Developing a SAX Filtering Intermediary Service for Protecting SVG Multimedia Contents in a Ubiquitous Publish/Subscribe Environment

Dr. Jinan Fiaidhi, Dr. Sabah Mohammed, Dr. Madan Garg and Ahmed Sabbir Arif

Abstract—***It is important to enable peers to protect and update their trust in multimedia documents that they are exchanging with other peers in open network for sharing files, especially services. In this paper, we propose a SAX filter intermediary service that can protect SVG sharing on JXTA network.***

Index Terms—**SVG Filter, JXTA Services, Java Crypto APIs, XML Encryption.**

I. Introduction

WITH the increased use of multimedia in daily communications, it is necessary to develop efficient and secure transmission mechanisms that are specifically tailored for multimedia. Ubiquitous computing characteristics and high bandwidth requirements of multimedia data requires efficient and scalable mechanisms. For success of commercial multimedia distribution, security mechanisms will be a major factor. Most of the researches on multimedia security have focused on watermarking related issues [1]. Security issues related to streaming is not researched in detail and requires further progress [2]. Pervasive Internet services today promise to provide users with a quick and convenient access to a variety of commercial applications. Moreover, the multimedia distribution based on the pervasive Internet services today didn't reach a satisfactory success level due to unsuitable architectures and poor performance user acceptance. To be a major success ubiquitous and mobile services have to provide device-adapted multimedia content and advanced value-added Web services. Innovative enabling technologies like XML and peer-to-peer and wireless communication may for the first time provide a facility to interact with online applications anytime anywhere.

There are two major standards currently available for representing multimedia in an XML form with the underlying APIs for their transcoding on mobile and ubiquitous devices, the ebXML standard (http://www.ebxml.org) and the World Wide Web Consortium (W3C) standard (http://www.w3c.org). The most common standard is the W3C which is based on SVG (Scalable Vector Graphics) standard for representing multimedia. There are three variants of SVG (SVG 1.2, SVG Basic and SVG Tiny) which can be used with resource-limited mobile devices. Unlike other multimedia formats, SVG becomes a powerful tool for anybody managing multimedia content for the Web or other environments [3]. By leveraging the force of XML and the visual strengths of dynamic and easily accessible vector graphics, the Apache XML Project's Batik team extends this power in building a successful APIs that can be used for transcoding. With SVG all the graphical information is stored in a sequence of commands to draw lines, shapes, and other objects. This information is eventually converted to application-specific bitmaps, also called raster graphics. The task of converting an SVG image to a raster image is accomplished either with a browser (plug-in) or using rendering APIs within an application. Note that this is different from bitmap-based multimedia graphics like GIFs or JPEGs, which must be generated on the server and delivered to the client as bitmap images. An SVG file is in XML format. When it is delivered to a browser/SVG renderer, all the contents are stored in the Document Object Model (DOM) and the DOM tree can be transcoded and optimized according to the capability of the receiving device. However, the heterogeneous issue in terms of device capabilities is a problem that have been highly investigated and in order to cater to the needs of these devices in many Web applications, smart intermediaries have been developed to increase the user satisfaction by hiding the inherent weakness of some of the small although handy devices like the PDAs and Web-tops [4]. Such intermediaries

Accepted Article (Paper ID Number: ICC3800) for Presentation at the International Conference on Internet Computing (**ICOMP2005**) at the 2005 International Multiconference in Computer Science and Computer Engineering, June 27-30, 2005, Las Vegas, USA.

Dr. Jinan Fiaidhi is an Associate Professor with the Department of Computer Science, Lakehead University, Ontario P7B 5E1 Canada (phone: 1-807-343-8224; fax: 1-807-343-8023; e-mail: jfiaidhi@lakeheadu.ca).

Dr. Sabah Mohammed is an Associate Professor with the Department of Computer Science, Lakehead University, Ontario P7B 5E1 Canada (e-mail: mohammed@lakeheadu.ca).

Dr. Madan Garg is a Principle Lecturer with the College of Information Technology, University Tenaga National, 43009 Kajang, Selangor, Malaysia (e-mail: madan@uniten.edu.my).

Ahmed Sabbir Arif is a Graduate Student of the Department of Computer Science, Lakehead University, Ontario P7B 5E1 Canada (e-mail: asarif@lakeheadu.ca).

provide a framework for creating intermediary applications which can monitor and modify the flow of Web data between clients, servers and proxies. For example, they can produce personalized content, connect local and remote information on the Web, route Web traffic, translate protocols, translates document formats or transcoding documents. All such intermediary operations aim to produce a more powerful and flexible web. Indeed, all such advantages are made possible by only maintaining limited number of central servers. Such intermediaries are only available for the Web client server architecture and cannot be adopted for peer-to-peer ubiquitous environment.

