A Stream-based Implementation of XML Encryption

Takeshi Imamura
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242-8502, Japan
+81-46-215-4479
imamu@jp.ibm.com

Andy Clark
977 Hollow Creek Dr., Milford, OH 45150, USA
+1-513-248-4169
andyc@apache.org

Hiroshi Maruyama
IBM Research, Tokyo Research Laboratory
1623-14, Shimotsuruma, Yamato, Kanagawa 242-8502, Japan
+81-46-215-4576
maruyama@jp.ibm.com

ABSTRACT
W3C has been working on the standardization of XML Encryption and released its specification as a W3C Proposed Recommendation in 2002. There are several implementations of the specification, all of which are implemented using DOM. However, it is commonly accepted that DOM has higher costs in time and space than other APIs. Also, even if SAX is used, with this kind of API, it is impossible to parse decrypted data both efficiently and correctly. Therefore, we thought of using the Xerces Native Interface (XNI) of Xerces2. Using this API, we prototyped a stream-based implementation of the specification. We also evaluated its performance. As compared with a DOM-based implementation, it achieves a 0.27%-26% reduction in processing time (i.e., 1.0x-1.3x performance) for encryption of XML documents with sizes larger than 2 KB, and 34%-88% (i.e., 1.5x-8.5x) for decryption of XML documents with any sizes.

Categories and Subject Descriptors
E.3 [**Data**]: Data Encryption – *Standards*

General Terms
Security, Standardization

Keywords
XML, encryption, stream-based processing

1. INTRODUCTION
1.1 Motivation
XML [1] is a language for representing tree-structured data and was standardized by W3C [2] in 1998. Because data represented in XML is just clear text, it is difficult to use such data, especially for business, unless some security is provided. For such reasons, the joint standardization of XML Signature [3] was started by W3C and IETF [4] in 1999. The specification for XML Signature became a W3C Recommendation in 2002. When this standardization was almost completed, the standardization of XML Encryption [5] was started by W3C in 2000. The specification for XML Encryption became a W3C Proposed Recommendation in 2002.

ACM Workshop on XML Security, Nov. 22, 2002, Fairfax VA, USA.

Because at least two independent and interoperable implementations are required for a specification to proceed to a W3C Recommendation, an interoperability test was done. According to this report [6], there are four implementations at present. Based on their API documentation, all of them are implemented using DOM [7]. Though DOM makes it possible to manipulate an XML document easily, it is commonly accepted that DOM has higher costs in time and space than other APIs. Because they are reference implementations of the specification and are intended for verifying whether it really works, efficient performance is not crucial. However, performance is very important for certain applications, such as online transactions.

One of the methods to improve performance is to process data as a stream. With this kind of processing, the costs in time and space are reduced and accordingly performance is improved. Actually, it is possible to process data as a stream if the encryption algorithms are limited to block encryption (e.g., Triple DES [8]), stream encryption (e.g., RC4 [9]), and similar.

We have SAX [10] as a stream-based API and using it, it would be possible to implement the specification. However, the use of APIs such as DOM and SAX raises an issue. With this kind of API, it is impossible to parse decrypted data both efficiently and correctly. The reason is that though it is very likely that decrypted data is a part of an XML document, the method to parse it directly is not defined anywhere. The XML specification is not helpful because it only defines how to parse a complete XML document, and therefore other methods have to be used. One of them is to serialize an XML document, replacing an encrypted part with its decrypted data, and then reparsing the XML document. Though this method would work, it is not efficient if an XML document is very large but decrypted data is very small. A better method is to parse decrypted data as if it is parsed as a part of an XML document. This method recreates the parsing context of the decrypted data. In the XML Encryption specification, namespace declarations [11] and general entities [1] are considered as the parsing context. These would be sufficient for most cases, and it is expected that this method is adopted in the implementations mentioned above. However, it is a temporary solution and is not regarded as correct.

