be advanced including:
1. Finalize the disposition of all last call comments.
2. We are not confident that our use of schema namespaces and qualifications provides a single schema that can be used for enveloped signatures (signature within content being signed), enveloping signatures (content is within signature being signed) and detached signatures (over data external to the signature document).
3. Further work and coordination needs to be done regarding our employment of URIs, IDs, and XPath
   - Including the specification of algorithm identifier as defined by XPath that eliminates the Signature element within enveloped signatures.
4. Ensure that if the syntax constraints of section 7.1 are followed, a validating parser is not needed.
5. Clarify any remaining ambiguities in the KeyInfo section.
6. Update the acknowledgements to reflect changes in working group participation and the comments sent during Last Call.

Please send comments to the editors and cc: the list <w3c-ietf-xmldsig@w3.org>. Publication as a Working Draft does not imply endorsement by the W3C membership or IESG. It is inappropriate to cite W3C Drafts as other than "work in progress." A list of current W3C working drafts can be found at http://www.w3.org/TR/. Current IETF drafts can be found at http://www.ietf.org/1id-abstracts.html.

Patent disclosures relevant to this specification may be found on the Working Group’s patent disclosure page.

Abstract
This document specifies XML digital signature processing rules and syntax. XML Signatures provide integrity, message authentication, and/or signer authentication services for data of any type, whether located within the XML that includes the signature or elsewhere.

Table of Contents
1. Introduction
   - Editorial Conventions
   - Design Philosophy
   - Versions, Namespaces and Identifiers
   - Acknowledgements
2. Signature Overview and Examples
   - Simple Example (Signature, SignedInfo, Methods, and References)
     - More on Reference
   - Extended Example (Object and SignatureProperty)
   - Extended Example (Object and Manifest)
3. Processing Rules
   - Signature Generation
   - Signature Validation
4. Core Signature Syntax
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 document.

This specification also defines other useful types including methods of referencing collections of resources, algorithms, and keying information and management.

1.1 Editorial Conventions

Eastlake, Reagle, Solo
For readability, brevity, and historic reasons this document uses the term "signature" to generally refer to digital authentication values of all types. Obviously, the term is also strictly used to refer to authentication values that are based on public keys and that provide signer authentication. When specifically discussing authentication values based on symmetric secret key codes we use the terms authenticators or authentication codes. (See section 8.3: Check the Security Model.)

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 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/2000/02/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/2000/02/xmldsig#SignatureProperties

XSLT is identified and defined by an external namespace
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/2000/02/xmldsig#sha1

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

```xml
<?xml version='1.0'?>
<!DOCTYPE Signature SYSTEM
"xmldsig-core-schema.dtd" [ <!ENTITY dsig
"http://www.w3.org/2000/02/xmldsig#" ]>
<Signature xmlns="&dsig;" Id="MyFirstSignature">
  ...
</Signature>
```

1.4 Acknowledgements

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

* Milton Anderson, FSTC
* Mark Bartel, JetForm Corporation (Author)
* John Boyer, PureEdge (Author)
* Richard Brown, Globeset
* Donald Eastlake 3rd, Motorola (Chair, Author/Editor)
* Barb Fox, Microsoft (Author)
* Tom Gindin, IBM
* Phillip Hallam-Baker, VeriSign Inc
* Richard Himes, US Courts
* Brian LaMacchia, Microsoft
* Peter Lipp, IAIK TU Graz
* Joseph Reagle, W3C (Chair, Author/Editor)
* Ed Simon , Entrust Technologies Inc. (Author)
* Chris Smithies, PenOp
* David Solo, Citigroup (Author/Editor)
2.0 Signature Overview and Examples

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

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

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

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

The content that is signed was, at the time of signature creation, referred to as an identified resource to which the specified transforms were applied. Within an XML document, signatures are related to data objects local fragment identifiers and the data can be included within an enveloping signature or can enclose an enveloped signature. Signatures are related to external data objects via URIs [URI] and the signature and data object are detached.

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

The CanonicalizationMethod is the algorithm that is used to canonicalize the SignedInfo element before it is digested as part of the signature operation. In the absence of a CanonicalizationMethod element, no canonicalization is done.

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.

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

[05] <Reference URI="http://www.w3.org/TR/2000/REC-xhtml1-20000126/">
[06]     <Transforms>
[07]       <Transform Algorithm="http://www.w3.org/2000/02/xmldsig#c14n"/>
[08]     </Transforms>
[09]     <DigestMethod Algorithm="http://www.w3.org/2000/02/xmldsig#sha1"/>
[10]     <DigestValue>j6lwx3rvEPO0vKtMup4NbeVu8nk=</DigestValue>

[05] 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.)

[05-08] 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.

[06-08] 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.
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 as foo:assuredby attribute within its own markup to references 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].)

```xml
<SignatureProperties Id="AMadeUpTimeStamp">
  <SignatureProperty Target="#MySecondSignature">
    <timestamp xmlns="http://www.ietf.org/rfcXXXX.txt">
      <date>19990908</date>
      <time>14:34:34:34</time>
    </timestamp>
  </SignatureProperty>
</SignatureProperties>
```
The optional Type attribute 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.