In this paper we are proposing a framework for developing a smart intermediary for SVG protection that can be used for ubiquitous publish/subscribe environment like JXTA. Through such an intermediary architecture that serves a heterogeneous client base we exploit the issue of protecting SVG contents through the notions of SVG Filtering and SVG Encryption.

II. THE PUBLISH SUBSCRIBE UBIQUITOUS ENVIRONMENT

Web Intermediaries (WBI, pronounced “webby”) was the first architecture and framework for enabling ubiquitous computing integrity by creating intermediary applications on the web that can act as proxies between the varieties of heterogeneous devices. Indeed WBI is a programmable web proxy and web server. The work on Web intermediaries is started by IBM in there WBI model [5]. However, some major concerns have been raised about the WBI model, for example overload and expensive operation cost of the central servers, needs of mutual and direct communications between network users. As a result, peer-to-peer technology has become popular and has been used in networks that manage vast amounts of data daily, and balance the load over a large number of servers. Peer-to-peer (P2P) applications such as distributed search applications, file sharing system, collaborative multimedia, distributed storage system, and distributed learning objects have been proposed and developed. Such P2P applications do not use central servers and are based on publish/subscribe model (PSM), which gained much attention and popularity these days for disseminating information in complex distributed applications. Unlike the traditional WBI model, the publish/subscribe middleware takes care of all the network programming and message-passing chores, dramatically simplifying application development for enterprise and Internet applications. Publish/subscribe data distribution is gaining popularity in many distributed applications, such as searching and exchanging learning objects in a highly distributed environment (e.g. Edutella [6]), financial communications [7], web-based push technologies [8], and command and control systems [9]. Its popularity is justified. Publish/subscribe substantially reduces development, deployment, and maintenance effort while delivering better performance for applications with complex data flow. Several features characterize the publish/subscribe architecture [10]:

1) Distinct Declaration and Delivery: Communications occur in three steps:

1. Publishers declare intent to publish a publication.
2. Subscribers declare interest in a publication.
3. Publishers send a publication issue.

2) Named Publications: Publish/subscribe applications distribute data using named publications. Each publication has a topic and a type. The topic is a name used by publishers and subscribers to create a logical data channel. The type describes the data format. Each incremental value of a publication is an issue. Most publish/subscribe architectures support arbitrary, user-defined types with automatic type conversion among computer architectures.

3) Many-to-many Communications Support: Publish/subscribe distributes each issue simultaneously from one publisher to many subscribers. The model’s flexibility also helps developers implement complex, many-to-many distribution schemes quite easily. For example, different publishers can declare the same topic so those subscribers get issues from multiple sources.

4) Event-driven Transfer: Publish/subscribe communication is naturally event driven. Publishers send each issue when it is ready. When the issue arrives, the subscribers receive notification.

5) Middleware: Publish/subscribe services are typically made available through middleware that sits on top of the operating system’s network interface and presents an application programming interface. The middleware handles three basic programming chores:

- Maintains the database that maps publishers to subscribers. The result is logical data channels for each publication between publishers and subscribers.
- Serializes and deserializes the data on its way to and from the network to reconcile publisher and subscriber platform differences.
- Delivers the published data.

III. SERVICES IN JXTA PUBLISH/SUBSCRIBE NETWORK

Capturing the advantages of a PSM requires more than just migrating to the use of XML Web Services for peer-to-peer transactions. To enable web services over ubiquitous and P2P environments it is necessary to establish open, asynchronous, scalable data exchange network among multiple entities with security and reliable delivery guarantees via XML routing. In this direction there are only few infrastructures that can be used for this purpose, like, *Jabber* [11], *JXTA* [12], and *NaradaBrokering* [13]. On one hand, Jabber represents an instant messaging service with file sharing. Jabber was originally designed to provide interoperability among popular Internet instant messaging systems (AOL, MSN, Yahoo!, ICQ, and so on). It is a powerful and flexible yet simple protocol that can wrap around all existing instant messaging protocols. However, Jabber is not a sophisticated protocol that is able to support smart middleware such security filters. On the other hand, JXTA and NaradaBrokering are more sophisticated infrastructures that can model security middleware based on JXTA Super Peer Proxies and Narada