This issue arises from the use of APIs such as DOM and SAX. In other words, it arises because we try to decrypt a part of an XML

document and then parse the resulting data at the application level. If it is possible to do so at the parser level, the issue would never arises. Actually, this is possible with Xerces2 [12], an XML parser developed by Apache XML Project [13]. It has an extensible architecture that makes it possible to place any parser components in a pipeline of components of which it consists. Also, these components are stream-based. These features are sufficient for our requirements.

1.2 Related Work

As mentioned in Section 1.1, according to the interoperability report of the XML Encryption specification, there are four implementations at present. However, based on their API documentation, all of them are implemented using DOM. This means that neither of them addresses the issues we present, and therefore we believe that our work is worthwhile.

1.3 Achievements

We prototyped a stream-based implementation of the XML Encryption specification as parser components of Xerces2. We also evaluated its performance. As compared with a DOM-based implementation, it achieves a 0.27%-26% reduction in processing time (i.e., 1.0x-1.3x performance) for encryption of XML documents with sizes larger than 2 KB, and 34%-88% (i.e., 1.5x-8.5x) for decryption of XML documents with any sizes.

1.4 Organization

In Sections 2 and 3, we outline the specification of XML Encryption and the design of Xerces2, respectively. In Section 4, we describe the design and development environment of our implementation. In Section 5, we present and discuss its performance through an experiment. Finally, in Section 6, we present a summary of this paper and future work.

2. XML ENCRYPTION

The XML Encryption specification [5] was released as a W3C Proposed Recommendation in 2002. It specifies (1) steps for encrypting data, (2) steps for decrypting encrypted data, and (3) the syntax in XML for representing encrypted data and the information used for decrypting it. XML Encryption can be applied to an XML element, XML element content, and arbitrary data (including an XML document). In this section, we outline the specification.

2.1 Syntax

The encrypted data is represented as an `EncryptedData` element, whose syntax is illustrated in Figure 1. In this figure, it is assumed that prefixes "`e`" and "`ds`" are associated with URI references [14] of the XML Encryption and XML Signature namespaces, respectively, i.e.:

```
xmlns:e="http://www.w3.org/2001/04/xmlenc#"
xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
```

2.2 Processing Rules

The steps to encrypt data are as follows:

- For each data item to be encrypted:
  1. Select the algorithm and parameters.

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

Figure 1. `EncryptedData` element

  2. Obtain the key. If the key itself is to be encrypted, construct an `EncryptedKey` element by applying these steps recursively.
  3. Encrypt the data. If it is of type 'element' or element 'content', serialize it in UTF-8 [15] first.
  4. Construct an `EncryptedData` element.
  5. Return the `EncryptedData` element. If the data is of type 'element' or element 'content' and is required to be replaced, replace it with the `EncryptedData` element.

The steps to decrypt the encrypted data are as follows:

- For each `EncryptedData` element to be decrypted:
  1. Identify the algorithm, parameters, and the `KeyInfo` element.
  2. Locate the key according to the `KeyInfo` element. If the key itself is encrypted, locate the corresponding key to decrypt it.
  3. Decrypt the data contained in the `CipherData` element.
  4. Return the decrypted data. If it is of type 'element' or element 'content' and the `EncryptedData` element is required to be replaced, replace the `EncryptedData` element with the decrypted data.

3. XERCES2

Xerces2 [12] is an XML parser developed by Apache XML Project. It has an extensible architecture that makes it possible to build any parser components and configurations. Our stream-based implementation of the XML Encryption specification, which is described in the next section, depends on it. In this section, we outline its design.

3.1 Architecture

In Xerces2, a parser is configured as a pipeline of parser components. Each component is either capable of producing data, consuming data, or both. The input data flows through this pipeline to produce some kind of programming interface as the output. For example, it could be a DOM tree or a series of SAX events.

A pipeline consists of a source, zero or more filters, and a target. The source is typically the XML scanner; filters are DTD [1] and XML Schema [16] valildators, the namespace binder, and similar; and the target is the parser to produce a programming interface such as DOM or SAX. A basic pipeline configuration is illustrated in Figure 2.