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

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

2.3 Extended Example (Object and Manifest)

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

```xml
<Reference URI="#MyFirstManifest"
     Type="http://www.w3.org/2000/02/xmldsig#Manifest">
  <DigestMethod Algorithm="http://www.w3.org/2000/02/xmldsig#sha1"/>
  <DigestValue>345x3rvEPO0vKtMup4NbeVu8nk=</DigestValue>
</Reference>

<Object>
  <Manifest Id="MyFirstManifest">
    <Reference>
    ...
    </Reference>
    <Reference>
    ...
    </Reference>
  </Manifest>
</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 if required, 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. 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.)
2. Digest the resulting data object using the DigestMethod specified in its Reference specification.
3. Compare the generated digest value against DigestValue in the SignedInfo Reference; if there is any mismatch, validation fails.

3.2.2 Signature Validation

1. Canonicalize the SignedInfo element based on the CanonicalizationMethod, if any, in SignedInfo.
2. Obtain the keying information from KeyInfo or from an external source.
3. Use the specified SignatureMethod to validate the SignatureValue over the (optionally canonicalized) SignedInfo element.

4.0 Core Signature Syntax

The general structure of an XML signature is described in section 2: Signature Overview. This section provides detailed syntax of the core signature features and actual examples. 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:

```xml
<?xml version='1.0'?>
<!DOCTYPE schema SYSTEM 'http://www.w3.org/1999/XMLSchema.dtd' [
<!ENTITY dsig 'http://www.w3.org/2000/02/xmldsig#'>
]>```
<schema targetNamespace='&dsig;' version='0.1' xmlns='http://www.w3.org/1999/XMLSchema' xmlns:ds='&dsig;' elementFormDefault='qualified'>

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

<simpleType name='CryptoBinary' base='binary'>
  <encoding value='base64'/>
</simpleType>

DTD:

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

<!ENTITY % Object.ANY '(#PCDATA|SignatureProperties|Manifest)*'>
<!ENTITY % Method.ANY '(#PCDATA|HMACOutputLength)*'>
<!ENTITY % Transform.ANY '(#PCDATA|XPath|XSLT)*'>
<!ENTITY % Key.ANY '(#PCDATA|DSAKeyValue|RSAKeyValue)*'>

4.1 The Signature element

The Signature element is the root element of a XML Signature. A simple example of a complete signature follows:

Schema Definition:

<element name='Signature'>
  <complexType content='elementOnly'>
    <sequence minOccurs='1' maxOccurs='1'>
      <element ref='ds:SignedInfo' minOccurs='1' maxOccurs='1'/>
      <element ref='ds:SignatureValue' minOccurs='1' maxOccurs='1'/>
      <element ref='ds:KeyInfo' minOccurs='0' maxOccurs='1'/>
      <element ref='ds:Object' minOccurs='0' maxOccurs='unbounded'/>
    </sequence>
    <attribute name='Id' type='ID'/>
  </complexType>
</element>

DTD:

<!ELEMENT Signature (SignedInfo, SignatureValue, KeyInfo?, Object*) >
<!ATTLIST Signature
  xmlns CDATA #FIXED 'http://www.w3.org/2000/02/xmldsig#'
  Id ID #IMPLIED >

4.2 The SignatureValue Element

The SignatureValue element contains the actual value of the digital signature; it is encoded according to the identifier specified in SignatureMethod. Base64 [MIME] is the encoding method for all SignatureMethods specified within this specification. While we specify a mandatory (and optional) SignatureMethod algorithm, user specified
algorithms (with their own encodings) are permitted.

Schema Definition:

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

DTD:

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

4.3 The SignedInfo Element

The structure of SignedInfo includes the canonicalization algorithm (if any), 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.

Schema Definition:

```xml
<element name='SignedInfo'>
  <complexType content='elementOnly'>
    <sequence minOccurs='1' maxOccurs='1'>
      <element ref='ds:CanonicalizationMethod' minOccurs='0' maxOccurs='1'/>
      <element ref='ds:SignatureMethod' minOccurs='1' maxOccurs='1'/>
      <element ref='ds:Reference' minOccurs='1' maxOccurs='unbounded'/>
    </sequence>
    <attribute name='Id' type='ID' use='optional'/>
  </complexType>
</element>
```

DTD:

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

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.

4.3.1 The CanonicalizationMethod Element

CanonicalizationMethod is an optional element [reagle suggests required] that specifies the canonicalization algorithm applied to the SignedInfo element prior to performing signature calculations. This element uses the general structure for algorithms described in section 6.1: Algorithm Identifiers. The default canonicalization algorithm (applied if this element is omitted) is Canonical XML [XML-C14N].

Alternatives, such as the minimal canonicalization algorithm (the CRLF and charset normalization specified in section 6.5.1: Minimal Canonicalization), may be explicitly specified but are NOT REQUIRED.
Consequently, their use may not interoperate with other applications that do not support the specified algorithm (see section 7: XML Canonicalization and Syntax Constraint Considerations). 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).

