A Streaming Validation Model for SOAP Digital Signature

Wei Lu[1], Kenneth Chiu[2], Aleksander Slominski[1], Dennis Gannon[1]
1. Computer Science Department, Indiana University
2. Department of Computer Science, State University of New York (SUNY) at Binghamton

Abstract

The XML Signature specification provides a rich and flexible message signature model for XML documents, and it has been adopted by SOAP applications to provide message-level security. However, the XML Signature design introduces a number of complex processing steps, such as canonicalization and XPath filtering, that often lead to performance and scalability problems when encountering extremes of size and rate in the processing of XML. In this paper, we focus on the performance of validating large signed XML messages, as might be sent by a scientific application using Grid Web Services. We present the design and implementation of the GHPX/SSSV system for the streaming validation of SOAP Digital Signature. Our model consists of a streaming canonicalization and optimized SOAP signature validation. We present an empirical study of the performance characteristics of these streaming validation features. Based on our evaluations we conclude that the streaming validation model can not only provide high performance, but is also memory efficient.

Keyword: *XML Signature, WS-Security, Performance, Streaming*

1 Introduction

Security is a crucial feature of Web Services and Grid Services [1]. A common technology to support security in distributed system is SSL, which provides a transport-level, secure tunnel for the communication of data. However, as Web Services and Grid Services adopt a message-based architecture, they require security that can be guaranteed during the entire processing of a SOAP message, including not only transport between every hop but also application level processing at the final destination. Though SSL does not match the full security requirement of Web and Grid Services, it can be used as one of the lower-level security mechanisms.

This work was partially supported by NSF Award SCI-0330568, NSF Award DBI-0446298, and NSF Award EIA-0202048.

To support message-level, end-to-end security [5], SOAP Digital Signature [14] and its successor WS-Security [2] define how to use the XML Security syntax [15] to sign or encrypt a SOAP message. Theoretically, they adhere very well to the architecture of Web and Grid services, without introducing any of the limitations of transport level security. Unfortunately, the present implementations are infamous for their extreme inefficiency. Shirasuna found in [7] that message-level security can degrade the performance of a SOAP based Grid service by orders of magnitude compared to transport level security. This makes message-level security almost impractical for scientific computing applications which send large, signed XML messages via Grid Web services. As a result Globus Toolkit 4, the newest Grid computing infrastructure toolkit, had to make the dilemmatic decision of setting transport level security by default [17].

The performance problem is rooted in the complexity of the XML Security specification, in which the flexibility and expressiveness of the security model rather than the performance are the major concern. The second factor contributing to the performance problem is the libraries, which adopt a layered architecture and conform to the specification layer by layer perfectly but at the cost of increased operations between layers.

Automaton-based methods for processing streaming data are attractive due to their efficiency and clean design. In this paper we present a novel streaming processing model for the validation of SOAP Digital Signatures. The model consists of streaming canonicalization and optimized SOAP signature validation. In our streaming canonicalization, SOAP messages can be canonicalized in a streaming fashion during XML parsing without the need for a full-fledged DOM. Furthermore, most signed SOAP messages have a similar and streamable structure and our method applies to a large fraction of them. Consequently, with the help of the streaming canonicalization, a specific and optimized validation model can be designed for these cases. This model basically eliminates the need for an explicit XPath nodeset to represent the data objects. This paper describes how much performance we can gain when we optimize processing for the most common cases. Our empirical study shows

that the streaming validation model can not only gain significant performance improvement, but is also memory efficient, hence being a crucial step toward the practical message level security for the Grid applications.

The rest of this paper is organized as follows. The preliminaries and motivation are introduced in Section 2, and the design and implementation of the system are given in Section 3 and 4. Section 5 presents the result from our empirical study of our system and related systems. We conclude in Section 6.

2 Motivation

2.1 XML Signature

XML signatures [15] are digital signatures designed for associating the result of a digital signature with arbitrary (but often XML) data. Their structure is defined by the grammar shown below

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

The association between the signature and the data is given by the <SignedInfo> element, under which <Reference> elements refer to the signed data whose digest is placed in corresponding <DigestValue>. The <SignatureValue> element contains the actual signature over the <SignedInfo> elements, and <KeyInfo> contains the related key information.