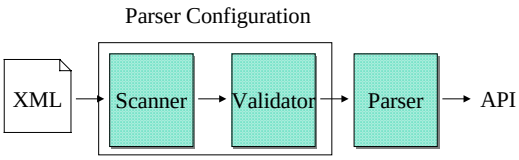

Figure 2. Basic pipeline configuration

The component manager is responsible for configuring a parser. More precisely, it does the following:

- Keeps track of parser settings and options.
- Instantiates and configures various components in a parser.
- Assembles a pipeline and initiates parsing.

What we want to do here is to place our own components for encryption and decryption in a pipeline. Therefore, all we have to do is to build those components and a component manager that places them in the pipeline.

3.2 Xerces Native Interface (XNI)

Parser components communicate with each other using a set of interfaces, called the Xerces Native Interface (XNI). The data communicated through XNI is a streaming XML document information set, which is the information obtained by parsing an XML document in a serial manner. While XNI is similar to SAX, it is different in several ways:

- XNI attempts to provide lossless communication of the streaming information set. For example, XNI passes an XML declaration, text declarations, encodings of external parsed entities, parameter entities, and so forth, which are lost when using SAX.
- XNI makes it possible to modify and augment the streaming information set in each component, whereas SAX is primarily read-only.

XNI breaks the streaming information set into several more manageable information sets: XML document structure and content information, basic DTD information, element declaration's content model information, and so forth, each of which is communicated through a different interface. For example, the interface for the XML document structure and content information, called `XMLDocumentHandler`, includes methods illustrated in Figure 3.

4. STREAM-BASED IMPLEMENTATION

Using the extensible architecture of Xerces2, we prototyped a stream-based implementation of the XML Encryption specification. In this section, we describe its design and development environment.

```
void startDocument(XMLLocator locator,
  String encoding,
  Augmentations augs)
void xmlDecl(String version,
  String encoding,
  String standalone,
  Augmentations augs)
void doctypeDecl(String rootElement,
  String publicId,
  String systemId,
  Augmentations augs)
void startPrefixMapping(String prefix,
  String uri,
  Augmentations augs)
void startElement(QName element,
  XMLAttributes attributes,
  Augmentations augs)
void startGeneralEntity(String name,
  XMLResourceIdentifier identifier,
  String encoding,
  Augmentations augs)
...
```

Figure 3. `XMLDocumentHandler` interface

4.1 Architecture

Each function for encryption and decryption is implemented as a parser component of Xerces2. The component receives a series of XNI events from an upper-level component. If the component for encryption (say, an encryptor) finds any elements to be encrypted in the series, it encrypts them or their contents and then sends the results as XNI events to a lower-level component. On the other hand, if the component for decryption (say, a decryptor) finds any `EncryptedData` elements to be decrypted in the series, it decrypts them and then sends the results also as XNI events to a lower-level component. Both the encryptor and the decryptor send the other XNI events in the series to lower-level components as they are. The pipeline containing the encryptor and that containing the decryptor are illustrated in Figures 4 and 5, respectively. In these figures, the encryptor is placed after the validator while the decryptor is placed before the validator. We note that this placement does not always have to be used though we believe that it is quite natural approach. If necessary, they can be placed anywhere else, as outlined in the previous section.

