns  CDATA   #FIXED 'http://www.w3.org/2000/09/xmldsig#'
  Id      ID  #IMPLIED >
```

4.2 The **SignatureValue** Element

The **SignatureValue** element contains the actual value of the digital signature; it is always encoded using base64 [MIME]. While we identify two **SignatureMethod** algorithms, one mandatory and one optional to implement, user specified algorithms may be used as well.

```xml
<element name="SignatureValue" type="ds:SignatureValueType"/>
<complexType name="SignatureValueType">
  <simpleContent>
    <extension base="base64Binary">
      <attribute name="Id" type="ID" use="optional"/>
    </extension>
  </simpleContent>
</complexType>
```

**DTD:**

```xml
<!ELEMENT SignatureValue (#PCDATA)>
<!ATTLIST SignatureValue
  Id  ID      #IMPLIED>
```

4.3 The **SignedInfo** Element

The structure of **SignedInfo** includes the canonicalization algorithm, a signature algorithm, and one or more references. The **SignedInfo** element may contain an optional ID attribute that will allow 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:
```xml
<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>
```

DTD:
```xml
<!ELEMENT SignedInfo (CanonicalizationMethod, SignatureMethod,  Reference+)  >
<!ATTLIST SignedInfo
  Id   ID      #IMPLIED
```

4.3.1 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 Algorithm Identifiers and Implementation Requirements (section 6.1). Implementations MUST support the REQUIRED canonicalization algorithms.

Alternatives to the REQUIRED canonicalization algorithms (section 6.5), such as Canonical XML with Comments (section 6.5.1) 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 XML Canonicalization and Syntax Constraint Considerations, section 7). 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 8.2: 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 a [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 [XML].

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 References being validated. Or, the method could massively 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.

### 4.3.2 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 6.1: Algorithm Identifiers and Implementation Requirements. While there is a single identifier, that identifier may specify a format containing multiple distinct signature values.

### 4.3.3 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>
```

DTD:

```
<!ELEMENT Reference (Transforms?, DigestMethod, DigestValue)  >
<!ATTLIST Reference
  Id  ID  #IMPLIED
  URI CDATA   #IMPLIED
  Type    CDATA   #IMPLIED>
```

4.3.3.1 The URI Attribute

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 Datatypes, 2nd Edition]. 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 [HTTP] (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 the Reference Validation (section 3.2.1) for a further information on reference processing.)

If the URI attribute is omitted altogether, 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. This attribute may be omitted from at
most one Reference in any particular SignedInfo, or Manifest.

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/xmlsig#Object"
Type="http://www.w3.org/2000/09/xmlsig#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.

4.3.3.2 The 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 is 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 [XML] 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 Reference Generation Model (section 3.1.1) 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 Transforms (section 4.3.3.1).

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. 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 [XPath-Filter-2].

When a fragment is not preceded by a URI in the URI-Reference, XML Signature applications
MUST support the null URI and shortname XPointer [XPointer-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 Same-Document
URI-References (section 4.3.3.3).) 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)' [XPointer-Element] when evaluated with respect to the document that contains
the URI attribute.

The original edition of this specification [XMLDSIG-2002] referenced the XPointer Candidate
Recommendation [XPTR-2001] and some implementations support it optionally. That
Candidate Recommendation has been superseded by the [XPointer-Framework],
[XPointer-xmlns] and [XPointer-Element] Recommendations, and -- at the time of this edition --
the [XPointer-xpointer] Working Draft. Therefore, the use of the xpointer() scheme
[XPointer-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.

4.3.3.3 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 [XPointer-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 [XPointer-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 The Reference Processing Model (section 4.3.3.2).

4.3.3.4 The Transforms Element

The optional Transforms element contains an ordered list of Transform elements; these describe how the signer obtained the data object that was digested. 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. When transforms are applied the signer is not signing the native (original) document but the resulting (transformed) document. (See Only What is Signed is Secure (section 8.1).)

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. (See Algorithm Identifiers and Implementation Requirements (section 6).)

As described in The Reference Processing Model (section 4.3.3.2), 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 [MIME], 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. Transform Algorithms (section 6.6) defines the list of standard transformations.

Schema Definition:

```xml
<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>
```

DTD:

```xml
<!ELEMENT Transforms (Transform+)>

<!ELEMENT Transform (#PCDATA|XPath %Transform.ANY;)* >
<!ATTLIST Transform
  Algorithm CDATA #REQUIRED >

<!ELEMENT XPath (#PCDATA) >
```

4.3.3.5 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
Algorithm Identifiers and Implementation Requirements (section 6.1).