We RECOMMEND that resource constrained applications that do not implement the Canonical XML [XML-C14N] transform and instead choose minimal canonicalization (or some other form) are implemented to generate Canonical XML as their output serialization to easily mitigate some of these interoperability and security concerns. For instance, such an implementation SHOULD (at least) generate standalone XML instances [XML].

Schema Definition:

```
<element name='CanonicalizationMethod'>
  <complexType content='elementOnly'>
    <any minOccurs='0' maxOccurs='unbounded'/>
    <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 content='elementOnly'>
    <any minOccurs='0' maxOccurs='unbounded'/>
    <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:

```xml
<element name='Reference'>
  <complexType content='elementOnly'>
    <sequence minOccurs='1' maxOccurs='1'>
      <element ref='ds:Transforms' minOccurs='0' maxOccurs='1'/>
      <element ref='ds:DigestMethod' minOccurs='1' maxOccurs='1'/>
      <element ref='ds:DigestValue' minOccurs='1' maxOccurs='1'/>
    </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:

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

The URI attribute identifies a data object using a URI-Reference, as specified by RFC2396 [URI]. (Non-ASCII characters in a URI should be represented in UTF-8 [UTF-8] as one or more bytes, and then escaping these bytes with the URI escaping mechanism. [XML]) Note that a null URI (URI="") is permitted and identifies the XML document that the reference is contained within (the root element). XML Signature applications MUST be able to parse URI syntax. We RECOMMEND they be able to dereference URIs and null URIs in the HTTP scheme. (See the section 3.2.1:Reference Validation for a further comment on URI dereferencing.) Applications should 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.

[URI] permits identifiers that specify a fragment identifier via a separating number/pound symbol ‘#’. (The meaning of the fragment is defined by the resource’s MIME type). XML Signature applications MUST support the XPointer ‘bare name’ [Xptr] shortcut after ‘#’ so as to identify IDs within XML documents. The results are serialized as
specified in section 6.6.3: XPath Filtering. For example,

URI="http://foo.com/bar.xml"
    Identifies the external XML resource 'http://foo.com/bar.xml'.

URI="http://foo.com/bar.xml#chapter1"
    Identifies the element with ID attribute value 'chapter1' of
    the external XML resource 'http://foo.com/bar.xml'.

URI=""
    Identifies the XML resource containing the signature.

URI="#chapter1"
    Identifies the element with ID attribute value 'chapter1' of
    the XML resource containing the signature.

Otherwise, support of other fragment/MIME types (e.g., PDF) or XML
addressing mechanisms (e.g., [XPath, Xptr]) is OPTIONAL, though we
RECOMMEND support of [XPath]. Regardless, such fragment identification
and addressing SHOULD be given under Transforms (not as part of the
URI) so that they can be fully identified and specified. For instance,
one could reference a fragment of a document that is encoded by using
the Reference URI to identify the resource, and one Transform to
specify decoding, and a second to specify an XPath selection.

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 digest algorithm is applied to the data octets being secured.
Typically that is done by locating (possibly using the URI if
provided) the data and transforming it. If the data is an XML
document, the document is assumed to be unparsed prior to the
application of Transforms. If there are no Transforms, then the data
is passed to the digest algorithm unmodified.

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/2000/02/xmldsig#Object"
Type="http://www.w3.org/2000/02/xmldsig#Manifest"
Type="http://www.w3.org/2000/02/xmldsig#SignatureProperty"

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.

4.3.3.1 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 (octets) serves as input to the next Transform. The input to the first Transform is the source data. 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 section 8.2: Only What is "Seen" Should be Signed).

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 input resource, (see section 6.1: Algorithm Identifiers and Implementation Requirements).

Some Transform may require explicit MimeType, 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 information. Such data characteristics are provided as parameters to the Transform algorithm and should be described in the specification for the algorithm.

Schema Definition:

```xml
<element name='Transforms' >
  <complexType content='elementOnly'>
    <element ref='ds:Transform' minOccurs='1' maxOccurs='unbounded'/>
  </complexType>
</element>

<element name='Transform'>
  <complexType content='mixed'>
    <any minOccurs='0' maxOccurs='unbounded'/>
    <element name='Xpath' type='string'/>
    <element name='XSLT' type='string'/>
    <attribute name='Algorithm' type='uriReference' use='required'/>
  </complexType>
</element>
</element>

<!ELEMENT Transforms (Transform+)>
<!ELEMENT Transform %Transform.ANY; >
<!ATTLIST Transform
  Algorithm CDATA #REQUIRED
  MimeType CDATA #IMPLIED
  Charset CDATA #IMPLIED >

<!ELEMENT XPath (#PCDATA) >
<!ELEMENT XSLT (#PCDATA) >
```
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. Section 6.6: Transform Algorithms defines the list of standard transformations.

4.3.3.2 The DigestMethod Element

DigestMethod is a required element that identifies the digest algorithm to be applied to the signed object. This element uses the general structure here for algorithms specified in section 6.1: Algorithm Identifiers.