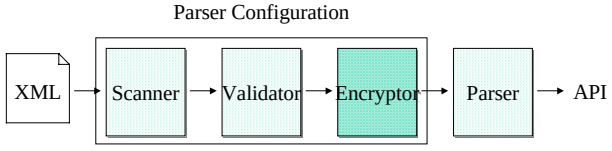

Figure 4. Pipeline containing encryptor

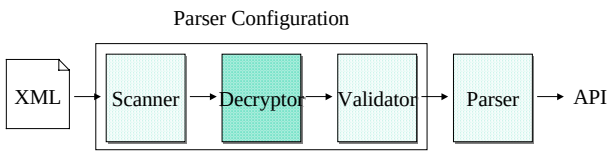

Figure 5. Pipeline containing decryptor

4.2 Processing of the Components

The processing of each component is described in more detail here. The encryptor watches each XNI event corresponding to an element in a series of XNI events received from an upper-level component and checks whether it is an element to be encrypted. If the XNI event is not such an element, it is sent to a lower-level component as it is. Otherwise, a subcomponent that is responsible for encryption (say, an encryptor body) is created and the XNI event and the following related XNI events are sent to it instead. Before those XNI events are sent, the encryptor body sends a series of XNI events corresponding to the part between the start tag of an `EncryptedData` element and that of a `CipherValue` element to the lower-level component. Then, the encryptor body serializes each XNI event received from the upper-level component, encrypts the resulting plaintext, and sends XNI events corresponding to the resulting ciphertext to the lower-level component. When all of the XNI events to be encrypted have been sent, the encryptor body sends an XNI event corresponding to the still remaining ciphertext, if any, and a series of XNI events corresponding to the part between the end tag of the `CipherValue` element and that of the `EncryptedData` element to the lower-level component. Finally, the encryptor body is discarded and the following XNI events are sent directly to the lower-level component. If an XNI event corresponding to another element to be encrypted is found while a series of XNI events are being encrypted, another encryptor body is created and the XNI event and the following related XNI events are sent to it instead. Therefore, multiple encryptor bodies may be chained. This case is illustrated in Figure 6.

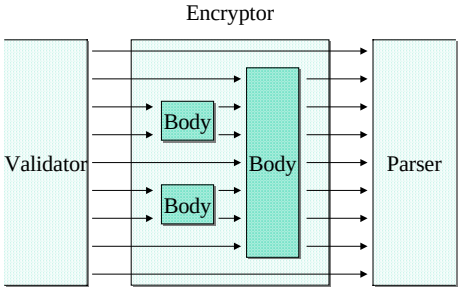

Figure 6. Chained encryptor bodies

In a similar manner, the decryptor watches each XNI event corresponding to an element in a series of XNI events received from an upper-level component and checks whether it is an `EncryptedData` element to be decrypted. If the XNI event is not such an `EncryptedData` element, it is sent to a lower-level component as it is. Otherwise, a subcomponent that is responsible for decryption (say, a decryptor body) is created (or its state is reset if it has been already created) and the XNI event and the following related XNI events are sent to it instead. The decryptor body buffers those XNI events, if necessary, and when all of the XNI events to be decrypted are received, decrypts the buffered ciphertext. More precisely, the ciphertext is not actually decrypted at this time, but just wrapped within a wrapper in which it will be decrypted when the wrapper is asked to return the resulting plaintext. The wrapper is then pushed on the top of the entity stack of a parser. Because the parser always reads data from the top of the entity stack, it reads the plaintext. Consequently, the plaintext is parsed in an appropriate context. Finally, the decryptor body is discarded and the following XNI events are sent directly to the lower-level component.

4.3 Environment

According to the design described above, we prototyped an implementation with Java. We used the following software:

- Java 2 SDK 1.3.1 [17]
- Java Cryptography Extension (JCE) 1.2.1 [18]
- Xerces2 Java Parser 2.0.1 [12]
- XML Security Suite [19]

5. PERFORMANCE EVALUATION

We performed an experiment to evaluate the performance of our stream-based implementation against that of a DOM-based implementation. In this section, we present the conditions, environment, and results, and discuss what influenced the results.