Brokers. Narada is a distributed event brokering system based on the publish/subscribe paradigm and is designed to run on a very large network of broker nodes. Actually, Narada is designed to support for P2P interactions only through JXTA via using a JMS compliant protocol [14]. This means that JXTA remains the essential infrastructure for constructing publish/subscribe networks with support of middleware services. Peers cooperate and communicate to publish, discover and invoke network services. With JXTA peers can publish as many services that it can provide. Peers discover network services via the Peer Discovery Protocol. JXTA describe services via *Modules* or *Module Advertisements.* JXTA modules are an abstraction used to represent any functionality that has to be implemented in the network. Modules are used to represent different implementations of a network service on different platforms. The module framework needs to distinguish there different module implementations, possibly by some metadata representation, such as an advertisement. The Module Class Advertisement is primarily used to advertise the existence of a class (behavior) of a module. The module class provides the information shown below:

```
<?xml version="1.0"?>
<!DOCTYPE jxta:MCA>
<jxta:MCA xmlns:jxta="http://jxta.org">
<MCID>
   urn:jxta:uuid-00CFA9212D884C61B8189F9C70AA35E105
</MCID>
<Name>
   urn:module/hello-service
</Name>
<Desc>
   A service to allow us to say hello
</Desc>
</jxta:MCA>
```

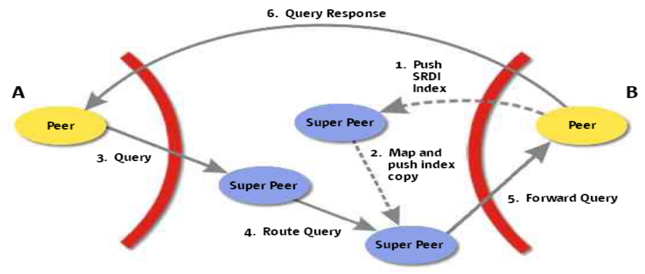

Fig. 1. The JXTA Service Publish Subscribe Mechanism.

The advertisement can be used by the peer to search for a module class based on its elements, Module Class ID (MCID), its Name (Name) and the description (DESC) [15]. Each module has a unique Module Class Advertisement to identify its Module Class (the service). In the above example, the 'Hello Service' is associated with its own Module Class Advertisement that defines a module class responsible for saying *"Hello World!"* However, creating a service requires also that these module advertisements which are published in the JXTA network to be discovered by other peers. Actually what JXTA provided for discovering services is an asynchronous mechanism for searching for advertisements (e.g. peers, peer groups, pipes, class advertisement module, services) which can be retrieved in JXTA local cache. The discovery mechanism can also sends Discovery Query Message to a specific peer or to be propagated to the JXTA network. Fig. 1 illustrates the JXTA service publish/subscribe mechanism.

IV. JXTA SVG Intermediary Security Filter Service

Commonly the Secure Socket Layer (SSL) is the protocol used for the Web application for achieving security, which is typically used with HTTP. Despite its popularity, SSL has some limitations when it comes to Intermediaries. SSL is designed to provide point-to-point security, which fall short for multiple intermediaries such as security and transcoding intermediaries because of the many nodes that could exist between the two endpoints [16]. Thus, various XML-based security initiatives are in the works to address the security problem of intermediaries. These schemes include:

- XML Encryption (W3C Standard),
- XML digital signature (W3C Standard),
- XKMS (XML Key Management Specification),
- XACML (Extensible Access Control Markup Language),
- SAML (Secure Assertion Markup Language),
- WS-Security (Web Services Security),
- ebXML Message Service, and
- The Liberty Alliance Project.

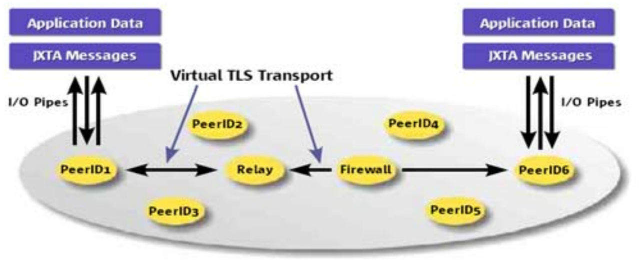

Fig. 2.1. JXTA TLS Secure Transportation Protocol.

On the light of these initiatives JXTA team proposed three choices for establishing various security levels [17]: the adoption of TLS (http://www.rtfm.com/puretls), an emerging industry protocol for the secure transport of information (Fig 2.1); Cryptix 3 (http://www.cryptix.org) is an implementation of Sun's JCE 1.1 Java Heavyweight Cryptographic Extensions and provides standard interfaces for cryptographic algorithms and services. It is used by the TLS for various algorithms, and the Cryptix ASN.1 Kit (http://sourceforge.net/projects/cryptix-asn1) which is a language that allows definitions of various data types, such as integers, strings, sequences, and so on. The ASN.1 kit can be used to use this notation in Java programs to support X509V3 authentication certificates.

Although such security directions might be appealing, they create several technical difficulties when implemented for securing newly created services in ubiquitous environment [18]. For the same reason the Bridge project [19] tries to use the same Web Security techniques with JXTA by making the JXTA messaging talk SOAP. The use of SOAP will enable JXTA to:

1. Let JXTA peers discover services without having to worry about building in protocol style interaction for each service.