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 the Reference Processing Model (section 4.3.3.2). If the result of URI dereference and application of transforms is an octet stream, then no conversion occurs (comments might be present if the Canonical XML with Comments was specified in the Transforms). The digest algorithm is applied to the data octets of the resulting octet stream.

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>

DTD:
<!ELEMENT DigestMethod (#PCDATA %Method.ANY;)* >
<!ATTLIST DigestMethod Algorithm CDATA #REQUIRED >

4.3.3.6 The DigestValue Element

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

Schema Definition:
<element name="DigestValue" type="ds:DigestValueType"/>
<simpleType name="DigestValueType">
  <restriction base="base64Binary"/>
</simpleType>

DTD:
<!ELEMENT DigestValue (#PCDATA) >
<!-- base64 encoded digest value -->

4.4 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-ns] 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.

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 4.4.2) and SHOULD implement RetrievalMethod (section 4.4.3).
The schema/DTD specifications 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 DTD that mixes their own and dsig qualified elements, or 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

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.


### Schema Definition:
```xml
<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"/>
    <any processContents="lax" namespace="##other"/>
  </choice>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

### DTD:
```xml
<!ELEMENT KeyInfo (#PCDATA|KeyName|KeyValue|RetrievalMethod|X509Data|PGPData|SPKIData|MgmtData %KeyInfo.ANY;)* >
<!ATTLIST KeyInfo
  Id  ID   #IMPLIED >
```

### 4.4.1 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.)

```xml
Schema Definition:
<element name="KeyName" type="string"/>
```

**4.4.2 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 Signature Algorithms (section 6.4). The `KeyValue` element may include externally defined public keys values represented as PCDATA or element types from an external namespace.

```xml
Schema Definition:
<element name="KeyValue" type="ds:KeyValueType"/>
<complexType name="KeyValueType" mixed="true">
  <choice>
    <element ref="ds:DSAKeyValue"/>
    <element ref="ds:RSAKeyValue"/>
    <any namespace="##other" processContents="lax"/>
  </choice>
</complexType>
```

**4.4.2.1 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 [DSS]. DSA public key values can have the following fields:

- **P**
  a prime modulus meeting the [DSS] 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 is calculatable from P and Q. Parameters seed and pgenCounter are used in the DSA prime number generation algorithm specified in [DSS]. 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 [MIME] 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:

```xml
<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"/>
    <element name="Seed" type="ds:CryptoBinary"/>
    <element name="PgenCounter" type="ds:CryptoBinary"/>
  </sequence>
</complexType>
```

DTD Definition:

```xml
<!ELEMENT DSAKeyValue ((P, Q)?, G?, Y, J?, (Seed, PgenCounter)?) >
<!ELEMENT P (#PCDATA) >
<!ELEMENT Q (#PCDATA) >
<!ELEMENT G (#PCDATA) >
<!ELEMENT Y (#PCDATA) >
<!ELEMENT J (#PCDATA) >
<!ELEMENT Seed (#PCDATA) >
<!ELEMENT PgenCounter (#PCDATA) >
```

4.4.2.2 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.

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

### 4.4.2.3 The ECKeyValue Element

**Identifier**

Type: [http://www.w3.org/2009/xmldsig11#ECKeyValue](http://www.w3.org/2009/xmldsig11#ECKeyValue)

(this can be used within a `RetrievalMethod` or `Reference` element to identify the referent's type)

The Working Group has considered, but not accepted, an alternative design for ECKeyValueType. This decision might be reconsidered if new information is made available that was not previously considered, in particular implementation experience.

The ECPublicKey element is defined in the [http://www.w3.org/2009/xmldsig11#](http://www.w3.org/2009/xmldsig11#) namespace.

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

```xml
<ECKeyValue xmlns="http://www.w3.org/2009/xmldsig11#">
  <NamedCurve URI="urn:oid:1.2.840.10045.3.1.7" />
  <PublicKey>
    vWccUP6Jp3pcaMCGIcAh3vOev4gaa2ukOANC7UfgCf8KDO7AtTosGJK7/TA8IC3vZoCy9I5oPjRhyTBu1
  </PublicKey>
</ECKeyValue>
```

Domain parameters can be encoded explicitly using the ECPParameters element or by reference using the NamedCurve element. A named curve is specified through the **URN** attribute. For named curves that are identified by OIDs, such as those defined in [RFC3279], and [SEC 1], the OID should be encoded according to [RFC3061]. Conformant applications MUST support the NamedCurve element and the 256-bit prime field curve as identified by the OID 1.2.840.10045.3.1.7.

The PublicKey element contains a 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 as specified in Section 2.3.3 of [SEC1]. 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:

```xml
<!-- 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>
</complexType>
<complexType name="NamedCurveType">
  <attribute name="URI" type="anyURI"/>
</complexType>
<simpleType name="ECPointType">
  <restriction base="ds:CryptoBinary"/>
</simpleType>
```

### DTD Definition:

```
TBD
```

#### 4.4.2.3.1 Explicit Curve Parameters

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

1. The 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 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 2.3.5 of [SEC 1], then the resultant octet string is encoded in base64. Seed is an optional parameter used to derive the coefficients of a randomly generated elliptic curve and is represented as a base64 encoded bit string.
3. The Base element specifies the base point \(P\) on the elliptic curve. The base point is represented as a value of type ECPointType.
4. The Order element specifies the order \(n\) of the base point and is encoded as a positiveInteger.
5. The Cofactor element is an optional element that specifies the integer \(h = \#E(F_q)/n\). 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 Hash element is an optional element that specifies the hash algorithm used to generate the elliptic curve E and/or base point G verifiably at random. If the Hash element is present then the optional Seed element in the Curve element must also be present.

The markup for the curve validation data (Seed and Hash) is under active discussion within the Working Group.

Schema Definition:

```xml
<!-- 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="Hash" type="anyURI" 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"/>
    <element name="Seed" type="ds:CryptoBinary" minOccurs="0"/>
  </sequence>
</complexType>
```

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

Schema Definition:

```xml
<!-- targetNamespace="http://www.w3.org/2009/xmldsig11#" -->
<element name="Prime" type="dsig11:PrimeFieldParamsType"/>
<complexType name="PrimeFieldParamsType">
  <sequence>
    <element name="PrimeFieldParamsType">
      <sequence>
        <element name="P" type="ds:CryptoBinary"/>
      </sequence>
    </element>
  </sequence>
</complexType>
```

DTD Definition:

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

### 4.4.2.3.2 Compatibility with RFC 4050

Implementations that need to support the [RFC4050](https://tools.ietf.org/html/rfc4050) format for ECDSA keys can avoid known interoperability problems with that specification by adhering to the following profile:

1. Avoid validating the ECDSAKKeyValue element against the [RFC4050](https://tools.ietf.org/html/rfc4050) schema. XML schema validators may not support integer types with decimal data exceeding 18 decimal digits. [XML-schema](https://www.w3.org/TR/xmlschema-1/).
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.

<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="58511065380174439324917904648233320204931884267326155134056258624" />
    <Y Value="102403352168277752409102672177795083590286425248815408780791198957" />
  </PublicKey>
</ECDSAKeyValue>

4.4.3 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 Reference’s URI (section 4.3.3.1) and The Reference Processing Model (section 4.3.3.2) except that there is no DigestMethod or DigestValue child elements and presence of the URI 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 4.4) 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.

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>

DTD

<!ELEMENT RetrievalMethod (Transforms?) >
<!ATTLIST RetrievalMethod
  URI CDATA #REQUIRED
  Type CDATA #IMPLIED >

Note: The schema for the URI attribute of RetrievalMethod erroneously omitted the attribute: 
use="required"

The DTD is correct. 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.

4.4.4 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:

1. 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 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
   - Elements from an external namespace which accompanies/complements any of the elements above.
   - The X509CRL element, which contains a base64-encoded certificate revocation list (CRL) [X509v3].
   - The dsig11:OCSPResponse element contains a base64-encoded OCSP response in DER encoding. [OCSP].

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

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:
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. Such deployments should avoid schema-validating the X509IssuerSerial element. XML Schema validators may not support integer types with decimal data exceeding 18 decimal digits [XML-schema].

4.4.4.1 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 a 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.
<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"/>
      <any namespace="##other" processContents="lax"/>
    </choice>
  </sequence>
</complexType>
<complexType name="X509IssuerSerialType">
  <sequence>
    <element name="X509IssuerName" type="string"/>
    <element name="X509SerialNumber" type="integer"/>
  </sequence>
</complexType>

<!-- targetNameSpace="http://www.w3.org/2009/xmldsig11#" -->
[element name="OCSPResponse" type="base64binary" />

DTD
<!ELEMENT X509Data ((X509IssuerSerial | X509SKI | X509SubjectName | X509Certificate | X509CRL)+ %X509.ANY;)>  
<!ELEMENT X509IssuerSerial (X509IssuerName, X509SerialNumber)>
<!ELEMENT X509IssuerName (#PCDATA)>  
<!ELEMENT X509SubjectName (#PCDATA)>
<!ELEMENT X509SerialNumber (#PCDATA)>  
<!ELEMENT X509SKI (#PCDATA)>  
<!ELEMENT X509Certificate (#PCDATA)>  
<!ELEMENT X509CRL (#PCDATA)>  

<!-- Note, this DTD and schema permit X509Data to be empty; this is precluded by the text in KeyInfo Element (section 4.4) 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. -->

4.4.5 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:
4.4.6 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:**

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

**DTD:**