5.1 Conditions

The scenario of the experiment was that an XML document was parsed, its root element was encrypted or decrypted, and the root element was replaced with the resulting element. For XNI, the root element was detected by evaluating an XPath-like [20] expression (e.g., "`/foo`") for each XNI event, and for DOM, it was extracted through the API of DOM. The times for parsing and for encryption or decryption with replacement were measured separately. An XML document before encryption had a balanced binary tree structure and contained text only in leaf elements. Various sizes of XML documents were used. The size of an XML document before encryption varied from 60 B to 1.6 MB, and that after encryption, from 350 B to 2.1 MB. We note that if an XML document is encrypted and the resulting ciphertext is contained in it, its size generally gets larger, because it contains an `EncryptedData` element and the base64-encoded [21] ciphertext. For simplicity, validation was not performed. Serialization was performed using the serializer of Xerces2 as it was for DOM and with some extension for XNI, because it does not provide any API for XNI. Triple DES [8] was used as an encryption algorithm. The key was generated and given in advance so as to avoid generating or retrieving it each time. The encrypted data was contained in the `CipherValue` element.

5.2 Environment

The experiment was done on the following environment:

- CPU: Pentium III 1 GHz
- Memory: 512 MB
- OS: Windows 2000
- Java VM: Sun HotSpot Client VM [17]
- JCE provider: IBM [22]
- DOM-based implementation: XML Security Suite [19]

5.3 Results

The results in encryption and decryption are illustrated in Figures 7 and 8, respectively. In these figures, the horizontal axis