Validation of an XML signature entails both (1) reference validation, the verification of the digest contained in each <Reference> element in a <SignedInfo> element; and (2) the cryptographic signature validation of the signature in the <SignatureValue>. Reference validation requires the following steps:

1. Dereferene the data object specified by the URI attribute of <Reference>. Specifically, dereferencing a same-document reference must result in an XPath node-set.

2. Apply the transforming algorithms specified in the <Transforms> one by one on the referred data. Some of transforms (e.g., XPath filtering) require the input be an XPath node-set while others require an octet stream. If the input does not meet the requirement, conversion (i.e., parsing or canonicalization) must be performed before transforming.

3. Apply the digest algorithm, which requires an octet stream as input, over the result of transform pipeline processing to calculate the digest. Compare the digest value against the content of <DigestValue>.

To perform signature validation the <SignedInfo> must be canonicalized first and then the digest and cryptographical signature algorithm is applied over the canonical form to calculate the signature.

From the above algorithm we see that two main data models are involved in the signature validation, XPath node-set and octet stream, and they may be converted back and forth within the transform pipeline and the final digest algorithm, as shown in Figure 1. The XPath node-set is recommended due to its ability to express a subset of the elements in an XML document. The straightforward implementation would represent the node-set via a set of references to DOM nodes. Thus, signature validation may require a transformation sequence which involves multiple conversions back and forth between DOM and an octet stream. It is well known that building DOM is very expensive and memory consuming, and, as we will show, the canonicalization over a DOM is even more expensive. Therefore the validation algorithm will definitely incur a big performance penalty if the XPath node-set adopts a DOM tree as its backing store.

2.2 SOAP Digital Signature and WS-Security

SOAP Digital Signature [14] is an application of XML Signature, and it uses the XML Signature syntax to sign SOAP 1.1 messages [12]. A SOAP header entry <Signature> is defined to contain a single digital signature conforming to the XML-Signature specification. As a superset of SOAP Digital Signature, WS-Security [2] provides more enhancements for security tokens. For simplicity in this paper we focus on the SOAP Digital Signature.

Figure 2 illustrates the usual message template of a signed SOAP message, in which the <head> of the SOAP envelope consists of the signature value as well as references to the data object in the <body>. Although the signed SOAP message could be more complicated (for example the whole SOAP message except its <SignedInfo> element can be signed), based on our experience the above simple structure can satisfy a large number of SOAP applications, which only require the entire in-

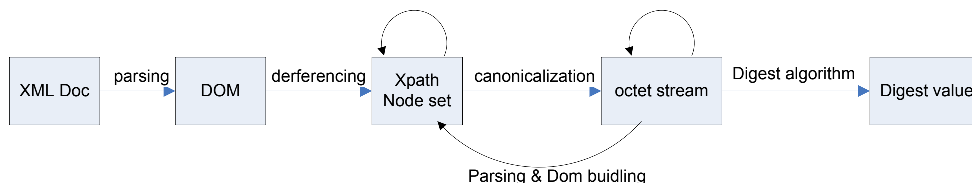

Figure 1. Data flow in the recommended XML signature processing pipeline.

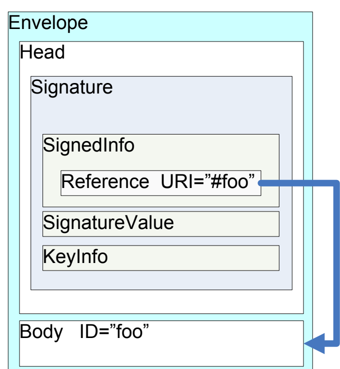

Figure 2. Structure of a general SOAP Digital Signature Message.

vocation data be signed. We call the above simple structure the Streamable Template, which basically defines two restrictions on the SOAP Digital Signature

- The `<Reference>` element precedes the referred-to data.
- Only streamable transformations are allowed in `<Transforms>`.

By streamable transformations we mean that the transformation algorithm can generate correct partial output from a partial input. As we will show later, canonicalization can be streamable and the XPath filtering also could be streamable with streaming XPath query technology [4].

2.3 Canonicalization

The XML 1.0 Recommendation [16] and the Namespaces in XML Recommendation [11] define multiple syntactic methods for expressing the same information. For example, the order of attributes in a start tag of an element can vary without impacting the content of the element. In other words an XML document can have different octet stream representations whose contents are the same. An octet-stream-based application (e.g., signature calculation) will be affected by these superficial differences, which can be introduced easily during the XML parsing and serialization, even though they have no impact on the information content of the document.

