Network Working Group
Request for Comments: 3075
Category: Standards Track

D. Eastlake
Motorola
J. Reagle
W3C/MIT
D. Solo
Citigroup

March 2001

XML-Signature Syntax and Processing

Status of this Memo

This document specifies an Internet standards track protocol for the Internet community, and requests discussion and suggestions for improvements. Please refer to the current edition of the "Internet Official Protocol Standards" (STD 1) for the standardization state and status of this protocol. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This document specifies XML (Extensible Markup Language) 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.

Table of Contents

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 uses both XML Schemas [XML-schema] and DTDs [XML]. (Readers unfamiliar with DTD syntax may wish to refer to Ron Bourret’s "Declaring Elements and Attributes in an XML DTD" [Bourret].) The schema definition is presently 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 interoperation or to limit behavior which has potential for causing harm (e.g., limiting retransmissions)"

Consequently, we use these capitalized keywords to unambiguously specify requirements over protocol and application features and behavior that affect the interoperability and security of implementations. These key words are not used (capitalized) to describe XML grammar; schema definitions unambiguously describe such requirements and we wish to reserve the prominence of these terms for the natural language descriptions of protocols and features. For instance, an XML attribute might be described as being "optional." Compliance with the XML-namespace specification [XML-ns] is described as "REQUIRED."

1.2 Design Philosophy

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

1.3 Versions, Namespaces and Identifiers

No provision is made for an explicit version number in this syntax. If a future version is needed, it will use a different namespace The XML namespace [XML-ns] URI that MUST be used by implementations of this (dated) specification is:
xmlns="http://www.w3.org/2000/09/xmldsig#"

This namespace is also used as the prefix for algorithm identifiers used by this specification. While applications MUST support XML and XML-namespaces, the use of internal entities [XML] or our "dsig" XML namespace prefix and defaulting/scoping conventions are OPTIONAL; we use these facilities to provide compact and readable examples.

This specification uses Uniform Resource Identifiers [URI] to identify resources, algorithms, and semantics. The URI in the namespace declaration above is also used as a prefix for URIs under the control of this specification. For resources not under the control of this specification, we use the designated Uniform Resource Names [URN] or Uniform Resource Locators [URL] defined by its normative external specification. If an external specification has not allocated itself a Uniform Resource Identifier we allocate an identifier under our own namespace. For instance:

SignatureProperties is identified and defined by this specification’s
      namespace
      http://www.w3.org/2000/09/xmldsig#SignatureProperties

XSLT is identified and defined by an external URI
      http://www.w3.org/TR/1999/PR-xslt-19991008

SHA1 is identified via this specification’s namespace and defined via
      a normative reference
      http://www.w3.org/2000/09/xmldsig#sha1
      FIPS PUB 180-1. Secure Hash Standard. U.S. Department of
      Commerce/National Institute of Standards and Technology.

Finally, in order to provide for terse namespace declarations we sometimes use XML internal entities [XML] within URIs. For instance:

```
   <?xml version='1.0'?>
   <!DOCTYPE Signature SYSTEM
     "xmldsig-core-schema.dtd" [ <!ENTITY dsig
     "http://www.w3.org/2000/09/xmldsig#"> ]>
   <Signature xmlns="&dsig;" Id="MyFirstSignature">
     <SignedInfo>
     ...
```

1.4 Acknowledgements

The contributions of the following working group members to this specification are gratefully acknowledged:

* Mark Bartel, JetForm Corporation (Author)
* John Boyer, PureEdge (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
* Peter Lipp, IAIK TU Graz
* Joseph Reagle, W3C (Chair, Author/Editor)
* Ed Simon, Entrust Technologies Inc. (Author)
* David Solo, Citigroup (Author/Editor)
* Petteri Stenius, DONE Information, Ltd
* 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.

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>
        <SignedInfo>
          (CanonicalizationMethod)
          (SignatureMethod)
          (<Reference (URI=)? >
            (Transforms)?
            (DigestMethod)
            (DigestValue)
          </Reference>)+
        </SignedInfo>
        (SignatureValue)
       (KeyInfo)?
       (Object)*
      </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 resides 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 References)

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

```
[s01] <Signature Id="MyFirstSignature"
       xmlns="http://www.w3.org/2000/09/xmldsig#">
[s02]   <SignedInfo>
[s03]   <CanonicalizationMethod
         Algorithm="http://www.w3.org/TR/2000/CR-xml-c14n-20001026"/>
[s04]   <SignatureMethod
         Algorithm="http://www.w3.org/2000/09/xmldsig#dsa-sha1"/>
[s05]   <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
[s06]     <Transforms>
[s07]       <Transform Algorithm="http://www.w3.org/TR/2000/
             CR-xml-c14n-20001026"/>
```

```
[s08]     </Transforms>
[s09]     <DigestMethod Algorithm="http://www.w3.org/2000/09/
           xmldsig#sha1"/>
[s10]     <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
[s11]   </Reference>
[s12] </SignedInfo>
[s13]   <SignatureValue>MC0CFFrVLtRlk=...</SignatureValue>
[s14]   <KeyInfo>
[s15a]    <KeyValue>
[s15b]      <DSAKeyValue>
[s15c]        <P>...</P><Q>...</Q><G>...</G><Y>...</Y>
[s15d]      </DSAKeyValue>
[s15e]    </KeyValue>
[s16]   </KeyInfo>
[s17] </Signature>
```

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

[s03] The CanonicalizationMethod is the algorithm that is used to canonicalize the SignedInfo element before it is digested as part of the signature operation.