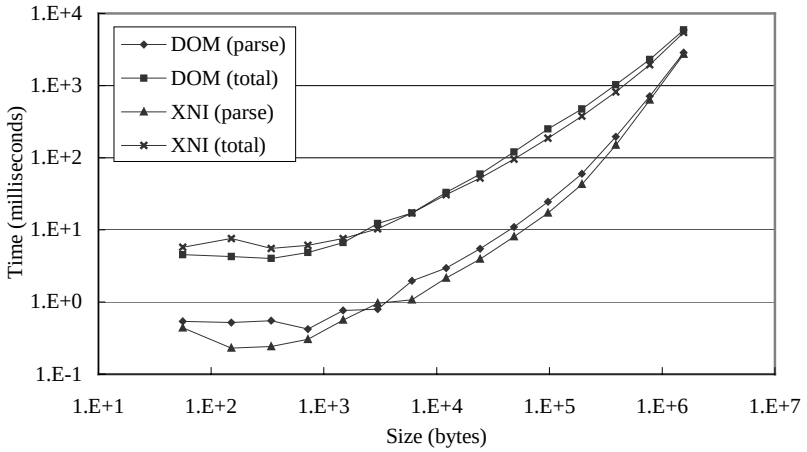

Figure 7. Result in encryption

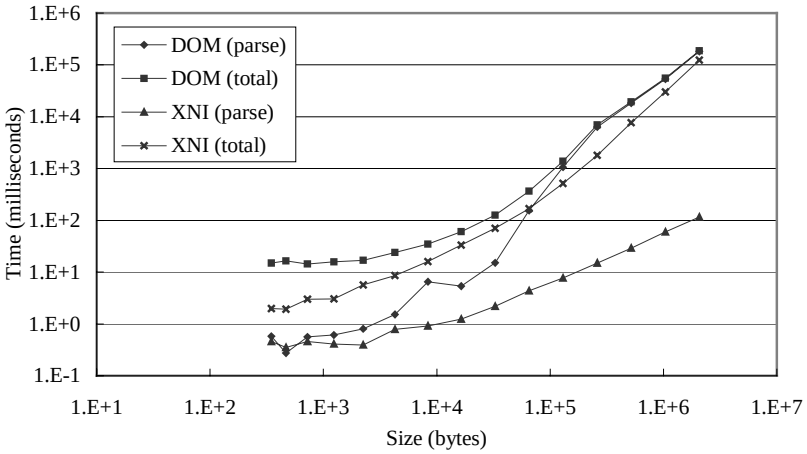

Figure 8. Result in decryption

represents the size of an XML document in bytes, and the vertical axis, the time for processing of an XML document in milliseconds. Each dot in Figure 7 corresponds to the dot at the same position in Figure 8. That is, the XML document represented by a dot in Figure 8 is the one obtained by encrypting the XML document represented by the dot at the same position in Figure 7. Naturally enough, in both encryption and decryption, as the size of an XML document increases, more time is taken for processing of the XML document. Also, in general, the performance of the stream-based implementation is better than that of the DOM-based implementation. The stream-based implementation achieves a 0.27%-26% reduction in processing time (i.e., 1.0x-1.3x performance) for encryption of XML documents with sizes larger than 2 KB, and 34%-88% (i.e., 1.5x-8.5x) for decryption of XML documents with any sizes. The best performance for the combination of encryption and decryption is achieved if the size of an XML document before encryption is in the range approximately from 100 KB to 200 KB. We note that processing time is superlinear in the size of an XML document. We suppose that this result is due to memory overhead of Java.

5.4 Discussion

The times for parsing and for actual encryption or decryption should be analyzed separately. In encryption, the time for parsing is reduced as expected. We believe that this result is due to avoiding the creation of any DOM nodes. However, the time for encryption increases for XML documents with sizes smaller than 2 KB and this increase contributes to the increase of total time. We suppose that this result is due to overhead for encryption, e.g., creating an encryptor body dynamically. Because the time for this creation is constant regardless of the size of an XML document under the conditions of the experiment, it cannot be ignored if the size of an XML document is small. Serialization can also be overhead because, as presented in Section 5.1, the serializer of Xerces2 was not used as it was, but extended for XNI. It may not much influence the result if the size of an XML document is small, though, because this overhead increases as the size of an XML document increases. These problems can be solved by creating an encryptor body beforehand and using another serializer specialized for XNI, respectively.

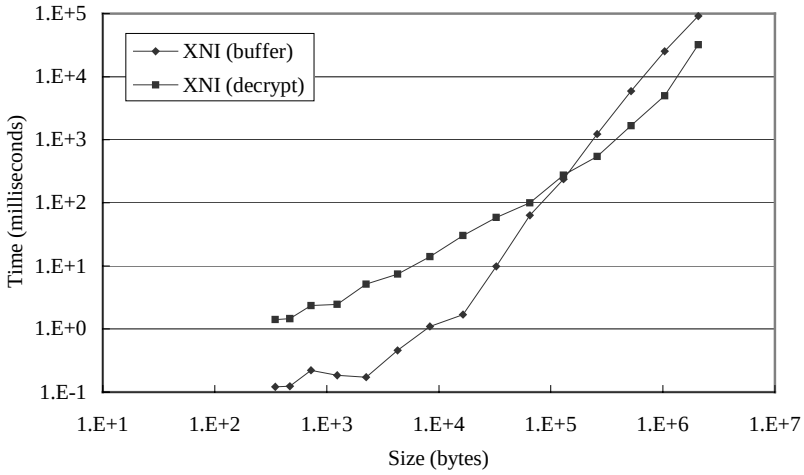

Figure 9. Details of time for decryption

In decryption, the result is very interesting. It is surprising that the time for parsing is drastically reduced. This result is the exact opposite of that in encryption. Before encryption, as the size of an XML document increases, the XML document has a wider and deeper structure. By contrast, after encryption, even if the size of an XML document increases, the XML document does not have a more complicated structure, but just contains longer text in the `CipherValue` element. With Xerces2, as the text gets longer, creating a DOM text node corresponding to the text takes more time because allocating space and copying text to the space occurs more frequently. By contrast, sending a series of XNI events corresponding to the text takes less time, because it does not do allocation or copying. We believe that this causes the result.

However, this advantage in parsing contributes to the disadvantage in decryption, because, as described in Section 4, the decryptor body buffers XNI events (including ones corresponding to text), if necessary, and this buffering can cause the same problem as above. We believe that this is why the time for decryption increases drastically as the size of an XML document increases. Actually, as illustrated in Figure 9, where the horizontal and vertical axes represent the same as in Figure 8, the details of the time for decryption, i.e., the times for buffering and for actual decryption (including parsing) indicate that as the text gets longer, much more time is required for buffering. Consequently, this problem can be solved (1) by reducing the number of times allocation or copying occurs or (2) by avoiding the buffering itself.