XML canonicalization [13],which is popularly abbreviated to C14N, is designed to be useful in these applications. It is a set of rules for converting an XML document or a document subset into its canonical form which is guaranteed to be identical if and only if the content of the document is identical. Thus those applications can process the canonical form instead of the original document so as to protect themselves from the superficial changes. Some canonicalization rules are listed as below.

- Empty elements are converted to start-end tag pairs.
- Superfluous namespace declarations are removed from each element.
- Default attributes are added to each element.
- Lexicographic order is imposed on the namespace declarations and attributes of each element.

2.4 Motivation for Streaming, Specific Processing

As discussed previously, by the recommended validation model validating an XML signature may incur expensive XML processing and data conversions. Most Grid applications, however, only require a simple XML signing processing without any need for transformations. But even to validate a simply signed message which doesn't have any transformations, from Figure 1 we can see that the recommended validation model will need to parse the XML document, build a DOM, create the XPath node-set from the DOM, and canonicalize the node-set into a canonical octet stream. And the recommended signature validation processing will adopt similar steps. As a result at least a DOM tree, a node-set, and two canonical forms are built and kept in the memory. When the message size is large, the amount

of memory used will quickly become a major performance problem. Transformations will only further exacerbate the problem. It has been shown [7] that implementing XML Signature by the recommended model can degrade the invocation response time of a simple Grid service by orders of magnitude, compared with the one using transport-level security implementation, resulting in a situation in which the canonicalization takes almost 90% of the time.

A streaming validation model, especially the streaming canonicalization, can save us from building those expensive data structures during the processing. Moreover since most signed SOAP messages conform to the Streamable Template, we propose a Streamable-Template-specific signature validation model based on streaming canonicalization. Our goal is to learn how much performance we can gain when we optimize processing for the most common cases.

3 Architecture

In our system, we adopt a streaming processing model for the canonicalization. In contrast to the DOM based implementation in which a full DOM is required, a streaming model is able to generate part of the canonical XML as soon as the needed information is available. As we will show later generating canonical form for a XML token (e.g., start tag) only needs limited syntactic and context information which can be obtained from the XML parsing context explicitly or implicitly, so we can utilize the partial context during the XML parsing for the streaming canonicalization. SAX could be the candidate to implement the streaming canonicalization model, but it requires the parsing context (e.g., the namespace context) to be copied to the SAX application, which is inefficient. To get the higher performance, we embed the canonicalization functionality in our XML parser, called GHPX, so that the canonicalization processing can share the parsing context with the parser, running within the XML parsing concurrently, and the canonical form of an XML segment can be obtained as a by-product of the XML parsing.

GHPX is designed as a generic, high-performance XML parser with inner streaming canonicalization support and it is implemented in C++. GHPX provides an interface similar to SAX, but tailored to streaming canonicalization. The application will get a sequence of events as well as the result of streaming canonicalization (i.e., the canonical data or its digest).

The Streaming Soap Signature Validator (SSSV) is the Streamable-Template-specific signature validator, it acts as an intermediary node between the GHPX and the real application in the chain of responsibility of the events issued by GHPX. SSSV checks those SOAP Signature syntax related events and validates the signature. Meanwhile SSSV will forward every event it has received to the application immediately no matter whether the signature validation is finished or not. As the result the application will receive a number of events which are in the scope of the signature when the signature has not been validated yet. For those events SSSV will annotate them to be *unvalidated*. Therefore, it is the application's responsibility to avoid doing some unrecoverable actions when the *unvalidated* events are received.

Figure 3 shows the basic architecture and data flow of the above streaming processing model.

4 Implementation

4.1 XML Parser

The XML 1.0 grammar [16] is a simple but well-designed context free grammar. The tag notation in XML syntax makes it easy to build the parser by the recursive-descent method, in which a procedure is associated with each nonterminal of the grammar and the set of procedures are executed recursively to process the input in a top-down syntactic analysis.

Thus in GHPX for every non-terminal of the XML 1.0 syntax grammar, we first create transition diagrams based on its production rules. For example, for the below productions for Element, EmptyElemTag, and STag:

element ::= EmptyElemTag | STag content Etag

EmptyElemTag ::= '<' Name (S Attribute)* S? '/>'

STag ::= '<' Name (S Attribute)* S? '>'

their transition diagram is shown in Figure 4 after proper left-factoring.