Schema Definition:

```xml
<element name='DigestMethod'>
  <complexType content='elementOnly'>
    <any minOccurs='0' maxOccurs='unbounded'/>
    <attribute name='Algorithm' type='uriReference' use='required'/>
  </complexType>
</element>
```

DTD:

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

4.3.3.3 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:

```xml
<element name='DigestValue' type='ds:CryptoBinary'/>
```

DTD:

```
<!ELEMENT DigestValue (#PCDATA) >
<!-- base64 encoded signature value -->
```

4.4 The KeyInfo Element

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]

Schema Definition:
KeyInfo is an optional element that enables the recipient(s) to obtain the key(s) needed to validate the signature. If 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 Section 4.4.2: KeyValue and SHOULD implement Section 4.4.3: RetrievalMethod.

### 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.)

### 4.4.2 The KeyValue Element

The KeyValue element contains one or more public keys that may be useful in validating the signature. Structured formats for defining DSA (REQUIRED) and RSA (RECOMMENDED) public keys are defined in Section 6.4: Signature Algorithms.
4.4.3 The RetrievalMethod Element

A RetrievalMethod element within KeyInfo is used to convey a pointer to KeyInfo-like information that is stored at a remote location. For example, an X.509v3 certificate chain may be published somewhere common to a number of documents; each document can reference this chain using a single RetrievalMethod element instead of including the entire chain with a sequence of X509Certificate elements.

Each RetrievalMethod element contains three children elements: Location, Method and Type. Location contains a URI identifying the actual object. Method describes the process by which the data retrieved from the Location URI should be converted into KeyInfo sub-elements. The Type sub-element describes the object type and encoding format of the data stored at the Location URI.

Schema Definition:

```xml
<element name='RetrievalMethod'>
    <complexType content='elementOnly'>
        <sequence minOccurs='1' maxOccurs='1'>
            <element name='Location' type='uriReference' minOccurs='1' maxOccurs='1'/>
            <element name='Method' type='string' minOccurs='1' maxOccurs='1'/>
            <element ref='ds:Type' minOccurs='1' maxOccurs='1'/>
        </sequence>
        <attribute name='Encoding' type='uriReference' use='optional'/>
    </complexType>
</element>
```

DTD:

```xml
<!ELEMENT RetrievalMethod (Location, Method, Type) >
```
4.4.4 The X509Data Element

An X509Data element within KeyInfo contains one or more identifiers of keys/X.509 certificates that may be useful for validation. Five types of X509Data pointers are defined:

1. The X509IssuerSerial element, which contains an X.509 issuer distinguished name/serial number pair,
2. The X509SubjectName element, which contains an X.509 subject distinguished name,
3. The X509SKI element, which contains an X.509 subject key identifier value.
4. The X509Certificate element, which contains a Base64-encoded X.509v3 certificate, and
5. The X509CRL element, which contains a Base64-encoded X.509v2 certificate revocation list (CRL).

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 multiple X509Data elements. For example, the following block contains two pointers to certificate-A (issuer/serial number & SKI) and a single reference to certificate-B (Subject Name):

```xml
<X509Data>
  <X509IssuerSerial>
    <X509IssuerName>My CA for Certificate A</X509IssuerName>
    <X509SerialNumber>12345678</X509SerialNumber>
  </X509IssuerSerial>
  <X509SKI>31d97bd7</X509SKI>
</X509Data>
<X509Data>
  <X509SubjectName>Subject of Certificate A</X509SubjectName>
</X509Data>
```

Schema Definition:

```xml
<element name='X509Data'>
  <complexType content='elementOnly'>
    <choice minOccurs='1' maxOccurs='unbounded'>
      <sequence minOccurs='1' maxOccurs='1'>
        <element ref='ds:X509IssuerSerial'/>
        <element name='X509SKI' type='string'/> <!-- should this be binary -->
      </sequence>
    </choice>
    <element name='X509SubjectName' type='string'/>
  </complexType>
</element>
```
4.4.5 The PGPData element

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 Section 11.2 of [PGP]. The PGPKeyPacket contains a base64-encoded Key Material Packet as defined in Section 5.5 of [PGP]. Other sub-types of the PGPData element may be defined by the OpenPGP working group.

Schema Definition:

```
<element name='PGPData'>
    <complexType content='elementOnly'>
        <sequence minOccurs='1' maxOccurs='1'>
            <any namespace='##other' minOccurs='1' maxOccurs='unbounded'/>
            <element name='PGPKeyID' type='string' minOccurs='1' maxOccurs='1'/>"
        </sequence>
    </complexType>
</element>
```

DTD:

```xml
<!ELEMENT PGPData ((PGPKeyID), (PGPKeyPacket*))>
<!ELEMENT PGPKeyID (#PCDATA) >
<!ELEMENT PGPKeyPacket (#PCDATA) >
```
4.4.6 The SPKIData element

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 open and can be defined elsewhere. Schema Definition:

```xml
<element name='SPKIData'>
  <complexType content='elementOnly'>
    <any namespace='##other' minOccurs='1' maxOccurs='unbounded'/>
  </complexType>
</element>
```

DTD:

```xml
<!ELEMENT SPKIData (#PCDATA) >
```

4.4.6 The MgmtData element

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:

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

DTD:

```xml
<!ELEMENT MgmtData (#PCDATA)>
```

4.5 The Object Element

The type identifier (that can be used within Reference) is:

```
Type="http://www.w3.org/2000/02/xmldsig#Object"
```

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 document. The digest is calculated over the entire Object element including start and end tags.
Note, if the application wishes to exclude the <Object> tags from the
digest calculation the Reference must identify the actual data object
(easy for XML documents) or a transform must be used to remove the
Object tags (likely where the data object is non-XML). Exclusion of
the object tags may be desired for cases where one wants the signature
to remain valid if the data object is moved from inside a signature to
outside the signature (or vice-versa), or where the content of the
Object is an encoding of an original binary document and it is desired
to extract and decode so as to sign the original bitwise
representation.

Schema Definition:

```xml
<element name='Object' >
  <complexType content='mixed'>
    <element ref='ds:Manifest' minOccurs='1' maxOccurs='unbounded'/>
    <any namespace='##any' minOccurs='1' maxOccurs='unbounded'/>
    <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:

```xml
<!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

The type identifier (that can be used within Reference) is:

```xml
Type="http://www.w3.org/2000/02/xmldsig#Manifest"
```

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
application discretion. If a Manifest is referenced from another
Manifest, even the overall digest of this two level deep Manifest might not be checked.

Schema Definition:

```xml
<element name='Manifest'>
  <complexType content='elementOnly'>
    <sequence minOccurs='1' maxOccurs='1'>
      <element ref='ds:Reference' minOccurs='1' 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

The type identifier (that can be used within Reference) is:

```
Type="http://www.w3.org/2000/02/xmldsig#SignatureProperty"
```

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:

```xml
<element name='SignatureProperties'>
  <complexType content='elementOnly'>
    <element ref='ds:SignatureProperty' minOccurs='1' maxOccurs='unbounded'/>
    <attribute name='Id' type='ID' use='optional'/>
  </complexType>
</element>
```

```xml
<element name='SignatureProperty'>
  <complexType content='mixed'>
    <any namespace='##other' minOccurs='1' maxOccurs='unbounded'/>
    <attribute name='Target' type='uriReference' use='required'/>
  </complexType>
</element>
```

DTD:

```
<!ELEMENT SignatureProperties (SignatureProperty+) >
<!ATTLIST SignatureProperties
  Id ID #IMPLIED >
<!ELEMENT SignatureProperty %Object.ANY; >
<!ATTLIST SignatureProperty
  Target CDATA #REQUIRED >
```
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, they will be signed. Consequently, 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 standard. 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 (or SignedInfo directly if there is no 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.

<table>
<thead>
<tr>
<th>Algorithm Type</th>
<th>Algorithm Requirements</th>
<th>Algorithm URI</th>
</tr>
</thead>
<tbody>
<tr>
<td>Digest</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
SHAl REQUIRED  http://www.w3.org/2000/02/xmldsig#sha1
Encoding
  Base64 REQUIRED  http://www.w3.org/2000/02/xmldsig#base64
MAC
  HMAC-SHA1 REQUIRED  http://www.w3.org/2000/02/xmldsig#hmac-sha1
Signature
  DSAwithSHA1 (DSS) REQUIRED  http://www.w3.org/2000/02/xmldsig#dsa
  RSAwithSHA1 RECOMMENDED  http://www.w3.org/2000/02/xmldsig#rsa-sha1
Canonicalization
  minimal RECOMMENDED  http://www.w3.org/2000/02/xmldsig#minimal
  XML-
  Canonicalization REQUIRED  
  http://www.w3.org/TR/1999/WD-xml-c14n-19991115
Transform
  XSLT OPTIONAL  http://www.w3.org/TR/1999/REC-xslt-19991116
  XPath RECOMMENDED  http://www.w3.org/TR/1999/REC-xpath-19991116

Note that the normative identifier is the complete URI in the table though they are frequently abbreviated in XML syntax as "&dsig;base64" or the like.

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/2000/02/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 or SignedInfo directly if there is no
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/2000/02/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:

```xml
<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 158BFC9D from the test vectors in [HMAC] would be

```xml
<SIGNATUREVALUE>kRyejY4uxwT9I74FYv8nQ==</SIGNATUREVALUE>
```

Schema Definition:

```xml
<element name='HMACOUTPUTLENGTH' type='integer' minOccurs='0' maxOccurs='1'/>
```

DTD:

```xml
<!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 or SignedInfo directly if there is no CanonicalizationMethod. Signature and MAC algorithms are syntactically identical but a signature implies public key cryptography.

6.4.1 DSA

Identifier: http://www.w3.org/2000/02/xmldsig#dsa

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

```xml
<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:

\[
\begin{align*}
r &= 88AC1AB6 \ 6410435C \ B7181F95 \ B16AB97C \ 92B341C0 \\
s &= 41E2345F \ 1F56DF24 \ 58F426D1 \ 55B4BA2D \ B6DCD8C8
\end{align*}
\]

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

\[
<\text{SignatureValue}>
\text{i6watmQQQ1y3GB+VsWq5fJKzQcBB4jRfH1bfJFj0JtFVyLotttzYyA==}</SignatureValue>
\]

DSA key values have the following set of fields: \( P, Q, G \) and \( Y \) are mandatory when appearing as a key value, \( J \), \( \text{seed} \) and \( \text{pgenCounter} \) are optional but SHOULD be present. (The \( \text{seed} \) and \( \text{pgenCounter} \) fields MUST appear together or be absent). All parameters are encoded as base64 values.