6. CONCLUSION

In this paper, we have described the design and performance of a stream-based implementation of the XML Encryption specification that we prototyped. It is implemented as parser components of Xerces2 and works as a part of a parser. As compared with a DOM-based implementation, it achieves a 0.27%-26% reduction in processing time (i.e., 1.0x-1.3x performance) for encryption of XML documents with sizes larger than 2 KB, and 34%-88% (i.e., 1.5x-8.5x) for decryption of XML documents with any sizes.

Future work includes a solution for the overhead problem in encryption. This can be solved by creating an encryptor body beforehand and/or using another serializer specialized for XNI. However, even if these solutions are taken, the performance for encryption may not be much improved. We have not thought of any ideas to improve it drastically yet, and therefore to find such ideas is another area for future work. The other area is to solve the buffering problem in decryption. This can be solved (1) by reducing the number of times allocation or copying occurs or (2) by avoiding the buffering itself. If these problems are effectively solved, it is expected that the performance for both encryption and decryption will be further improved.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous referees for their valuable comments.

REFERENCES

[1] T. Bray, J. Paoli, C. M. Sperberg-McQueen, and E. Maler. Extensible Markup Language (XML) 1.0 (Second Edition), W3C Recommendation, 2000. *http://www.w3.org/TR/2000/REC-xml*

[2] World Wide Web Consortium (W3C). *http://www.w3.org*

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

[4] Internet Engineering Task Force (IETF). *http://www.ietf.org*

[5] D. Eastlake and J. Reagle. XML Encryption Syntax and Processing, W3C Proposed Recommendation, 2002. *http://www.w3.org/TR/xmlenc-core*

[6] XML Encryption Implementation and Interoperability Report, 2002. *http://www.w3.org/Encryption/2002/02-xenc-interop.html*

[7] A. L. Hors, P. L. Hégaret, L. Wood, G. Nicol, J. Robie, M. Champion, and S. Byrne. Document Object Model (DOM) Level 2 Core Specification Version 1.0, W3C Recommendation, 2000.
http://www.w3.org/TR/DOM-Level-2-Core

[8] Triple Data Encryption Algorithm Modes of Operation, ANSI X9.52, 1998.

[9] B. Schneier. Applied Cryptography, Second Edition, John Wiley & Sons, Inc., 1996, Section 17.1, pp. 397-398.

[10] Simple API for XML (SAX) 2.0, 2000.
http://sax.sourceforge.net

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

[12] Apache XML Project. Xerces2 Java Parser 2.0.1.
http://xml.apache.org/xerces2-j

[13] Apache XML Project.
http://xml.apache.org

[14] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax, RFC 2396, 1998.
http://www.ietf.org/rfc/rfc2396.txt

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

[16] H. S. Thompson, D. Beech, M. Maloney, and N. Mendelsohn. XML Schema Part 1: Structures, W3C Recommendation, 2001.
http://www.w3.org/TR/xmlschema-1
P. V. Biron and A. Malhotra. XML Schema Part 2: Datatypes, W3C Recommendation, 2001.
http://www.w3.org/TR/xmlschema-2

[17] Sun Microsystems. Java 2 SDK, Standard Edition, Version 1.3.1.
http://java.sun.com/j2se/1.3

[18] Sun Microsystems. Java Cryptography Extension (JCE) 1.2.1.
http://java.sun.com/products/jce/index-121.html

[19] IBM. XML Security Suite.
http://www.alphaworks.ibm.com/tech/xmlsecuritysuite

[20] J. Clark and S. DeRose. XML Path Language (XPath) Version 1.0, W3C Recommendation, 1999.
http://www.w3.org/TR/xpath

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

[22] IBM. Developer Kit for Windows, Release 1.3.0.
http://www7b.boulder.ibm.com/wsdd/wspvtdevkit-info.html