Once the transition diagrams are completed, the recursive procedure for the non-terminal are constructed by applying the below rules [9]:

- A transition on a terminal means we should take that transition if that terminal is the next input symbol.
- A transition on a non-terminal A is a call of the procedure for A.

To support the Namespace in XML [11] specification, GHPX maintains a stack of namespace definitions to record the namespace declarations during the parsing.

GHPX adopts a SAX-like interface. For each opening (closing) tag of an element, it generate a begin (end) event. The begin event of an element comes with a list of (attribute name, attribute value) pairs. For text content enclosed by the opening and closing tag, the parser generate a text event.

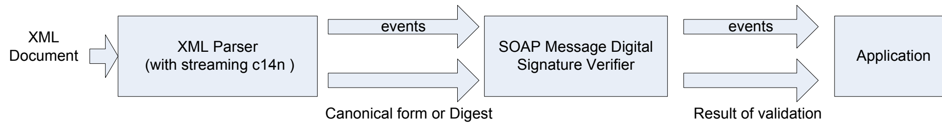

Figure 3. The architecture of the streaming validation model.

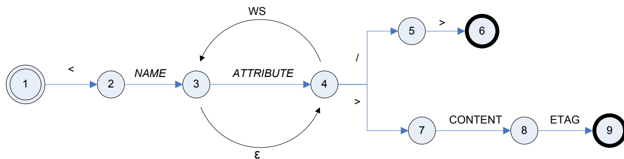

Figure 4. Transition Diagram for Element, EmptyElemTag, Stag.

4.2 Streaming Canonicalization Processing

XML canonicalization [13] defines a set of rules to convert the original XML document into its canonical form. Depending on the information needed by a given canonicalization rule, they can divided into two categories:

- syntax-based rule
- context-based rule

By syntax-based rule, we mean that what the conversion needs is just the current, local syntactic context, such as the current state in the transition diagram. Line-break normalization, attribute value normalization and empty element conversion belong to this category. In contrast, the context-based rules need the parsing context, such as the namespace context and DTD type context. For example, the lexicographical ordering of the attributes requires the corresponding URI of a namespace prefix, and adding the default attribute value requires the DTD definition. Fortunately, according to XML syntax, both the namespace context and the DTD type context should be available when a context-based rule is executed. In GHPX the current status of the transition diagram implicitly provides the needed information for all the syntax-based canonicalization rules, while the namespace stack and DTD parsing result maintained by GHPX provide the explicit information for the context-based canonicalization rules. This property gives us a sufficient condition to perform streaming processing.

To embed the canonicalization processing within the XML parser, we associate the transitions in the transition diagrams with the canonicalization related actions When the transition is triggered during the parsing, its associated canonicalization action will also be executed as a side-effect of the state transition. The output of the action (i.e., the canonical form of the syntax token) will be saved into a buffer.

For example the transition diagram in the Figure 4 will be augmented to the one in Figure 5 to correctly canonicalize the start-tag in the XML document. Those associated actions are explained as below:

- Canonicalization requires all the namespace declarations to precede (in document order) all the other attributes in a start tag. So at State 4, when it is determined whether or not an attribute is a namespace declaration, the parsed value will be put into either the namespace buffer or attributes buffer.
- Canonicalization requires that superfluous namespace declarations be removed and that lexicographic order be imposed on the namespace declarations and attributes. So at State 5 or 7, when all the attributes and namespace declarations in the start tag have been parsed and buffered respectively, all the prefixes appearing in the start tag can be resolved and superfluous namespace declaration can be checked and be removed. We then sort and output the namespace declarations and attributes in lexicographic order respectively.
- Canonicalization requires that empty elements are converted to start-end tag pairs. So when State 6 is reached, the extra “</$name>” will be outputted.

The above example shows that it is not hard for the XML parser to perform canonicalization based the information it has available.