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

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

[s14-16] KeyInfo indicates the key to be used to validate the signature. Possible forms for identification include certificates, key names, and key agreement algorithms and information -- we define only a few. KeyInfo is optional for two reasons. First, the signer may not wish to reveal key information to all document processing parties. Second, the information may be known within the application’s context and need not be represented explicitly. Since KeyInfo is outside of SignedInfo, if the signer wishes to bind the keying information to the signature, a Reference can easily identify and include the KeyInfo as part of the signature.

2.1.1 More on Reference

```
[s05]   <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
[s06]     <Transforms>
[s07]       <Transform
             Algorithm="http://www.w3.org/TR/2000/
             CR-xml-c14n-20001026"/>
[s08]     </Transforms>
[s09]     <DigestMethod Algorithm="http://www.w3.org/2000/09/
           xmldsig#sha1"/>
[s10]     <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
[s11]   </Reference>
```

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

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

[s06-08] Transforms is an optional ordered list of processing steps that were applied to the resource’s content before it was digested. Transforms can include operations such as canonicalization, encoding/decoding (including compression/inflation), XSLT and XPath. 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 we specify mandatory (and

optional) canonicalization and decoding algorithms, user specified transforms are permitted.

[s09-10] DigestMethod is the algorithm applied to the data after Transforms is applied (if specified) to yield the DigestValue. The signing of the DigestValue is what binds 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 (message authentication, integrity, 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].)

```
[   ]  <Signature Id="MySecondSignature" ...>
[p01]  <SignedInfo>
[   ]   ...
[p02]   <Reference URI="http://www.w3.org/TR/xml-stylesheet/">
[   ]   ...
[p03]   <Reference URI="#AMadeUpTimeStamp"
[p04]         Type="http://www.w3.org/2000/09/
                    xmldsig#SignatureProperties">
[p05]    <DigestMethod Algorithm="http://www.w3.org/2000/09/
          xmldsig#sha1"/>
[p06]    <DigestValue>k3453rvEPO0vKtMup4NbeVu8nk=</DigestValue>
[p07]   </Reference>
```

```
[p08]  </SignedInfo>
[p09]  ...
[p10]  <Object>
[p11]   <SignatureProperties>
[p12]     <SignatureProperty Id="AMadeUpTimeStamp"
           Target="#MySecondSignature">
[p13]        <timestamp xmlns="http://www.ietf.org/rfc3075.txt">
[p14]          <date>19990908</date>
[p15]          <time>14:34:34:34</time>
[p16]        </timestamp>
[p17]     </SignatureProperty>
[p18]   </SignatureProperties>
[p19]  </Object>
[p20]</Signature>
```

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

[p10] Object is an optional element for including data objects within the signature element or elsewhere. The Object can be optionally typed and/or encoded.

[p11-18] 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 follows.

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.

```
[   ] ...
[m01]   <Reference URI="#MyFirstManifest"
[m02]     Type="http://www.w3.org/2000/09/xmldsig#Manifest">
[m03]     <DigestMethod Algorithm="http://www.w3.org/2000/09/
           xmldsig#sha1"/>
[m04]     <DigestValue>345x3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
[m05]   </Reference>
[   ] ...
[m06] <Object>
[m07]   <Manifest Id="MyFirstManifest">
[m08]     <Reference>
[m09]     ...
[m10]     </Reference>
[m11]     <Reference>
[m12]     ...
[m13]     </Reference>
[m14]   </Manifest>
[m15] </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.

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.

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.

3.2.1 Reference Validation

For each Reference in SignedInfo:

1. Canonicalize the SignedInfo element based on the CanonicalizationMethod in SignedInfo.
2. Obtain the data object to be digested. (The signature application may rely upon the identification (URI) and Transforms provided by the signer in the Reference element, or it may obtain the content through other means such as a local cache.)
3. Digest the resulting data object using the DigestMethod specified in its Reference specification.
4. 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 to ensure the application Sees What is Signed, which is the canonical form. For instance, if the CanonicalizationMethod rewrote the URIs (e.g., absolutizing relative URIs) the signature processing must be cognizant of this.

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 validate 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, internal entity, and simpleType:

Schema Definition:

```
<!DOCTYPE schema
   PUBLIC "-//W3C//DTD XMLSCHEMA 200010//EN"
          "http://www.w3.org/2000/10/XMLSchema.dtd"
  [
   <!ATTLIST schema
     xmlns:ds CDATA #FIXED "http://www.w3.org/2000/09/xmldsig#">
   <!ENTITY dsig 'http://www.w3.org/2000/09/xmldsig#'>
  ]>

<schema xmlns="http://www.w3.org/2000/10/XMLSchema"
      xmlns:ds="&dsig;"
      targetNamespace="&dsig;"
      version="0.1"
      elementFormDefault="qualified">

<!-- Basic Types Defined for Signatures -->

<simpleType name="CryptoBinary">
  <restriction base="binary">
   <encoding value="base64"/>
  </restriction>
</simpleType>
```

DTD:

```
<!-- These entity declarations permit the flexible parts of Signature
     content model to be easily expanded -->

<!ENTITY % Object.ANY '(#PCDATA|Signature|SignatureProperties|
                         Manifest)*'>
<!ENTITY % Method.ANY '(#PCDATA|HMACOutputLength)*'>
<!ENTITY % Transform.ANY '(#PCDATA|XPath|XSLT)'>
<!ENTITY % SignatureProperty.ANY '(#PCDATA)*'>
<!ENTITY % Key.ANY '(#PCDATA|KeyName|KeyValue|RetrievalMethod|
           X509Data|PGPData|MgmtData|DSAKeyValue|RSAKeyValue)*'>
```

4.1 The Signature element

The Signature element is the root element of an XML Signature. Signature elements MUST be laxly schema valid [XML-schema] with respect to the following schema definition:
Schema Definition:

```
<element name="Signature">
  <complexType>
    <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>
</element>
```

DTD:

```
<!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 specify a mandatory and optional to implement SignatureMethod algorithms, user specified algorithms are permitted. Schema Definition:

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

DTD:

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

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:

```
   <element name="SignedInfo">
     <complexType>
       <sequence>
         <element ref="ds:CanonicalizationMethod"/>
         <element ref="ds:SignatureMethod"/>
         <element ref="ds:Reference" maxOccurs="unbounded"/>
       </sequence>
```

```
        <attribute name="Id" type="ID" use="optional"/>
        </complexType>
      </element>
      DTD:

      <!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 Canonical XML [XML-C14N] method.

Alternatives to the REQUIRED Canonical XML algorithm (section 6.5.2), such as Canonical XML with Comments (section 6.5.2) and Minimal Canonicalization (the CRLF and charset normalization specified in section 6.5.1), may be explicitly specified but are NOT REQUIRED. Consequently, their use may not interoperate with other applications that do no 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 minimal or other 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 the two types of algorithms specified by this document:

* Canonical XML [XML-C14N] (with or without comments) implementation MUST be provided with an XPath node-set originally formed from the document containing the SignedInfo and currently indicating the SignedInfo, its descendants, and the attribute and namespace nodes of SignedInfo and its descendant elements (such that the namespace context and similar ancestor information of the SignedInfo is preserved).

* Minimal canonicalization implementations MUST be provided with the 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.

We RECOMMEND that resource constrained applications that do not implement the Canonical XML [XML-C14N] algorithm and instead choose minimal canonicalization (or some other form) be implemented to generate Canonical XML as their output serialization so as to easily mitigate some of these interoperability and security concerns. (While a result might not be the canonical form of the original, it can still be in canonical form.) For instance, such an implementation SHOULD (at least) generate standalone XML instances [XML].
Schema Definition:

```
   <element name="CanonicalizationMethod">
     <complexType>
       <sequence>
         <any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
       </sequence>
       <attribute name="Algorithm" type="uriReference" use="required"/>
     </complexType>
   </element>
```

DTD:

```
   <!ELEMENT CanonicalizationMethod %Method.ANY; >
   <!ATTLIST CanonicalizationMethod
             Algorithm CDATA #REQUIRED >
```

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.
Schema Definition:

```
   <element name="SignatureMethod">
     <complexType>
       <sequence>
         <any namespace="##any" minOccurs="0" maxOccurs="unbounded"/>
       </sequence>
       <attribute name="Algorithm" type="uriReference" use="required"/>
      </complexType>
   </element>
   DTD:

   <!ELEMENT SignatureMethod %Method.ANY; >
   <!ATTLIST SignatureMethod
             Algorithm CDATA #REQUIRED >
```

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">
     <complexType>
       <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="uriReference" use="optional"/>
       <attribute name="Type" type="uriReference" use="optional"/>
     </complexType>
   </element>
   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, as specified by RFC2396 [URI]. The set of allowed characters for URI attributes is the same as for XML, namely [Unicode]. However, some Unicode characters are disallowed from URI references including all non-ASCII characters and the excluded characters listed in RFC2396 [URI, section 2.4]. However, the number sign (#), percent sign (%), and square bracket characters re-allowed in RFC 2732 [URI-Literal] are permitted. Disallowed characters must be escaped as follows:

1. Each disallowed character is converted to [UTF-8] as one or more bytes.
2. Any octets corresponding to a disallowed character are escaped with the URI escaping mechanism (that is, converted to %HH, where HH is the hexadecimal notation of the byte value).
3. The original character is replaced by the resulting character sequence.

XML signature applications MUST be able to parse URI syntax. We RECOMMEND they 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 a 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. This is represented as a URI. For example:

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

The Type attribute applies to the item being pointed at, not its contents. For example, a reference that identifies 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 nodesets" 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 a an octet stream and the next transformrequires a node-set, the signature application MUST attempt to parse the octets.

* If the data object is a node-set and the next transformrequires octets, the signature application MUST attempt to convert the node-set to an octet stream using the REQUIRED canonicalization algorithm [XML-C14N].

Users may specify alternative transforms that over-ride 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.

Unless the URI-Reference is a ’same-document’ reference as defined in [URI, Section 4.2], 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 transformthat 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 that the URI attribute not include fragment identifiers and that such processing be specified as an additional XPath Transform.

When a fragment is not preceded by a URI in the URI-Reference, XML signature applications MUST support the null URI and barename XPointer. We RECOMMEND support for the same-document XPointers ’#xpointer(/)’ and ’#xpointer(id("ID"))’ if the application also intends to support Minimal Canonicalization or Canonical XML with Comments. (Otherwise URI="#foo" will automatically remove comments before the Canonical XML with Comments can even be invoked.) All other support for XPointers is OPTIONAL, especially all support for barename 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).

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 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 transformrather than a URI fragment (barename XPointer resolution in external resources is not REQUIRED in this specification).

URI=""
    Identifies the nodeset (minus any comment nodes) of the XML resource containing the signature

URI="#chapter1"
: Identifies a nodeset containing the element with ID attribute value 'chapter1' of the XML resource containing the signature. XML Signature (and its applications) modify this nodeset to include the element plus all descendents 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. Specifically, dereferencing a null URI (URI="") MUST result in an XPath node-set that includes every non-comment node of the XML document containing the URI attribute. In a fragment URI, the characters after the number sign ('#') character conform to the XPointer syntax [Xptr]. When processing an XPointer, the application MUST behave as if the root node of the XML document containing the URI attribute were used to initialize the XPointer evaluation context. The application MUST behave as if the result of XPointer processing were a node-set derived from the resultant location-set as follows:

1. discard point nodes
2. replace each range node with all XPath nodes having full or partial content within the range
3. replace the root node with its children (if it is in the node-set)
4. 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.
5. if the URI is not a full 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, barename XPointers and child sequence XPointers. To retain comments while selecting an element by an identifier ID, use the following full XPointer: URI='#xpointer(id("ID"))'. To retain comments while selecting the entire document, use the following full XPointer: URI='#xpointer(/)'. This XPointer contains a simple XPath expression that includes the root node, which the second to last step above replaces with all nodes of the parse tree (all descendants, plus all attributes, plus all namespaces nodes).

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 Transform 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:

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

  <element name="Transform">
    <complexType>
      <choice maxOccurs="unbounded">
        <any namespace="##other" processContents="lax" minOccurs="0"
         maxOccurs="unbounded"/>
        <element name="XSLT" type="string"/>
        <!-- should be an xsl:stylesheet element -->
        <element name="XPath" type="string"/>
      </choice>
      <attribute name="Algorithm" type="uriReference" use="required"/>
    </complexType>
  </element>
```