```xml
<!ELEMENT SPKIData (SPKISexp %SPKIData.ANY;) >
<!ELEMENT SPKISexp (#PCDATA) >
```

4.4.7 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. For example, DH key exchange, RSA key encryption, etc. Use of this element is NOT RECOMMENDED. It provides a syntactic hook where in-band key distribution or agreement data can be placed. However, superior interoperable child elements of `KeyInfo` for the transmission of encrypted keys and for key agreement are being specified by the W3C XML Encryption Working Group and they should be used instead of `MgmtData`.

**Schema Definition:**
```xml
<element name="MgmtData" type="string"/>
```

**DTD:**
```xml
<!ELEMENT MgmtData (#PCDATA)> 
```

### 4.5 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` attribute 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 [MIME]. 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 which require normative type and encoding information for signature validation should specify `Transforms` with well defined resulting types and/or encodings.

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 (easy for XML documents) or a transform must be used to remove the `Object` tags (likely where the data object is non-XML). 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:**
```xml
<element name="Object" type="ds:ObjectType"/>
<complexType name="ObjectType" mixed="true">
  <sequence minOccurs="0" maxOccurs="unbounded">
  
  </sequence>
</complexType>
```
5.0 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.

5.1 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 References. 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:**

```xml
<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>
```

**DTD:**

```xml
<!ELEMENT Manifest (Reference+) >
<!ATTLIST Manifest
  Id ID #IMPLIED >
```

5.2 The SignatureProperties Element

**Identifier**
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"/>
  </choice>
  <attribute name="Target" type="anyURI" use="required"/>
  <attribute name="Id" type="ID" use="optional"/>
</complexType>
```

**DTD:**

```
<!ELEMENT SignatureProperties (SignatureProperty+)>  
<!ATTLIST SignatureProperties
  Id ID   #IMPLIED  >

<!ELEMENT SignatureProperty (#PCDATA %SignatureProperty.ANY;)*>
<!ATTLIST SignatureProperty
  Target CDATA    #REQUIRED
  Id  ID   #IMPLIED  >
```

### 5.3 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 `CanonicalizationMethods` 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.

### 5.4 Comments in Signature Elements

XML comments are not used by this specification.

Note that unless `CanonicalizationMethod` removes comments within `SignedInfo` or any other referenced XML (which [XML-C14N](http://www.w3.org/TR/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](http://www.w3.org/TR/xml-c14n), is specified.

### 6.0 Algorithms
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.

### 6.1 Algorithm Identifiers and Implementation Requirements

There is currently no consensus on mandatory to implement algorithms; the current draft text represents one possible outcome. Positions of some Working Group members against the currently expressed set of mandatory to implement algorithms include:

- **RSA and DSA** are acceptable as a mandatory to implement signature algorithms. Given limited support in parts of the industry, elliptic curve DSA is not acceptable as a mandatory to implement algorithm, and might lead to lack of implementation of this version of the specification.
- **There should be recommended algorithms, but no mandatory to implement algorithms.** The rationale is that this gives greater flexibility to deployments. (Other WG members argued against this since it could harm interoperability not having mandatory algorithms.)