Consequently, after the parsing the canonical form of the XML fragment can be produced as a by-product. The need to build an expensive data structure is eliminated via the streaming model. It should be noticed that XPath node-set is recommended as the input XML document or document subset for XML canonicalization. Unfortunately, fully supporting XPath query is challenging for streaming canonicalization processing. However streaming XPath query technology [8, 18, 4] integrated with the streaming canonicalization processing will be a promising solution. But as we

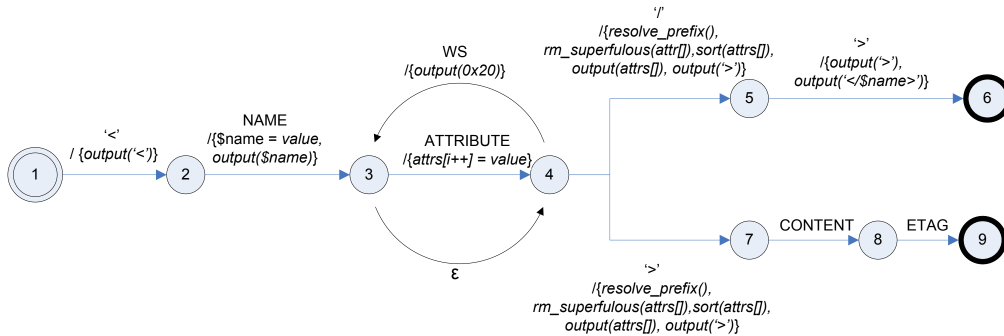

Figure 5. Transition Diagram augmented by canonicalization processing actions.

have stressed, we believe that most SOAP applications usually need only a simple XPath node-set to represent the input data model. In those cases a simple mechanism, which is introduced later, will suffice and be economical.

4.3 C14N Session

A C14N session is defined as the period during which the parser will generate the canonical form for every XML token it is fed. GHPX provides an interface to allow the application to decide when to open or close the C14N session. Once the C14N session is opened, the following XML fragment will be parsed and canonicalized automatically until the session is closed and the result of canonicalization will be saved into a buffer. An application can set a session option, *DIGEST-ONLY*, to indicate what it wants from the canonicalization: digest or canonical form. For XML security applications, the digest of the message segment is what they are really interested in. In that case, GHPX will keep a fixed-size chunk buffer to save the output of canonicalization processing. Whenever the buffer is full the message digest algorithm (e.g., SHA1 [3]) is invoked to calculate the digest incrementally and the chunk buffer will be reused for the following canonicalization. The incremental digest calculation algorithm makes the space complexity of the system constant, regardless the size of the XML message.

The C14N session provides a flexible facility for streaming canonicalization. If the application opens the C14N session at the beginning of document and closes it at the end of document, we will get the canonical form of the entire document. If an application opens the C14N session whenever an special open tag is encountered, say <foo>, and closes it when the close tag </foo> is encountered, then the result is the canonical form of the node-set of the XPath expression “//foo” if there is not a recursive structure. More generally, if the application implements the streaming XPath evaluation model, such as [4], and controls the C14N session properly, a canonical form of most XPath evaluation results can be obtained via the stream.

4.4 Streaming SOAP Signature Validation

Recall that a large number of SOAP messages conform to our Streamable Template. This structure provides an opportunity for streaming signature validation since all the metadata will be encountered before the signed data in the document order. The Streaming Soap Signature Verifier (SSSV) is designed specifically for this Streamable Template. It runs as an automaton to recognize the syntax of Soap Digital Signature and to validate the signature. Just as in the GHPX parser, the transitions in the automaton are also associated with actions. But those actions are SOAP Signature syntax oriented instead of XML syntax oriented. The basic structure of the automaton is shown in Figure 6. The crucial transitions are listed below in the processing order.

- At State 1 when <SignedInfo> is encountered, the C14N session will be opened since we need a canonical <SignedInfo>...</SignedInfo> fragment over which the cryptographic signature algorithm will be applied. (Note at this time we do not know exactly what canonicalization rule will be used, but it can be solved by buffering.)
- From State 2 to State 6, the metadata (such as CanonicalizationMethod or SignatureMethod) about the signature/digest calculation will be saved into temporary variables for later computation. Especially when <Reference> is encountered, its value of the “URI” attribute will be saved for the later locating the signed data object in the following stream.
- At the State 6 when </SignedInfo> is encountered, the C14N session will be