DTD:

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

<!ELEMENT Transform %Transform.ANY; >
<!ATTLIST Transform
          Algorithm    CDATA    #REQUIRED >

<!ELEMENT XPath (#PCDATA) >
<!ELEMENT XSLT (#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 Minimal Canonicalization or 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">
     <complexType>
       <sequence>
         <any namespace="##any" processContents="lax" minOccurs="0"
         maxOccurs="unbounded"/>
       </sequence>
       <attribute name="Algorithm" type="uriReference" use="required"/>
     </complexType>
   </element>
```

DTD:

```
   <!ELEMENT DigestMethod %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:CryptoBinary"/>
```

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 place their own key identification and exchange semantics within this element type through the XML-namespace facility [XML-ns].

If KeyInfo is omitted, the recipient is expected to be able to identify the key based on application context information. 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 following list summarizes the KeyInfo types defined by this specification; these can be used within the RetrievalMethod Type attribute to describe the remote KeyInfo structure as represented as an octect stream.

* 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 structures, we specify one additional type to indicate a binary X.509 Certificate

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

Schema Definition:

```
<element name="KeyInfo">
  <complexType>
    <choice maxOccurs="unbounded">
      <any processContents="lax" namespace="##other" minOccurs="0"
       maxOccurs="unbounded"/>
      <element name="KeyName" type="string"/>
      <element ref="ds:KeyValue"/>
      <element ref="ds:RetrievalMethod"/>
      <element ref="ds:X509Data"/>
      <element ref="ds:PGPData"/>
      <element ref="ds:SPKIData"/>
      <element name="MgmtData" type="string"/>
    </choice>
    <attribute name="Id" type="ID" use="optional"/>
  </complexType>
</element>
```

DTD:

```
<!ELEMENT KeyInfo %Key.ANY; >
<!ATTLIST KeyInfo
          Id ID  #IMPLIED >
```

4.4.1 The KeyName Element

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

Schema Definition:

```
<!-- type declared in KeyInfo -->
```

DTD:

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

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) and RSA (RECOMMENDED) public keys are defined in Signature Algorithms (section 6.4).
Schema Definition:

```
<element name="KeyValue">
  <complexType mixed="true">
    <choice>
      <any namespace="##other" processContents="lax" minOccurs="0"
       maxOccurs="unbounded"/>
      <element ref="ds:DSAKeyValue"/>
      <element ref="ds:RSAKeyValue"/>
    </choice>
  </complexType>
</element>
```

DTD:

```
<!ELEMENT KeyValue    %Key.ANY; >
```

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. Note, if the result of dereferencing and transforming the specified URI is a node set, then it may need to be to be canonicalized. All of the KeyInfo types defined by this specification (section 4.4) represent octets, consequently the Signature application is expected to attempt to canonicalize the nodeset via the The Reference Processing Model (section 4.3.3.2)

Type is an optional identifier for the type of data to be retrieved. Schema Definition

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

DTD

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

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 revocation lists). Five types of X509Data are defined

1. The X509IssuerSerial element, which contains an X.509 issuer distinguished name/serial number pair that SHOULD be compliant with RFC2253 [LDAP-DN],
2. The X509SubjectName element, which contains an X.509 subject distinguished name that SHOULD be compliant with RFC2253 [LDAP-DN],
3. The X509SKI element, which contains an X.509 subject key identifier value.
4. The X509Certificate element, which contains a base64-encoded [X509v3] certificate, and
5. The X509CRL element, which contains a base64-encoded certificate revocation list (CRL) [X509v3].

Multiple declarations about a single certificate (e.g., a X509SubjectName and X509IssuerSerial element) MUST be grouped inside a single X509Data element; multiple declarations about the same key but different certificates (related to that single key) MUST be grouped within a single KeyInfo element but MAY occur in multiple X509Data elements. For example, the following block contains two pointers to certificate-A (issuer/serial number and SKI) and a single reference to certificate-B (SubjectName) and also shows use of certificate elements

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

Note, there is no direct provision for a PKCS#7 encoded "bag" of certificates or CRLs. However, a set of certificates or a CRL 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.
Schema Definition

```
 <element name="X509Data">
    <complexType>
     <choice>
       <sequence maxOccurs="unbounded">
         <choice>
           <element ref="ds:X509IssuerSerial"/>
           <element name="X509SKI" type="ds:CryptoBinary"/>
           <element name="X509SubjectName" type="string"/>
```

```
             <element name="X509Certificate" type="ds:CryptoBinary"/>
           </choice>
         </sequence>
         <element name="X509CRL" type="ds:CryptoBinary"/>
       </choice>
     </complexType>
   </element>