Schema:

```xml
<element name='DSAKeyValue'>
  <complexType content='elementOnly'>
    <sequence minOccurs='1' maxOccurs='1'>
      <element name='P' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
      <element name='Q' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
      <element name='G' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
      <element name='Y' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
      <element name='J' type='ds:CryptoBinary' minOccurs='0' maxOccurs='1'/>
    </sequence>
    <sequence minOccurs='0' maxOccurs='1'>
      <element name='Seed' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
      <element name='PgenCounterQ' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
    </sequence>
  </complexType>
</element>
```

DTD:

```xml
<!ELEMENT DSAKeyValue (P, Q, G, Y, J?, (Seed, PgenCounter)?) >
<!ELEMENT P (#PCDATA) >
<!ELEMENT Q (#PCDATA) >
<!ELEMENT G (#PCDATA) >
<!ELEMENT Y (#PCDATA) >
<!ELEMENT J (#PCDATA) >
<!ELEMENT Seed (#PCDATA) >
<!ELEMENT PgenCounter (#PCDATA) >
```
6.4.2 PKCS1

Identifier: http://www.w3.org/2000/02/xmldsig#rsa-sha1

The expression "RSA algorithm" as used in this specification refers to the RSASSA-PKCS1-v1_5 algorithm described in RFC 2437 [PKCS1]. The RSA algorithm takes no explicit parameters. An example of an RSA SignatureMethod element is:

```xml
<SignatureMethod Algorithm="&dsig;rsa-sha1"/>
```

The output of the RSA algorithm is an octet string. The SignatureValue content for an RSA signature shall be the base64 encoding of this octet string. [Example: TBD]

RSA key values have two fields: Modulus and Exponent.

Schema:

```xml
<element name='RSAKeyValue'>
  <complexType content='elementOnly'>
    <element name='Modulus' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
    <element name='Exponent' type='ds:CryptoBinary' minOccurs='1' maxOccurs='1'/>
  </complexType>
</element>
```

DTD:

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

6.5 Canonicalization Algorithms

Canonicalization algorithms take one implicit parameter when they appear as a CanonicalizationMethod within the SignedInfo element.

6.5.1 Minimal Canonicalization

Identifier: http://www.w3.org/2000/02/xmldsig#minimal

An example of a minimal canonicalization element is:

```xml
<CanonicalizationMethod Algorithm="&dsig;minimal"/>
```

The minimal canonicalization algorithm:
* converts the character encoding to UTF-8, removing the encoding pseudo-attribute
* normalizes line endings as provided by [XML]. (See section 7: XML and Canonicalization and Syntactical Considerations.)

6.5.2 Canonical XML
An example of an XML canonicalization element is:

```
<CanonicalizationMethod Algorithm="http://www.w3.org/TR/1999/WD-xml-c14n-19991115"/>
```

The normative specification of Canonical XML is [XML-C14N].

### 6.6 Transform Algorithms

A Transform algorithm has three implicit parameters. The first is an octet stream from the Reference or the output of an earlier Transform. The second and third are the optional MimeType and Charset attributes that can be specified on the Transform element.

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. The Working Group goal is to maximize application interoperability on XML signatures, and the working group expects ubiquitous availability of software to support these transforms that can be incorporated into applications without extensive development.

#### 6.6.1 Canonicalization

Any canonicalization algorithm that can be used for CanonicalizationMethod can be used as a Transform.

#### 6.6.2 Base64 and Quoted-Printable Decoding

Identifiers:

- http://www.w3.org/2000/02/xmldsig#base64
- http://www.w3.org/2000/02/xmldsig#qp

The normative specification for base 64 and quoted-printable decoding transforms is [MIME]. Neither the base64 nor the quoted-printable Transform element has 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. Quoted-printable is provided, in addition to base64, in keeping with the XML support of a roughly human readable final format.

#### 6.6.3 XPath Filtering

Identifier:

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

The XPath transform output is the result of applying an XPath expression to an input string. The XPath expression appears in a parameter element named XPath. The input string is equivalent to the result of dereferencing the URI attribute of the Reference element containing the XPath transform, then, in sequence, applying all
transforms that appear before the XPath transform in the Reference element’s Transforms.

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. The XPath expression can be created such that it includes all elements except those meeting specific criteria. It is the responsibility of the XPath expression author to ensure that all necessary information has been included in the output such that modification of the excluded information does not affect the interpretation of the output in the application context. One simple example of this is the omission of an enveloped signature from a DigestValue calculation.