2. SOAP can be used for encodings, faults, etc. over the JXTA network.
3. JXTA services from different parties can interact via SOAP without having to worry about protocol level details.
4. Services that are built for JXTA can also be deployed over other networks. HTTP, SMTP, etc.

Fig. 2.2 illustrates the use of JXTA SOAP for JXTA services publishing and discovery.

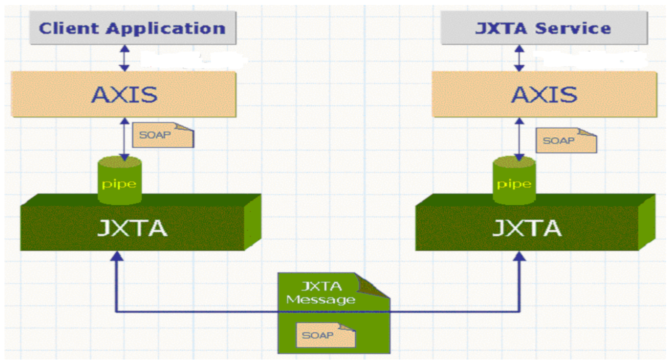

Fig. 2.2. XTA SOAP Service Publishing and Discovery Infrastructure.

The Axis represents the SOAP engine -- a framework for constructing SOAP processors such as clients, servers, and gateways. The Bridge project intention of adopting SOAP as the encoding format for inter-peer communication for JXTA and for publishing and subscribing services is to adopt or extend the good WBI Web Services Standards into JXTA (e.g. WS-Security, WS-Management). This type of integration scheme is not fully developed as well as it clearly adds more complexity on top of JXTA which need to avoid when dealing with ubiquitous environments.

The method that we are proposing in this paper is to use SAX as a steam processing API to perform security operations such as SVG Signature and SVG Encryption. 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 whatever the encryption algorithms are involved (e.g. block encryption or stream encryption). The SAX model is quite different from the DOM model. Rather than building a complete representation of the document, the SAX fires off a series of events as it reads the document from beginning to end. Those events are passed to event handlers, which provide access to the contents of the document. There are three classes of event handlers: `DTDHandlers`, for accessing the contents of SVG Metadata; `ErrorHandlers`, for low-level access to parsing errors; and, by far the most often used, `DocumentHandlers`, for accessing the contents of the document. For clarity's sake, we will only cover `DocumentHandler` events. A SAX processor will pass the following events to a `DocumentHandler`:

- The start of the document.
- A processing instruction element.
- A comment element.
- The beginning of an element, including that element's attributes.
- The text contained within an element.
- The end of an element.
- The end of the document.

The `DocumentHandler` transforms the various objects into stream which is known as XML serialization. This process transforms a Java object tree into a textual XML stream. Many APIs helps in performing SAX processing (e.g. Xerces, Castor) and sometimes be used for pre-processing the input to an XSLT transformation, or for post-processing the output (see Fig. 2.3).

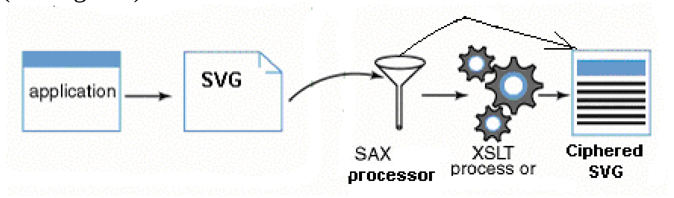

Fig. 2.3. SAX Processor without or with XSLT engine.

A SAX filter is simply a class that is passed as the event handler to another class that generates SAX events, then forwards all or some of those events on the next handler (or filter) in the processing chain [20]. The basic idea is that an `XMLReader`, instead of receiving XML text directly from a file, socket, or other source, receives already parsed events from another `XMLReader`. It can change these events before passing them along to the client application through the usual methods of `ContentHandler` and the other callback interfaces [21]. For example, it can add a unique `ID` attribute to every element or delete all elements in the SVG namespace from the input stream. Fig. 3.1 diagrams the normal course of XML processing where Fig. 3.2 diagrams the course of XML processing with a filter.

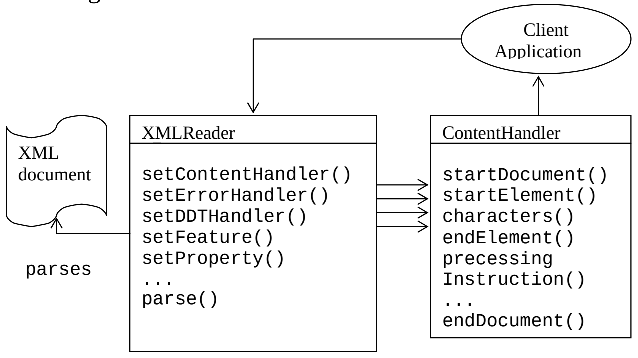

Fig. 3.1. The normal course of XML processing. A client application instructs a parser, represented in SAX by an `XMLReader` object, to read the text of an XML document. As it reads, the parser calls back to the client application's `ContentHandler`.