The opposing position is that, going forward, this specification needs to have credible algorithm agility for both hash and public-key algorithms: Should one set of algorithms prove weak, this would enable a quick switch-over. Therefore, there should be two mandatory to implement public-key algorithms from different families. At this time, elliptic curve based algorithms are the only credible contenders. They have the additional benefit of providing a reasonable balance between key sizes and security level. As profiles built on top of XML Signature that currently rely on DSA- SHA1 or RSA-SHA1 as the only supported signature algorithm will need to be updated in the future, the Signature core specification should outline a clear way forward in terms of choice of algorithms. This choice should be **Elliptic Curve DSA**.

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  
   [http://www.w3.org/2000/09/xmldsig#sha1](http://www.w3.org/2000/09/xmldsig#sha1)
2. SHA256
Optional
1. SHA384
   http://www.w3.org/2001/04/xmldsig-more#sha384
2. SHA512
   http://www.w3.org/2001/04/xmlenc#sha512

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

MAC
Required
1. HMAC-SHA1
   http://www.w3.org/2000/09/xmldsig#hmac-sha1

Recommended
1. HMAC-SHA256
   http://www.w3.org/2001/04/xmldsig-more#hmac-sha256

Optional
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

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

Recommended
1. RSAwithSHA1
   http://www.w3.org/2000/09/xmldsig#rsa-sha1

Optional
1. RSAwithSHA384
   http://www.w3.org/2001/04/xmldsig-more#rsa-sha384
2. RSAwithSHA512
   http://www.w3.org/2001/04/xmldsig-more#rsa-sha512
3. ECDSAwithSHA1
   http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha1
4. ECDSAwithSHA384
   http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha384
5. ECDSAwithSHA512
   http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha512
6. DSAwithSHA1
   (signature generation)
   http://www.w3.org/2000/09/xmldsig#dsa-sha1

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

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 (omits comments)
   http://www.w3.org/2001/10/xml-exc-c14n#
4. Exclusive XML Canonicalization 1.0 with Comments
   http://www.w3.org/2001/10/xml-exc-c14n#WithComments

Transform

Required
1. Enveloped Signature*
   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

* 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.

6.2 Message Digests

This specification defines several possible digest algorithms for the DigestMethod element, including REQUIRED algorithms SHA-256 and SHA-1. Use of SHA-256 is strongly recommended over SHA-1 because recent advances in cryptanalysis have cast doubt on the long-term collision resistance of SHA-1. However, SHA-1 support is REQUIRED in this specification to support interoperability with implementations of prior versions of this specification.

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.

6.2.1 SHA-1

Identifier:
   http://www.w3.org/2000/09/xmldsig#sha1
The **SHA-1** algorithm [SHA-1] takes no explicit parameters. An example of an SHA-1 DigestAlg element is:

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

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

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

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

```xml
<DigestValue>qZk+NkcGwq6PiVxeFDCbJzQ2J0=</DigestValue>
```

### 6.2.2 SHA-256

**Identifier:**

[http://www.w3.org/2001/04/xmlenc#sha256](http://www.w3.org/2001/04/xmlenc#sha256)

The **SHA-256** algorithm [SHA-256] 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.

### 6.2.3 SHA-384

**Identifier:**

[http://www.w3.org/2000/09/xmldsig#sha384](http://www.w3.org/2000/09/xmldsig#sha384)

The **SHA-384** algorithm [SHA-384] 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.

### 6.2.4 SHA-512

**Identifier:**

[http://www.w3.org/2001/04/xmlenc#sha512](http://www.w3.org/2001/04/xmlenc#sha512)

The **SHA-512** algorithm [SHA-512] 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.