6.6.3.1 Evaluation Context Initialization

The XPath transform establishes the following evaluation context for the XPath expression given in the XPath parameter element:

* A context node, initialized to the input XML document’s root node.
* A context position, initialized to 1.
* A context size, initialized to 1.
* A library of functions equal to the function set defined in XPath.
* 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.

6.6.3.2 Parsing Requirements for XPath Evaluation

An XML processor is used to read the input XML document and produce a parse tree capable of being used as the initial context node for the XPath evaluation, as described in the previous section. If the input is not a well-formed XML document, then the XPath transform must throw an exception.

Validating and non-validating XML processors only behave in the same way (e.g. with respect to attribute value normalization and entity reference definition) until an external reference is encountered. If the XPath transform implementation uses a non-validating processor, and it encounters an external reference in the input document, then an exception must be thrown to indicate that the necessary algorithm is unavailable (The XPath transform cannot simply generate incorrect output since many applications distinguish an unverifiable signature from an invalid signature).

As a result of reading the input with an XML processor, linefeeds are normalized, attribute values are normalized, CDATA sections are replaced by their content, and entity references are recursively replaced by substitution text. In addition, consecutive characters are grouped into a single text node.
The XPath implementation is expected to convert the information in the input XML document and the XPath expression string to the character domain prior to making any comparisons such that the result of evaluating the expression is equivalent regardless of the initial encoding of the input XML document and XPath expression.

The namespace prefix of each node appearing in the original document must be preserved by the XML processor used by the XPath transform implementation. This is necessary in order to produce the serialized result.

Although an node-set is unordered, based on the expression evaluation requirements of the XPath function library, the document order position of each node must be available, except for the attribute and namespace axes. The XPath transform imposes no order on attribute and namespace nodes during XPath expression evaluation, and expressions based on attribute or namespace node position are not interoperable. The XPath transform does define an order for namespace and attribute nodes during serialization.

### 6.6.3.3 XPath Transform Serialization Algorithm

A node-set is converted into a string by generating the representative text for each node in the node-set. The nodes of a node-set are processed in ascending order of the nodes’ document order positions except for attribute and namespace nodes, which do not have document order positions.

The nodes in the attribute and namespace axes will each be processed in lexicographic order, with the namespace axis preceding the attribute axis. Lexicographic comparison is performed using namespace URI as the primary key and local name as secondary key (namespace nodes and attribute nodes with no namespace qualification have an empty namespace URI, which is defined to be lexicographically least). Lexicographic comparison is based on the UCS codepoint values, which is equivalent to lexical ordering based on UTF-8.

The method of text generation is dependent on the node type and given in the following list:

- **Root Node**- Nothing (no byte order mark, no XML declaration, no document type declaration).
- **Element Nodes**- An open angle bracket (<>), the element QName, the nodes of the namespace axis, the nodes of the attribute axis, a close angle bracket (>), the descendant nodes of the element that are in the node-set (in document order), an open angle bracket, a forward slash (/), the element QName, and a close angle bracket.
- **Namespace and Attribute Nodes**- a space, the node’s QName, an equals sign, an open double quote, the modified string value, and a close double quote. The string value of the node is modified by replacing all ampersands (&) with &amp;, all double quote characters with &quot;, and all whitespace characters (#x9, #xA, #xD, and #x20) with two digit hexadecimal character references except for #x20 characters with no preceding #x20.
* Text Nodes- the string value, except all ampersands are replaced by &amp;, all open angle brackets (&lt;) are replaced by &lt;, and all #xD characters are replaced by &#xD;.
* Processing Instruction Nodes- an open angle bracket, a question mark, the PI target name of the node, a leading space and the modified string value if the string value is not empty, the question mark, and a close angle bracket. The modified string value is the string value except all #xD characters are replaced by &#xD;. If the string value is empty, then the leading space is not added.
* Comment Nodes- the open comment sequence (<!--), the string value of the node, and the close comment sequence (-->).

The QName of a node is either the local name if the namespace prefix string is empty or the namespace prefix, a colon, then the local name of the element. The namespace prefix used in the QName MUST be the same one which appeared in the input document.

6.6.3.4 XPath Transform Output

The result of the XPath expression is a string, boolean, number, or node-set. If the result of the XPath expression is a string, then the string converted to UTF-8 is the output of the XPath transform. If the result is a boolean or number, then the XPath transform output is computed by converting the boolean or number to a string as if by a call to the XPath string() function, then converting to UTF-8. If the result of the XPath expression is a node-set, then the XPath transform result is computed by serializing the node-set with a UTF-8 encoding.

For example, consider creating an enveloped signature S1 (a Signature element with an id attribute equal to "S1"). The signature S1 is enveloped because its Reference URI indicates some ancestor element of S1. Elements within S1 are changed during signature creation (e.g. the digest value must be put inside the DigestValue and S1’s SignatureValue must be subsequently calculated). To prevent these changes from invalidating the digest value in DigestValue, we add a transform that omits S1 from the digest calculation. This can be done with an XPath transform containing the following XPath expression in its XPath parameter element:

/descendant-or-self::node()[not(ancestor-or-self::dsig:Signature[@id='S1'])]

The ‘/descendant-or-self::node()’ means that all nodes in the entire parse tree starting at the root node are candidates for the result node-set. For each node candidate, the node is included in the resultant node-set if and only if the node test (the boolean expression in the square brackets) evaluates to "true" for that node. The node test returns true for all nodes except S1 and its descendant nodes.