   <element name="X509IssuerSerial">
      <complexType>
       <sequence>
         <element name="X509IssuerName" type="string"/>
         <element name="X509SerialNumber" type="integer"/>
       </sequence>
      </complexType>
   </element>
```

DTD

```
  <!ELEMENT X509Data ((X509IssuerSerial | X509SKI | X509SubjectName |
                       X509Certificate)+ | X509CRL)>
   <!ELEMENT X509IssuerSerial (X509IssuerName, X509SerialNumber) >
   <!ELEMENT X509IssuerName (#PCDATA) >
   <!ELEMENT X509SubjectName (#PCDATA) >
   <!ELEMENT X509SerialNumber (#PCDATA) >
   <!ELEMENT X509SKI (#PCDATA) >
   <!ELEMENT X509Certificate (#PCDATA) >
   <!ELEMENT X509CRL (#PCDATA) >
```

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 string 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]. Other sub-types of the PGPData element may be defined by the OpenPGP working group.
Schema Definition:

```
<element name="PGPData">
  <complexType>
    <choice>
```

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

DTD:

```
   <!ELEMENT PGPData (PGPKeyID, PGPKeyPacket)  >
   <!ELEMENT PGPKeyPacket  (#PCDATA)  >
   <!ELEMENT PGPKeyID  (#PCDATA)  >
```

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.
The content of this element type is expected to be a Canonical S-
expression.
Schema Definition:

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

DTD:

```
   <!ELEMENT SPKIData (#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.
Schema Definition:

```
   <!-- type declared in KeyInfo -->
   DTD:

   <!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 MimeType attribute is an optional attribute which describes the data within the Object. This is a string with values defined by [MIME]. For example, if the Object contains XML, the MimeType could be text/xml. This attribute is purely advisory; no validation of the MimeType information is required by this specification.

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.

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

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:

```
   <element name="Object">
     <complexType mixed="true">
       <sequence maxOccurs="unbounded">
         <any namespace="##any" processContents="lax"/>
```

```
        </sequence>
        <attribute name="Id" type="ID" use="optional"/>
        <attribute name="MimeType" type="string" use="optional"/>
           <!-- add a grep facet -->
        <attribute name="Encoding" type="uriReference" use="optional"/>
      </complexType>
    </element>
```

DTD:

```
   <!ELEMENT Object %Object.ANY; >
   <!ATTLIST Object
             Id ID  #IMPLIED
             MimeType   CDATA   #IMPLIED
             Encoding   CDATA   #IMPLIED >
```

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:

```
   <element name="Manifest">
     <complexType>
       <sequence>
         <element ref="ds:Reference" maxOccurs="unbounded"/>
```

```
        </sequence>
        <attribute name="Id" type="ID" use="optional"/>
      </complexType>
   </element>
   DTD:

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

5.2 The SignatureProperties Element

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

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

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

      <element name="SignatureProperty">
        <complexType mixed="true">
          <choice minOccurs="0" maxOccurs="unbounded">
            <any namespace="##other" processContents="lax" minOccurs="0"
            maxOccurs="unbounded"/>
          </choice>
          <attribute name="Target" type="uriReference" use="required"/>
          <attribute name="Id" type="ID" use="optional"/>
          </complexType>
      </element>
   DTD:

   <!ELEMENT SignatureProperties (SignatureProperty+)  >
   <!ATTLIST SignatureProperties
             Id ID   #IMPLIED  >
   <!ELEMENT SignatureProperty %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 specified 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] does), they will be signed. Consequently, if they are retained, a change to the comment will cause a signature failure. Similarly, the XML signature over any XML data will be sensitive to comment changes unless a comment-ignoring canonicalization/transform method, such as the Canonical XML [XML-C14N], is specified.

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

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.

(Note that the normative identifier is the complete URI in the table though they are sometimes abbreviated in XML syntax (e.g., "&dsig;base64").)

```
Algorithm Type
   Algorithm - Requirements - Algorithm URI
Digest
   SHA1  - REQUIRED - &dsig;sha1
Encoding
   base64  - REQUIRED - &dsig;base64
MAC
   HMAC-SHA1 - REQUIRED - &dsig;hmac-sha1
Signature
   DSAwithSHA1(DSS) - REQUIRED - &dsig;dsa-sha1
   RSAwithSHA1 - RECOMMENDED - &dsig;rsa-sha1
Canonicalization
   minimal - RECOMMENDED - &dsig;minimal
   Canonical XML with Comments - RECOMMENDED -
      http://www.w3.org/TR/2000/CR-xml-c14n-20001026#WithComments
   Canonical XML (omits comments) - REQUIRED -
      http://www.w3.org/TR/2000/CR-xml-c14n-20001026
Transform
   XSLT - OPTIONAL - http://www.w3.org/TR/1999/REC-xslt-19991116
   XPath - RECOMMENDED -
      http://www.w3.org/TR/1999/REC-xpath-19991116
   Enveloped Signature* - REQUIRED - &dsig;enveloped-signature
```

* 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 RECOMMENDED XPath specification specified in 6.6.4: Enveloped Signature Transform; it MUST have the same effect as that specified by the XPath Transform.

6.2 Message Digests

Only one digest algorithm is defined herein. However, it is expected that one or more additional strong digest algorithms will be developed in connection with the US Advanced Encryption Standard effort. Use of MD5 [MD5] is NOT RECOMMENDED because recent advances in cryptography have cast doubt on its strength.