### 6.3 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.

#### 6.3.1 HMAC
The HMAC algorithm (RFC2104 [HMAC]) takes the truncation length in bits as a parameter; if the parameter is not specified then all the bits of the hash are output. An example of an HMAC SignatureMethod element:

```xml
<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

9294727A 3638BB1C 13F48EF8 158BFC9D

from the test vectors in [HMAC] would be

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

**6.4 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.

**6.4.1 DSA**

Identifier:

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

The DSA algorithm [DSS] takes no explicit parameters. An example of a DSA SignatureMethod element is:

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

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 2437 [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:

\[
\begin{align*}
r &= 8BAC1AB6 \ 6410435C \ B7181F95 \ B16AB97C \ 92B341C0 \\
s &= 41E2345F \ 1F56DF24 \ 58F426D1 \ 55B4BA2D \ B6DCD8C8
\end{align*}
\]

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

\[
<\text{SignatureValue}>
\text{i6watmQQQ1y3GB+VsWq5fJKzQcBB4jRfH1bfJFj0JtFVtLotttzYyA==}</\text{SignatureValue}>
\]

### Security considerations regarding DSA key sizes

Implementers of XML Signature 1.1 should be aware that as of the time of publication the permitted parameter sizes for DSA are too small to be used for long-term signatures. Per FIPS 186-2 Change Notice 1 [DSS], the DSA security parameter L is defined to be exactly 1024 and the corresponding DSA prime modulus p is defined to be in the interval \(2^{1023} < p < 2^{1024}\). However, in Special Publication SP 800-57 Part 1 [SP800-57], NIST recommends using at least at 2048-bit public keys for securing information beyond 2010 (and 3072-bit keys for securing information beyond 2030). (A forthcoming revision to FIPS 186 (FIPS 186-3) will allow DSA to be used with longer prime moduli and the SHA-256/SHA-384/SHA-512 hash functions.)

Since XML Signature 1.0 required implementations to support DSA-based digital signatures, this XML Signature 1.1 revision REQUIRES signature verifiers to implement DSA in order to guarantee interoperability with XML Signature 1.0 generators. XML Signature 1.1 implementations MAY but are NOT REQUIRED to support DSA-based signature generation, and given the short key size and the SP800-57 guidelines we do not recommend use of DSA as currently limited to 1024-bit prime moduli for signatures that will be verified beyond 2010.

#### 6.4.2 RSA (PKCS#1 v1.5)

**Identifier:**

- [http://www.w3.org/2000/09/xmldsig#rsa-sha1](http://www.w3.org/2000/09/xmldsig#rsa-sha1)
- [http://www.w3.org/2001/04/xmldsig-more#rsa-sha256](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-sha384)
- [http://www.w3.org/2001/04/xmldsig-more#rsa-sha512](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 2437 [PKCS1]. The RSA algorithm takes no explicit parameters. An example of an RSA SignatureMethod element is:

\[
<\text{SignatureMethod} \text{ Algorithm}=\text{http://www.w3.org/2000/09/xmldsig#rsa-sha1} />
\]

The **SignatureValue** content for an RSA signature is the base64 [MIME] encoding of the octet string computed as per RFC 2437 [PKCS1], section 8.1.1: Signature generation for the RSASSA-PKCS1-v1_5 signature scheme]. As specified in the EMSA-PKCS1-V1_5-ENCODE function RFC 2437 [PKCS1], section 9.2.1], the value input to the signature function MUST contain a pre-pended algorithm object identifier for the hash function, but the availability of an ASN.1 parser and recognition of OIDs is not required of a signature verifier. The PKCS#1 v1.5
representation appears as:

CRYPT (PAD (ASN.1 (OID, DIGEST (data))))

Note that the padded ASN.1 will be of the following form:

01 | FF* | 00 | prefix | hash

where "|" is concatenation, "01", "FF", and "00" are fixed octets of the corresponding hexadecimal value, "hash" is the SHA1 digest of the data, and "prefix" is the ASN.1 BER SHA1 algorithm designator prefix required in PKCS1 [RFC 2437], that is,

hex 30 21 30 06 05 06 05 2B 0E 03 02 1A 05 00 04 14

This prefix is included to make it easier to use standard cryptographic libraries. The FF octet MUST be repeated the maximum number of times such that the value of the quantity being CRYPTed is one octet shorter than the RSA modulus.

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

<SIGNATUREVALUE>
IWijxQjUrCXBYoCe14QxjWo9Kg8D3p9t1WcO7t0/gvTE96639In0FZY02/rvP/+bMJ01EArmKZsR5VW3rwoPzw=<
</SIGNATUREVALUE>

6.4.3 ECDSA

Identifiers:

http://www.w3.org/2001/04/xmldsig-more#ecdsa-sha1
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 [DSS] takes no explicit parameters. An example of a ECDSA SignatureMethod element is:

<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 2437 [PKCS1] specification with the l parameter equal to the size of the output of the digest function in bytes (e.g. 32 for SHA-256).

6.5 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, RFC2376 [XML-MT] 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, NFC-Corrigendum]. This is because the XML processor used to prepare the XPath data model input is required (by 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 [XML-Japanese] Note.)

This specification REQUIRES implementation of both Canonical XML 1.0 [XML-C14N] and Canonical XML 1.1 [XML-C14N11]. 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.

6.5.1 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:

```xml
<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.