Note that this expression works even if the XPath transform is implemented with a non-validating processor because S1 is identified
by comparison to the value of an attribute named ‘id’ rather than by using the XPath id() function. Although the id() function is useful when the ‘id’ attribute is not named ‘id’, the XPath expression author will know the ‘id’ attribute’s name when writing the expression.

It is RECOMMENDED that the XPath be constructed such that the result of this operation is a well-formed XML document. This should be the case if root element of the input resource is included by the XPath (even if a number of its descendant nodes are omitted by the XPath expression). It is also RECOMMENDED that nodes should not be omitted from the input if they affect the interpretation of the output nodes in the application context. The XPath expression author is responsible for this since the XPath expression author knows the application context.

6.6.4 XSLT Transform

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

The Transform element contains a single parameter child element called XSLT, whose content MUST conform to the XSL Transforms [XSLT] language syntax. The processing rules for the XSLT transform are stated in the XSLT specification [XSLT].

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 data object 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 the W3C Canonical XML [XML-C14N], that coded character set is UTF-8.

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, [I18N:In 7.1, second numbered list, item 2, characters not representable in the encoding chosen should be mentioned.]
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, and

Note that items (2), (4), (5C), and (6) depend on specific schema, DTD, or similar declarations. In the general case, such declarations will not be available to or used by the signature verifier. Thus, to interoperate between different XML implementations, the following syntax constraints MUST be observed when generating any signed material to be processed as XML, including the SignedInfo element:
1. attributes having default values be explicitly present,
2. all entity references (except "amp", "lt", "gt", "apos", and "quot" which are pre-defined) 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 syntax constraints given in section 7.1 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 Only What is Signed is Secure

A requirement of this specification is to permit signatures to "apply to a part or totality of a XML document." (See section 3.1.3 of [XML-Signature-RD].) 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 [XPath] to exclude those portions the user needs to change. Transforms may be arbitrarily specified and may include canonicalization instructions or even XSLT transformations. Of course, signatures over such a derived document do not secure any information discarded by the Transforms.

Furthermore, 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.

8.2 Only What is "Seen" Should be Signed

If signing is intended to convey the judgment or consent of an automated mechanism or person, then it is normally necessary to secure as exactly as practical the information that was presented to that mechanism or person. 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.

Also 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.

8.3 Check the Security Model

This standard specifies 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 standard 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.
8.4 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 validators 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 Example

XML Signature Schema Instance
xmlsig-core-schema.xsd
XML schema instance based on the Last Call 20000407 Schema/DTD [XML-Schema].

XML Signature DTD
xmlsig-core-schema.dtd

RDF Data Model
xmlsig-datamodel-20000112.gif

XML Signature Object Example
signature-example.xml
A cryptographically 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 schema is modified accordingly).

XML RSA Signature Valid Example
signature-example-rsa.xml
An XML Signature example by Petteri Stenius that has generated cryptographic values and validates under the schema.

XML DSA Signature Valid Example
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, 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 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
http://www.w3.org/TR/1998/REC-DOM-Level-1-19981001/

DOMHASH

DSS
FIPS PUB 186-1. Digital Signature Standard (DSS). U.S.
Department of Commerce/National Institute of Standards and Technology.

HMAC

HTTP

KEYWORDS
RFC2119 Key words for use in RFCs to Indicate Requirement Levels. S. Bradner. March 1997.
MD5

MIME

PGP

RANDOM

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

PKCS1

SAX
http://www.megginson.com/SAX/index.html

SHA-1

UTF-8

URI

URL

URN

XLink
http://www.w3.org/1999/07/WD-xlink-19990726

XML
http://www.w3.org/TR/1998/REC-xml-19980210

XML-C14N
http://www.w3.org/TR/1999/WD-xml-c14n-19991115

XML-ns
http://www.w3.org/TR/1999/REC-xml-names-19990114

XPath
http://www.w3.org/TR/1999/PR-xpath-19991008

XPointer
http://www.w3.org/1999/07/WD-xptr-19990709

XML-schema
http://www.w3.org/TR/2000/WD-xmlschema-1-20000407/
http://www.w3.org/TR/2000/WD-xmlschema-2-20000407/

XML-Signature-RD
Will be RFC 2807. XML Signature Requirements. J. Reagle, April 2000.
http://www.w3.org/TR/xmldsig-requirements

XSL
http://www.w3.org/TR/2000/WD-xsl-20000327/xslspec.html

XSLT

Eastlake, Reagle, Solo
http://www.w3.org/TR/1999/REC-xslt-19991116.html

WebData
http://www.w3.org/1999/04/WebData

12. Authors’ Address

Donald E. Eastlake 3rd
Motorola, Mail Stop: M4-10
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
666 Fifth Ave, 3rd Floor
NY, NY 10103 USA
Phone: +1-212-830-8118
Email: dsolo@alum.mit.edu