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:

```
<DigestMethod Algorithm="&dsig;sha1"/>
```

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

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

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

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

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

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

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:

```
<SignatureMethod Algorithm="&dsig;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

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

Schema Definition:

```
<element name="HMACOutputLength" type="integer"/>
```

DTD:

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

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:

```
<SignatureMethod Algorithm="&dsig;dsa"/>
```

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. Integer to octet-stream conversion must be done according to the I2OSP operation defined in the RFC 2437 [PKCS1] specification with a k parameter equal to 20. For example, the SignatureValue element for a DSA signature (r, s) with values specified in hexadecimal:

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

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

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

DSA key values have the following set of fields: P, Q, G and Y are mandatory when appearing as a key value, J, seed and pgenCounter are optional but should be present. (The seed and pgenCounter fields must appear together or be absent). All parameters are encoded as base64 [MIME] values.
Schema:

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

DTD:

```
<!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) >
```

6.4.2 PKCS1

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

Arbitrary-length integers (e.g., "bignums" such as RSA modulii) are represented in XML as octet strings. The integer value is first converted to a "big endian" bitstring. The bitstring is then padded with leading zero bits so that the total number of bits == 0 mod 8 (so that there are an even number of bytes). If the bitstring contains entire leading bytes that are zero, these are removed (so the high-order byte 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).

The expression "RSA algorithm" as used in this document 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: <SignatureMethod Algorithm="&dsig;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 concatentation, "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 09 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>IWijxQjUrcXBYoCei4QxjWo9Kg8D3p9tlWoT4
   t0/gyTE96639In0FZFY2/rvP+/bMJ01EArmKZsR5VW3rwoPxw=
   </SignatureValue>
```

RSA key values have two fields Modulus and Exponent

```
   <RSAKeyValue>

<Modulus>xA7SEU+e0yQH5rm9kbCDN9o3aPIo7HbP7tX6WOocLZAtNfyxSZDU16ksL6W

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

Schema:

```
<element name="RSAKeyValue">
  <complexType>
    <sequence>
      <element name="Modulus" type="ds:CryptoBinary"/>
      <element name="Exponent" type="ds:CryptoBinary"/>
    </sequence>
  </complexType>
</element>
```

DTD:

```
<!ELEMENT RSAKeyValue (Modulus, Exponent) >
<!ELEMENT Modulus (#PCDATA) >
<!ELEMENT Exponent (#PCDATA) >
```

6.5 Canonicalization Algorithms

If canonicalization is performed over octets, the canonicalization algorithms take two implicit parameter: 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 two 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 two algorithms below perform text normalization during transcoding [NFC]. 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.)

6.5.1 Minimal Canonicalization

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

An example of a minimal canonicalization element is:
<CanonicalizationMethod Algorithm="&dsig;minimal"/>

The minimal canonicalization algorithm:

* converts the character encoding to UTF-8 (without any byte order mark (BOM)). If an encoding is given in the XML declaration, it must be removed. Implementations MUST understand at least [UTF-8] and [UTF-16] as input encodings. Non-Unicode to Unicode transcoding MUST perform text normalization [NFC].
* normalizes line endings as provided by [XML]. (See XML and Canonicalization and Syntactical Considerations (section 7).)

This algorithm requires as input the octet stream of the resource to be processed; the algorithm outputs an octet stream. When used to canonicalize SignedInfo the algorithm MUST be provided with the octets that represent the well-formed SignedInfo element (and its children and content) as described in The CanonicalizationMethod Element (section 4.3.1).

If the signature application has a node set, then the signature application must convert it into octets as described in The Reference Processing Model (section 4.3.3.2). However, Minimal Canonicalization is NOT RECOMMENDED for processing XPath node-sets, the results of same-document URI references, and the output of other types of XML based transforms. It is only RECOMMENDED for simple character normalization of well formed XML that has no namespace or external entity complications.

6.5.2 Canonical XML

Identifier for REQUIRED Canonical XML (omits comments):
      http://www.w3.org/TR/2000/CR-xml-c14n-20001026

Identifier for Canonical XML with Comments:
      http://www.w3.org/TR/2000/CR-xml-c14n-20001026#WithComments

An example of an XML canonicalization element is:

```
<CanonicalizationMethod Algorithm="http://www.w3.org/TR/2000/CR-xml-
c14n-20001026"/>
```

The normative specification of Canonical XML 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.

6.6 Transform Algorithms

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

Application developers are strongly encouraged to support all transforms listed in this section 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

The normative specification for base 64 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 requires an octet stream for input. If an XPath node-set (or sufficiently functional alternative) is given as input, then it is converted to an octet stream by performing operations logically equivalent to 1) applying an XPath transform with expression self::text(), then 2) taking the string-value of the node-set. Thus, if an XML element is identified by a barename 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

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. Note that if the actual input is an XPath node-set resulting from a null URI or barename 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 also an XPath node-set. The XPath expression appearing in the XPath parameter is evaluated once for each node in the input node-set. The result is converted to a boolean. If the boolean is true, then the node is included in the output node-set. If the boolean is false, then the node is omitted from the output node-set.

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

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

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 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.

Note: The function definition for here() is intended to be consistent with its definition in XPointer. However, some minor differences are presently being discussed between the Working Groups.

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,

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

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

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

```
<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

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

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

The input and output requirements of this transform are identical to those of the XPath transform. 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

The normative specification for XSL Transformations is [XSLT]. The XSL style sheet or transform to be evaluated appears as the character content of a transform parameter child element named XSLT. The root element of a XSLT style sheet SHOULD be <xsl:stylesheet>.

This transform requires an octet stream as input. If the actual input is an XPath node-set, then the signature application should attempt to covert it to octets (apply Canonical XML]) as described in the Reference Processing Model (section 4.3.3.2).

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 transformauthors 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 transformafter the XSLT transformto perform 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 three 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. And, 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.