6.5.2 Canonical XML 1.1

Identifier for REQUIRED Canonical XML 1.1 (omits comments):
http://www.w3.org/2006/12/xml-c14n11
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.

### 6.5.3 Exclusive XML Canonicalization 1.0

The normative specification of Exclusive XML Canonicalization 1.0 is [XML-C14N11].

### 6.6 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 Algorithm Identifiers and Implementation Requirements. 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.

### 6.6.1 Canonicalization

Any canonicalization algorithm that can be used for CanonicalizationMethod (such as those in Canonicalization Algorithms (section 6.5)) can be used as a Transform.

### 6.6.2 Base64

**Identifiers:**

[http://www.w3.org/2000/09/xmldsig#base64](http://www.w3.org/2000/09/xmldsig#base64)

**Input:**

- octet-stream, node-set

**Output:**

- octet-stream

The normative specification for base64 decoding transforms is [MIME]. 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
\(\text{self::text()}\), then 2) taking the string-value of the node-set. 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.

6.6.3 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

\(\text{self::text()}[\text{string()="Hello, world!"}])\)

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 [XPath-Filter-2] 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] plus a function named `here`.
- 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 [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,

```xml
<Document>
  ...
</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:

```xml
<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).

### 6.6.4 Enveloped 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:
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.

6.6.5 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 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 [XHTML].

7.0 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 [XML], as described in 7.1 below. There are those related to [DOM], [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 [XML] 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, NFC-Corrigendum] 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.

### 7.1 XML 1.0, Syntax Constraints, and Canonicalization

XML 1.0 [XML] 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 [XML-schema], 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

7.2 DOM/SAX Processing and Canonicalization

In addition to the canonicalization and syntax constraints discussed above, many XML applications use the Document Object Model [DOM] 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 the XML 1.0 syntax constraints given in the previous section 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.

7.3 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 contain namespace declarations from its ancestor context. In addition, the Canonical XML and Canonical XML with Comments algorithms import all xml namespace attributes (such as xml:lang) from the 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 \( A \):

\[
\begin{align*}
&A \text{ xmlns:n1="&foo;";} \\
&B \text{ xmlns:n2="&bar;";} \\
&\text{<Signature xmlns="&dsig;"> } \ldots \\
&\text{<Reference URI="#signme"/> } \ldots \\
&\text{</Signature>} \\
&C \text{ ID="signme" xmlns="&baz;"/}
\end{align*}
\]

when either the element \( B \) or the signed element \( C \) is moved into a [SOAP] envelope for transport:

\[
\begin{align*}
&\text{<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope/"}> \\
&\text{ } \ldots \\
&\text{<SOAP:Body}> \\
&\text{<B xmlns:n2="&bar;";} \\
&\text{<Signature xmlns="&dsig;"> } \ldots \\
&\text{</Signature>} \\
&\text{<C ID="signme" xmlns="&baz;"/>}
\end{align*}
\]

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.

### 8.0 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.

#### 8.1 Transforms

A requirement of this specification is to permit signatures to "apply to a part or totality of a XML document." (See [XML-Signature-RD, 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.

### 8.1.1 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 cannot 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.

### 8.1.2 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 to 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.

### 8.1.3 "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 or NFC-Corrigendum (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.

8.2 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.

8.3 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 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.

9.0 Schema, DTD, Data Model, and Valid Examples

XML Signature Schema Instance
xmldsig-core-schema.xsd
Valid XML schema instance based on the 20001024 Schema/DTD [XML-Schema].

XML Signature 1.1 Schema Instance
xmldsig11-schema.xsd
This schema document defines the additional elements defined in this version of the XML Signature specification.

XML Signature DTD
xmldsig-core-schema.dtd

RDF Data Model
xmldsig-datamodel-20000112.gif

XML Signature Object Example
signature-example.xml
A cryptographical fabricated XML example that includes foreign content and validates under the schema, it also uses schemaLocation to aid automated schema fetching and validation.

RSA XML Signature Example
signature-example-rsa.xml
An XML Signature example with generated cryptographic values by Merlin Hughes and validated by Gregor Karlinger.

DSA XML Signature Example
signature-example-dsa.xml
Similar to above but uses DSA.

10.0 Definitions

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." [SEC]

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." [Digital Signature Guidelines, ABA].

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." [Digital Signature Guidelines, ABA] 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." [SEC]

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 [HTTP]. 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 [XML]. The term XML document is used to describe data objects which conform to the XML specification [XML].