Since the filter sits in the middle between the real parser and the client application, it can change the stream of events that gets passed back and forth between the two. For example, it can add new shapes to SVG on the fly. It can add namespaces to elements and attributes that don't normally have them. It can work with the stream without actually changing the data itself. And as we will see in later sections, it can call a cryptography library to encrypt elements, decrypt encrypted elements, sign a document or verify the signature.

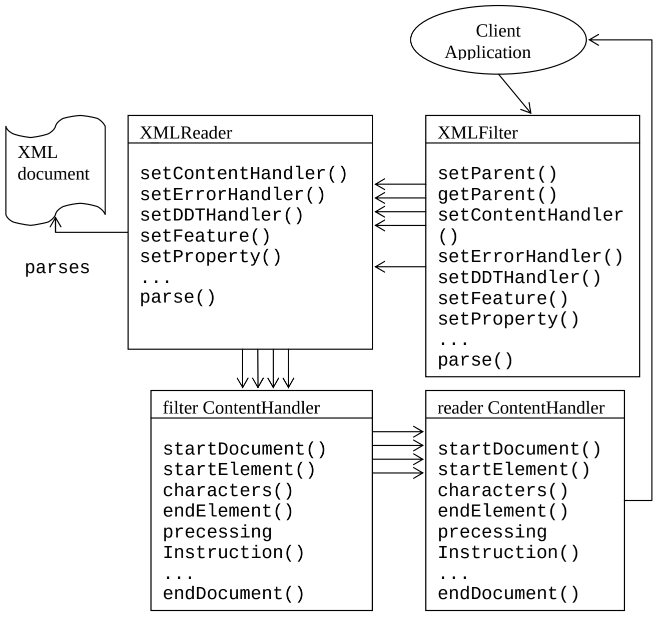

Fig. 3.2. The course of XML processing with a filter. A client application instructs the filter, represented in SAX by an XMLFilter object, to read the text of an XML document. The filter then instructs the parser to read the text of an XML document. As it reads, the parser calls back to the filter's ContentHandler. The filter's ContentHandler then calls back to the client application's ContentHandler.

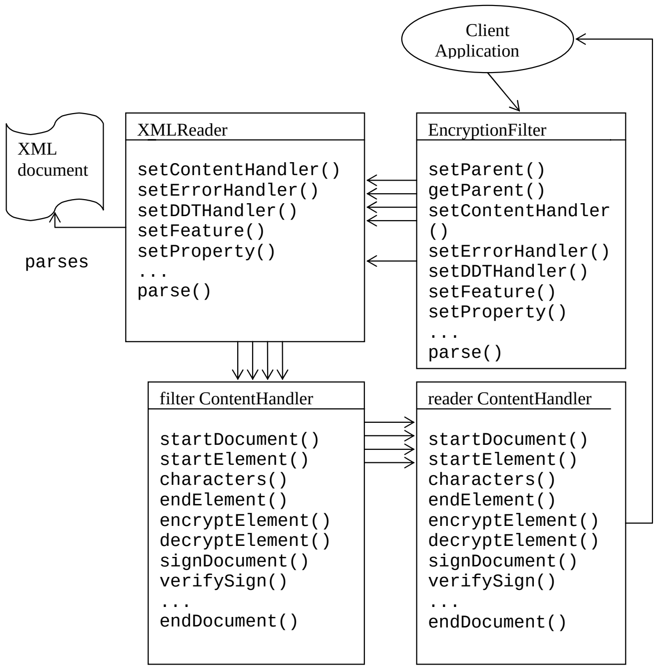

Fig. 4. The course of XML crypto processing with a filter.

V. Using the SAX Filter for Cryptography Tasks

The proposed model for protecting SVG multimedia contents is very simple. A client application will instructs the SAX filter to read the text of an XML (which is an SVG indeed) document. The filter then instructs the parser to read the text of an XML document. As it reads, the parser calls back to the filter's `ContentHandler`. The filter's `ContentHandler` will contain the `processingInstruction()` function, which could be an instruction to encrypt/decrypt elements, or sign/verify the document. The filter's `ContentHandler` will do the cryptographic tasks (by calling functions like `encryptElement()`, `decryptElement()`, `signDocument()` or `verifyDocument()`) and will call back to the client application's `ContentHandler` (or may be another filter's `ContentHandler`, if necessary. In this paper we will only look at single filters. This is because using more than one filter makes the diagrams complex and hard to fit in the page. But adopting the idea with multiple filters is possible by following the same architecture). Fig. 4 illustrates the architecture. Now the question arises, which cryptography libraries to use with this model. We will focus on this topic in the next section.