Any canonicalization algorithm should yield output in a specific fixed coded character set. For both the minimal canonicalization defined in this specification and Canonical XML [XML-C14N] that coded character set is UTF-8 (without a byte order mark (BOM)).Neither the minimal canonicalization nor the Canonical XML [XML-C14N] algorithms provide character normalization. We RECOMMEND that signature applications create XML content (Signature elements and their descendents/content) in Normalization Form C [NFC] and check that any XML being consumed is in that form as well (if not, signatures may consequently fail to validate). Additionally, none of these algorithms provide data type normalization. Applications that normalize data types in varying formats (e.g., (true, false) or (1,0)) may not be able to validate each other’s signatures.

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
   A. replacing character and entity references as above,
   B. replacing occurrences of #x9, #xA, and #xD with #x20 (space) except that the sequence #xD#xA is replaced by a single space, and

   C. 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 (5C) 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 XML1.0 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 octect stream that was signed.

8.0 Security Considerations

The XML Signature specification provides a very flexible digital signature mechanism. Implementors 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 application 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 can not rely upon canonicalization to do this for them.

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 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 it "sees," persons and automated mechanisms 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 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. Consequently, while we RECOMMEND all documents operated upon and generated by signature applications be in [NFC] (otherwise intermediate processors might unintentionally break the signature) encoding normalizations SHOULD NOT be done as part of a signature transform, or (to state it another way) if normalization does occur, the application SHOULD always "see" (operate over) the normalized form.

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 [RANDOM] and the size of the key, the security of key and certificate authentication and distribution mechanisms, certificate chain validation policy, protection of cryptographic processing from hostile observation and tampering, etc.

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

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

9.0 Schema, DTD, Data Model, and Valid Examples

```
   XML Signature Schema Instance
         http://www.w3.org/TR/2000/CR-xmldsig-core-20001031/xmldsig-
           core-schema.xsd   Valid XML schema instance based on the
         20000922 Schema/DTD [XML-Schema].

   XML Signature DTD
         http://www.w3.org/TR/2000/CR-xmldsig-core-20001031/xmldsig-
           core-schema.dtd

   RDF Data Model
         http://www.w3.org/TR/2000/CR-xmldsig-core-20001031/xmldsig-
           datamodel-20000112.gif

   XML Signature Object Example
         http://www.w3.org/TR/2000/CR-xmldsig-core-20001031/signature-
           example.xml   A cryptographical invalid XML example that
         includes foreign content and validates under the schema.  (It
         validates under the DTD when the foreign content is removed or
         the DTD is modified accordingly).

   RSA XML Signature Example
         http://www.w3.org/TR/2000/CR-xmldsig-core-20001031/signature-
           example-rsa.xml
         An XML Signature example with generated cryptographic values by
            Merlin Hughes and validated by Gregor Karlinger.

   DSA XML Signature Example
         http://www.w3.org/TR/2000/CR-xmldsig-core-20001031/signature-
           example-dsa.xml   Similar to above but uses DSA.
```

10.0 Definitions

Authentication Code
: 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 (integrity) but not signer authentication

Authentication, Message
: "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
: "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]

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 inability to change a message without also changing the signature value. See message authentication.

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 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 signer authentication and message authentication (integrity). (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 the structure of the Signature element type and its children (including SignatureValue) and mandatory to support 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 octet stream from source content to derived content. 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

ABA	Digital Signature Guidelines. http://www.abanet.org/scitech/ec/isc/dsgfree.html
Bourret	Declaring Elements and Attributes in an XML DTD. Ron Bourret. http://www.informatik.tu-darmstadt.de/DVS1/staff/bourret/xml/xmldtd.html
DOM	Document Object Model (DOM) Level 1 Specification. W3C Recommendation. V. Apparao, S. Byrne, M. Champion, S. Isaacs, I. Jacobs, A. Le Hors, G. Nicol, J. Robie, R. Sutor, C. Wilson, L. Wood. October 1998. http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

DSS	FIPS PUB 186-1. Digital Signature Standard (DSS). U.S. Department of Commerce/National Institute of Standards and Technology. http://csrc.nist.gov/fips/fips1861.pdf
HMAC	Krawczyk, H., Bellare, M. and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, February 1997. http://www.ietf.org/rfc/rfc2104.txt
HTTP	Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999. http://www.ietf.org/rfc/rfc2616.txt
KEYWORDS	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997. http://www.ietf.org/rfc/rfc2119.txt
LDAP-DN	Wahl, M., Kille, S. and T. Howes, "Lightweight Directory Access Protocol (v3): UTF-8 String Representation of Distinguished Names", RFC 2253, December 1997. http://www.ietf.org/rfc/rfc2253.txt
MD5	Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992. http://www.ietf.org/rfc/rfc1321.txt
MIME	Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, November 1996. http://www.ietf.org/rfc/rfc2045.txt
NFC	TR15. Unicode Normalization Forms. M. Davis, M. Drst. Revision 18: November 1999.
PGP	Callas, J., Donnerhacke, L., Finney, H. and R. Thayer, "OpenPGP Message Format", November 1998. http://www.ietf.org/rfc/rfc2440.txt
RANDOM	Eastlake, D., Crocker, S. and J. Schiller, "Randomness Recommendations for Security", RFC 1750, December 1994. http://www.ietf.org/rfc/rfc1750.txt