Integrity
"The property that data has not been changed, destroyed, or lost in an unauthorized or accidental manner." [SEC] 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 Core Validation (section 3.2).

**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.

### 11.0 References

*The references section needs to be updated, and split into normative and informative references.*

**ABA**

[Digital Signature Guidelines.](http://www.abanet.org/scitech/ec/isc/dsgfree.html)

**DOM**

[Document Object Model (DOM) Level 1 Specification.](http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/)

**DSS**


**HMAC**


**HTTP**

[RFC 2616. Hypertext Transfer Protocol -- HTTP/1.1.](http://www.w3.org/TR/1998/REC-HTTP-1.1-19990604/)
KEYWORDS

http://www.ietf.org/rfc/rfc2119.txt

LDAP-DN

http://www.ietf.org/rfc/rfc4514.txt

MD5

http://www.ietf.org/rfc/rfc1321.txt

MIME

http://www.ietf.org/rfc/rfc2045.txt

NFC


NFC-Corrigendum

Normalization Corrigendum. The Unicode Consortium.
http://www.unicode.org/unicode/uni2errata/Normalization_Corrigendum.html.

OCSP

http://www.ietf.org/rfc/rfc2560.txt

PGP

http://www.ietf.org/rfc/rfc2440.txt

RANDOM

http://www.ietf.org/rfc/rfc1750.txt

RDF

http://www.w3.org/TR/2000/CR-rdf-schema-20000327/
http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/


RFC3279

http://www.ietf.org/rfc/rfc3279.txt

RFC4050

http://www.ietf.org/rfc/rfc4050.txt
RFC4051
http://www.ietf.org/rfc/rfc4051.txt

PKCS1
http://www.ietf.org/rfc/rfc2437.txt

SAX
http://www.megginson.com/downloads/SAX/

SEC
http://www.faqs.org/rfcs/rfc2828.html

SEC1

SEC2

SHA-1

SHA-256

SHA-384

SHA-512

SOAP
http://www.w3.org/TR/2000/NOTE-SOAP-20000508/

SP800-57

UTF-16
http://www.ietf.org/rfc/rfc2781.txt

UTF-8
URI
http://www.ietf.org/rfc/rfc3986.txt

URL
http://www.ietf.org/rfc/rfc1738.txt

URN
http://www.ietf.org/rfc/rfc2141.txt
http://www.ietf.org/rfc/rfc2611.txt

URN-OID
http://www.ietf.org/rfc/rfc3061.txt

X509v3

XHTML 1.0
http://www.w3.org/TR/2000/REC-xhtml1-20000126/

XLink
http://www.w3.org/TR/2001/REC-xlink-20010627/

XML
http://www.w3.org/TR/2006/REC-xml-20060816/

XML-C14N
http://www.w3.org/TR/2001/REC-xml-c14n-20010315
http://www.ietf.org/rfc/rfc3076.txt

XML-C14N11
http://www.w3.org/TR/2008/REC-xml-c14n11-20080502/

XML-exc-C14N
http://www.w3.org/TR/2002/REC-xml-exc-c14n-20020718/

XML-Enc
http://www.w3.org/TR/xmlenc-core/

XML-Japanese
XML-MT

http://www.ietf.org/rfc/rfc2376.txt

XML-nS

http://www.w3.org/TR/2006/REC-xml-names-20060816/

XML-Schema

http://www.w3.org/TR/2004/REC-xmlschema-1-20041028/
http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/

XML-Signature-RD

http://www.w3.org/TR/1999/WD-xmldsig-requirements-19991014.html
http://www.ietf.org/rfc/rfc2807.txt
http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/

XMLDSIG-2002

http://www.w3.org/TR/2002/REC-xmldsig-core-20020212/

XPath

http://www.w3.org/TR/1999/REC-xpath-19991116

XPath Filter-2

http://www.w3.org/TR/2002/REC-xmldsig-filter2-20021108/

XPTR-2001

http://www.w3.org/TR/2001/CR-xptr-20010911/

XPointer-Element

http://www.w3.org/TR/2003/REC-xptr-element-20030325/

XPointer-Framework

http://www.w3.org/TR/2003/REC-xptr-framework-20030325/

XPointer-Xpointer

http://www.w3.org/TR/2002/WD-xptr-xpointer-20021219/

XPointer-xmLns

http://www.w3.org/TR/2003/REC-xptr-xmlns-20030325/

XSL

XSLT