VI. Choosing the XML Security API Library

XML Encryption and Signature is very new. IBM has submitted a Java Specification Request (JSR 106) that will define its standard API [22]. Currently, however, no standard exists on how the API should look. Only a handful of libraries provide XML Encryption and Signature, and most of them are heavyweight. Among the claimed lightweight libraries (like the Bouncy Castle lightweight cryptography API or IAIK JCE Toolkit) which are not yet tested for ubiquitous environments. However and in order to suite the constraints of ubiquitous devices, we decided to use the libraries provided by the Java Card Technology [23]. In this direction, our filter's `ContentHandler` will perform the cryptographic tasks by calling functions like `encryptElement()`, `decryptElement()`, `signDocument()` or `verifyDocument()`. Then it will call back to the client application's `ContentHandler` or to other filter's `ContentHandler`. These functions can call java card's security and cryptography classes to do its task (see Fig. 5).

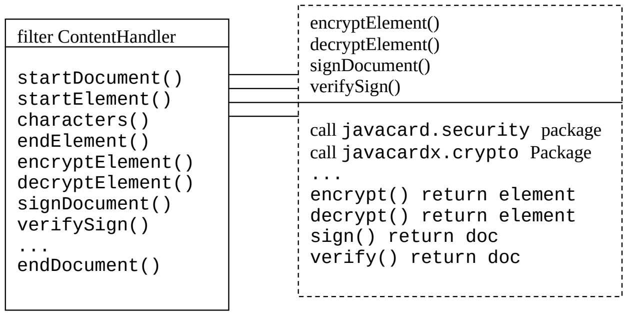

Fig. 5. The dependency between filter ContentHandler and the Java Card Security and Crypto packages.

Java Card™ offers the following algorithms for cryptography purposes [25]:

1) Secret-key Algorithms: The `javacardx.crypto.Cipher` class is the abstract base class for Cipher algorithms. It provides encryption/decryption using secret-key algorithms (DES, Triple DES). The `javacard.security.Signature` class, which is the base class for Signature algorithms, provides signature/verification using secret-key algorithms (DES, Triple DES).

2) Public-key Algorithms: The `javacardx.crypto.Cipher` class provides encryption/decryption using public-key algorithms (RSA) as well. `javacard.security.Signature` class also provides signature/verification using public-key algorithms (RSA, DSA).

3) One-way Hash Functions: The base class for hashing algorithms is class `javacard.security.MessageDigest`. It provides mess-age digests, using one-way hash functions (MD5, SHA-1, RIPEMD160).

The latest Java Card Specification (Version 2.2.1) also supports AES (Advanced Encryption Standard) and Elliptic Curves [24].

VII. Implementation Issues

A. Canonicalization

The Canonicalization of XML (which is in form of SVG) documents must be done before applying our proposed model. The XML 1.0 Recommendation describes the syntax of a class of data objects called XML documents [26]. It is possible, however, for logically equivalent XML documents to differ in their physical representation. In particular, two equivalent XML documents may differ on such issues as physical (i.e. entity) structure, attribute ordering, character encoding and insignificant whitespace. This means, equivalence testing cannot be done at the byte level for arbitrary XML/SVG documents. Such equivalence testing is useful in a number of domains including digital signatures and encryption. The Canonical XML specification aims to introduce a notion of equivalence between XML documents which can be tested at the syntactic level and, in particular, by byte-for-byte comparison. It describes the canonicalization of XML documents such that logically equivalent documents will have the same byte-for-byte representation. We will not discuss on canonicalization and will proceed further assuming all the XML/SVG documents we are working on are in Canonical Standard.

B. Implementing the SAX Portion

At first we have to create a filter interface that will call methods like `getParent()` and `setParent()`. It will inherit from a superinterface. The *parent* of a filter is eventually the `XMLReader` to which the filter delegates most of its work. In the context of SAX filters, the parent is not normally understood to be the superclass of the filter class.

In the superinterface we should call methods like `getParent()`, `setParent()` and others (like, `setFeature()`, `getFeature()`, `setProperty()`, `getProperty()` so forth) if necessary.

In the superinterface we will call the cryptography methods we mentioned before. But before doing that we have to collect the element or attribute we are planning to encrypt or decrypt. Let's say we are interested to encrypt `d` which is an attribute of SVG `Path` element.