RDF	RDF Schema W3C Candidate Recommendation. D. Brickley, R.V. Guha. March 2000. http://www.w3.org/TR/2000/CR-rdf-schema-20000327/ RDF Model and Syntax W3C Recommendation. O. Lassila, R. Swick. February 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/
1363	IEEE 1363: Standard Specifications for Public Key Cryptography. August 2000.
PKCS1	Kaliski, B. and J. Staddon, "PKCS #1: RSA Cryptography Specifications Version 2.0", RFC 2437, October 1998. http://www.ietf.org/rfc/rfc2437.txt
SAX	SAX: The Simple API for XML David Megginson et. al. May 1998. http://www.megginson.com/SAX/index.html
SHA-1	FIPS PUB 180-1. Secure Hash Standard. U.S. Department of Commerce/National Institute of Standards and Technology. http://csrc.nist.gov/fips/fip180-1.pdf
Unicode	The Unicode Consortium. The Unicode Standard. http://www.unicode.org/unicode/standard/standard.html
UTF-16	Hoffman, P. and F. Yergeau, "UTF-16, an encoding of ISO 10646", RFC 2781, February 2000. http://www.ietf.org/rfc/rfc2781.txt
UTF-8	Yergeau, F., "UTF-8, a transformation format of ISO 10646", RFC 2279, January 1998. http://www.ietf.org/rfc/rfc2279.txt
URI	Berners-Lee, T., Fielding, R. and L. Masinter, "Uniform Resource Identifiers (URI): Generic Syntax", RFC 2396, August 1998. http://www.ietf.org/rfc/rfc2396.txt
URI-Literal	Hinden, R., Carpenter, B. and L. Masinter, "Format for Literal IPv6 Addresses in URL’s", RFC 2732, December 1999. http://www.ietf.org/rfc/rfc2732.txt
URL	Berners-Lee, T., Masinter, L. and M. McCahill, "Uniform Resource Locators (URL)", RFC 1738, December 1994. http://www.ietf.org/rfc/rfc1738.txt

```
   URN                  Moats, R., "URN Syntax" RFC 2141, May 1997.
                        http://www.ietf.org/rfc/rfc2141.txt

                        Daigle, L., van Gulik, D., Iannella, R. and P.
                        Faltstrom, "URN Namespace Definition Mechanisms",
                        RFC 2611, June 1999.
                        http://www.ietf.org/rfc/rfc2611.txt

   X509v3               ITU-T Recommendation X.509 version 3 (1997).
                        "Information Technology - Open Systems
                        Interconnection - The Directory Authentication
                        Framework" ISO/IEC 9594-8:1997.

   XHTML 1.0            XHTML(tm) 1.0: The Extensible Hypertext Markup
                        Language Recommendation. S. Pemberton, D. Raggett,
                        et. al. January 2000.
                        http://www.w3.org/TR/2000/REC-xhtml1-20000126/

   XLink                XML Linking Language. Working Draft. S. DeRose, D.
                        Orchard, B. Trafford. July 1999.
                        http://www.w3.org/1999/07/WD-xlink-19990726

   XML                  Extensible Markup Language (XML) 1.0
                        Recommendation. T. Bray, J. Paoli, C. M. Sperberg-
                        McQueen. February 1998.
                        http://www.w3.org/TR/1998/REC-xml-19980210

   XML-C14N             J. Boyer, "Canonical XML Version 1.0", RFC 3076,
                        September 2000.  http://www.w3.org/TR/2000/CR-xml-
                        c14n-20001026
                        http://www.ietf.org/rfc/rfc3076.txt

   XML-Japanese         XML Japanese Profile. W3C NOTE. M. MURATA April
                        2000 http://www.w3.org/TR/2000/NOTE-japanese-xml-
                        20000414/

   XML-MT               Whitehead, E. and M. Murata, "XML Media Types",
                        July 1998.  http://www.ietf.org/rfc/rfc2376.txt

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

   XML-schema           XML Schema Part 1: Structures Working Draft. D.
                        Beech, M. Maloney, N. Mendelshohn. September 2000.
                        http://www.w3.org/TR/2000/WD-xmlschema-1-20000922/
                        XML Schema Part 2: Datatypes Working Draft. P.
                        Biron, A. Malhotra. September 2000.
                        http://www.w3.org/TR/2000/WD-xmlschema-2-20000922/

   XML-Signature-RD     Reagle, J., "XML Signature Requirements", RFC 2907,
                        April 2000.  http://www.w3.org/TR/1999/WD-xmldsig-
                        requirements-19991014
                        http://www.ietf.org/rfc/rfc2807.txt

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

   XPointer             XML Pointer Language (XPointer). Candidate
                        Recommendation. S. DeRose, R. Daniel, E. Maler.
                        http://www.w3.org/TR/2000/CR-xptr-20000607

   XSL                  Extensible Stylesheet Language (XSL) Working Draft.
                        S. Adler, A. Berglund, J. Caruso, S. Deach, P.
                        Grosso, E. Gutentag, A. Milowski, S. Parnell, J.
                        Richman, S. Zilles. March 2000.
                        http://www.w3.org/TR/2000/WD-xsl-
                        20000327/xslspec.html

   XSLT                 XSL Transforms (XSLT) Version 1.0. Recommendation.
                        J. Clark. November 1999.
                        http://www.w3.org/TR/1999/REC-xslt-19991116.html
```

12. Authors' Addresses

Donald E. Eastlake 3rd
Motorola, Mail Stop: M2-450
20 Forbes Boulevard
Mansfield, MA 02048 USA

Phone: 1-508-261-5434
EMail: Donald.Eastlake@motorola.com

Joseph M. Reagle Jr., W3C
Massachusetts Institute of Technology
Laboratory for Computer Science
NE43-350, 545 Technology Square
Cambridge, MA 02139

Phone: 1.617.258.7621
EMail: reagle@w3.org

David Solo
Citigroup
909 Third Ave, 16th Floor
NY, NY 10043 USA

Phone: +1-212-559-2900
EMail: dsolo@alum.mit.edu

13. Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.