Filtering an element is very straightforwardly. The complete set of attributes for any element is completely available a single method call as an `Attributes` object. Thus it's easy to read through the list and respond appropriately without building complicated data structures to maintain state between method invocations. But as `Attributes` interface is read-only we also need to create our own object that implements the `Attributes` interface and is mutable. Doing so would be a simple matter of programming, most of the time it's easier to use the class SAX provides for this purpose, `org.xml.sax.helpers.AttributesImpl`. This implements the `Attributes` interface and adds methods for copying existing `Attributes` objects and adding attributes to and deleting attributes from the list -- which is quite capable to satisfy our purpose.

Which means we can get the value of `d` from the `Path` element by calling a method like `getD(type c)` in the `AttributesImpl` helper class. After getting the value we are going to pass it to another class called `encrypt`. This is because as we are using a different API (Java Card) to do the encryption task. Doing the cryptography tasks in different classes will not only avoid same reserved word conflict but will also increase robustness and reusability.

C. Implementing the Java Card Portion

Let's assume we already have generated the *Public* and *Private Key Pair* and stored them to separate files and concentrate on *Sealing the Symmetric Key*.

The pseudo code below shows an `encrypt` method to encrypt `d`, seal the symmetric key, and send the encrypted value and sealed key to the server:

```
private void encrypt(d)
{
   Create cipher for symmetric key encryption (DES)
   Create a key generator
   Create a secret (session) key with key generator
   Initialize cipher for encryption with session key
   Encrypt d with cipher
   Get public key from server
   Create cipher for asymmetric encryption
                                (do not use RSA)
   Initialize cipher for encryption with public key
   Seal session key using asymmetric Cipher
   Send encrypted d and sealed session key to server
}
```

The pseudocode above says "`do not use RSA`" because it has the size restrictions, and the sealing process makes the session key too large to use with the RSA algorithm.

We can also encrypting the symmetric key with the RSA algorithm. The RSA algorithm imposes size restrictions on the object being encrypted. RSA encryption uses the PKCS#1 standard with PKCS#1 block type 2 padding [27]. The PKCS RSA encryption padding scheme needs 11 spare bytes to work. So, if we generate an RSA key pair with a key size of 512 bits, we cannot use the keys to encrypt more than 53 bytes (53 = 64 - 11) [27].

So, if we have a session key that is only 8 bytes long, sealing expands it to 3644 bytes, which is way over the size restriction imposed by the RSA algorithm. In the process of sealing, the object to be sealed (the session key, in this case) is first serialized, and then the serialized contents are encrypted. Serialization adds more information to the session key such as the class of the session key, the class signature, and any objects referenced by the session key. The additional information makes the session key too large to be encrypted with an RSA key, and the result is a `javax.crypto.IllegalBlockSizeException` run time error.

The pseudo code below shows how to encrypt `d`, seal (encrypt) the session key, and send the encrypted `d` and sealed session key to the server:

```
private void encrypt(d)
{
   Create cipher for symmetric key encryption (DES)
   Create a key generator
   Create a secret (session) key with key generator
   Initialize cipher for encryption with session key
   Encrypt d with cipher
   Get public key from server
   Create cipher for asymmetric encryption (RSA)
   Initialize cipher for encryption with public key
   Encrypt session key
   Send encrypted d and session key to server
}
```

VIII. Conclusion

SVG is spreading to back multimedia office systems, business exchanges and wireless applications. Recently, SVG is used by almost 50% of Websites, according to some researchers. SVG has many advantages over other multimedia formats, and particularly over JPEG and GIF, the most common multimedia graphic formats used on the Web today. Specifically because of are open-standard, XML syntax, and scalability -- although the security awareness today is higher than it's ever been. We have industrial-strength encryption that's almost certainly unbreakable for at least the next decade. But the very strength of our encryption capabilities is only governed by those centralized servers which support such powerful security algorithms. This paper presents a framework for securing SVG contents on ubiquitous peer-to-peer environments that is fully decentralized. The paper investigates the available peer-to-peer infrastructures and identifies JXTA as an effective one. Also it introduces a SAX filter for protecting SVG contents using lightweight cryptographic API such as Java Card. There are many implementation issues remains to be addressed at our next research work.

References

[1] A. M. Eskicioglu and E. J. Delp, "An Overview of Multimedia Content Protection in Consumer Electronics Devices," Signal Processing: Image Communication, Vol. 16, 2000, pp. 681-699.

[2] J. Roberto Bayardo, "An Evaluation of Binary XML Encoding Optimizations for Fast Stream Based XML Processing", ACM *WWW2004 Conference*, May 17–22, 2004, New York, New York, USA.

[3] Tien Tran Thuong and Cecile Roisin, "Structured Media for Authoring Multimedia Documents", *First International Workshop on* Web Document Analysis (WDA2001) Seattle, Washington, USA September 8, 2001.

[4] Mingchao Ma and Christoph Meinel "Independent Trust Intermediary Service (ITIS)", In Proceedings of IADIS International Conference WWW/Internet 2002, Nov.2002, Lisbon, Portugal, pp.785-790.

[5] IBM Research. Web Intermediaries (WBI) Development Kit [Online]. Available: http://www.almaden.ibm.com/cs/wbi/doc

[6] Wolfgang Nejdl et al, Edutella: "A P2P Networking Infrastructure Based on RDF", ACM *WWW2002*, May 7-11, 2002, Honolulu, Hawaii, USA.

[7] Ju Long et al, "Securing newEra of Financial services", IEEE IT Pro Journal, July/August Issue, 2003.

[8] Brena Pinero, Ramón Felipe; Aguirre Cervantes, José Luis. "Push Technologies leveraged by Intelligent Agents", the 8th World Multi-Conference on Systemics, Cybernetics and Informatics. Orlando, Florida, July 2004.

[9] Graeme Burnett, "From CORBA to Command and Control Web Services: - Web Services in Evolution", Enhyper, May 2003.

[10] Ramnivas Laddad, Gerardo Pardo-Castellote, Stan Schneider, "Publish-Subscribe Model: A question of architectural advantages and limitations", ISA InTech Journal, Issue of 31 May 2001.

[11] Peter Saint-André, "XML Messaging With Jabber", O'Reilly XML Journal, October 30, 2000 [Online]. Available: http://www.oreillynet.com/pub/a/p2p/2000/10/06/jabber_xml.html

[12] Bilal Siddiqui, "JXTA for Wireless Java Programmers", Java Developer Journal, September 16, 2002.

[13] Shrideep Pallickara and Geoffrey Fox, "NaradaBrokering: A Distributed Middleware Framework and Architecture for Enabling Durable Peer-to-Peer Grids", in Proceedings of ACM/IFIP/USENIX International Middleware Conference Middleware-2003, Rio Janeiro, Brazil June 2003.

[14] Faheem Khan, "Implement JXTA-for-JMS", IBM Resarch Journal, 22 Feb 2005.

[15] Sing Lee, "Making P2P interoperable: The JXTA command shell", IBM Resarch Journal, 01 Sep 2001.

[16] Zhenhai Duan et al., "Push vs. Pull: Implications of Protocol Design on Controlling Unwanted Traffic", Technical Report, University of Minnesota in June 2003 [Online]. Available: http://www.cs.fsu.edu/~duan/publications/pushpull.pdf

[17] Daniel Brookshier, Navaneeth Krishnan and Darren Govoni," JXTA: Java P2P Programming", Sun Micro Systems, (June 2002).

[18] S. Mohammed, J. Fiaidhi, and L. Yang, "Developing Multitier Lightweight Techniques for Protecting Medical Images within Ubiquitous Environments", IEEE 2nd Communications, Networks and Service Research Conference (CNSR04), Fredericton, N.B., Canada, May 19-21, 2004.

[19] Kevin A. Burton, "The JXTA SOAP Bridge", http://soap.jxta.org/servlets/ProjectHome

[20] Kip Hampton, "Transforming XML With SAX Filters," *O'Reilly XML Online Articles*, Oct. 2001 [Online]. Available: http://www.xml.com/pub/a/2001/10/10/sax-filters.html

[21] Elliotte Rusty Harold, *Processing XML with Java: A Guide to SAX, DOM, JDOM, JAXP, and TrAX*, 1st ed. Boston: Addison-Wesley Professional, 2002, ch. 8.

[22] Ray Djajadinata, "Yes, you can secure your Web services documents", *Java World Online Articles*, Aug. 2002 [Online]. Available: http://www.javaworld.com/javaworld/jw-08-2002/jw-0823-securexml.html

[23] Sun Microsystems, Inc.. Java Card™ Technology Overview [Online]. Available: http://java.sun.com/products/javacard/overview.html

[24] Sun Microsystems, Inc. Release Notes (October 2003). Java Card™ Specification (V2.2.1) [Online]. Available: http://java.sun.com/products/javacard/RELEASENOTES_jcspecs.html

[25] Java Card™ 2.1 Platform API Specification (V2.1) [Online]. Available: http://java.sun.com/products/javacard/htmldoc

[26] World Wide Web Consortium (March 2001). Canonical XML (V1.0) [Online]. Available: http://www.w3.org/TR/xml-c14n

[27] Monica Pawlan, "Essentials of the Java Programming Language", *Sun Developers Network Tutorials & Code Camps*, July 1999. Available: http://java.sun.com/developer/onlineTraining/Programming/BasicJava2