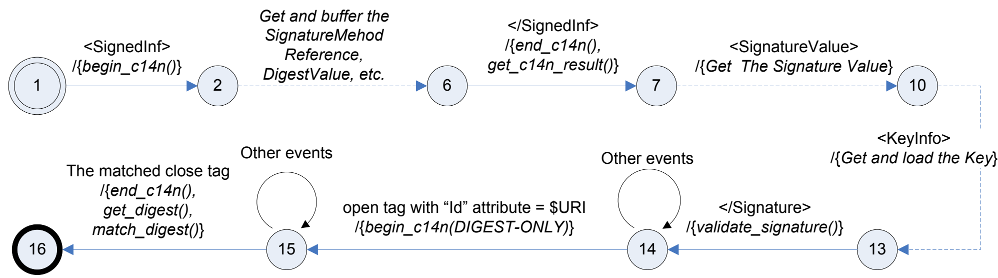

Figure 6. Streaming SOAP Signature Validation Model.

closed. And the the result of the canonical `<SignedInfo>...</SignedInfo>` fragment can be retrieved from the parser.

- At the State 13 when the `</Signature>` is encountered, the signature generation algorithm (e.g., RSA-SHA1) will be invoked to calculate the signature over the canonical form of the `<SignedInfo>` element, and the result will be compared against the one retrieved from `<SiganureValue>` for signature validation.

- At the State 14 when the element with the matched the ID attribute is encountered, a new C14N session will be set up again and this time the *DIGEST-ONLY* option will be set since only digest of the canonical form of the fragment is needed.

- The new session will be opened until the corresponding close tag is encountered. Then we can retrieve the digest from the parser and compared it against the one we get from `<Digest>` for reference validation.

After running the automaton, we verify (1) the correctness of SOAP Digital Signature syntax in the message and (2) the signature validation and reference validation of the signed message.

Here we assume there are no transforms in the signature template, and essentially the above model just dereferences the signed data. However as discussed previously, SSSV could be augmented with streaming XPath query to further support most XPath filtering transforms in a streaming fashion. Essentially SSSV has generated an implicit node-set, whose backing store is the sequence of SAX events rather than a DOM, for the signed data object; and the C14N session is applied on it in streaming fashion. This specific streaming model saves us from building an explicit node-set with the DOM as backing store.

5 Measurements

We conducted the experiments on a Pentium 4 2.8 GHz machine with 2 GB memory running the 2.6.9-gentoo-r9 distribution. We compare the GHPX system with Libxml2 (2.6.15) [10], which is a well-known fast XML C parser and toolkit developed for the Gnome project. Also we compare the GHPX/SSSV validation system with with XML Security Library (version 1.2.6) [6], which implements the XML Signature specification. The reason for using the XML Security Library is that its kernel functionalities, such as parsing and canonicalization, are based on Libxml2, which makes the comparison consistent.

5.1 Performance of Parsing

We first compare the performance of parsing between GHPX and Libxml2 to show the upper bound of the message validation. Both GHPX and Libxml2 use the SAX2 API to parse the data, and the call back functions of SAX2 API will almost do nothing. The framework for the test SOAP signature messages is given in the Appendix, which conforms to the Streamable Template. The body of the SOAP envelope is filled simply with an array of elements, in which each element simply contains an double value and two random attributes.

Figure 7 shows the result of the comparison of the parsing performance when they parse the messages whose number of elements ranges from 0 to 10000. We can see that Libxml has a slightly better performance than GHPX when using SAX API. We believe the major factor is GHPX's recursive parsing style, which makes a function call for every grammar symbol.

5.2 Performance of Canonicalization

Canonicalization was considered the bottleneck of the secure Web Service system. We next describe the experiment to show how much performance we can obtain by the

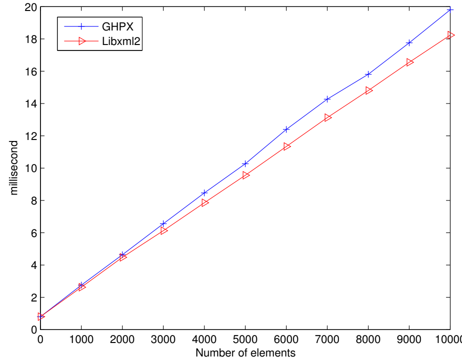

Figure 7. Parsing performance on SAX API.

streaming canonicalization model. Libxml2 provides the canonicalization functions which are DOM based. In other words, to canonicalize an XML fragment using Libxml2 we first need to parse the XML document into a DOM, then do the canonicalization function. So the time needed by Libxml2 consists of a parsing portion and a canonicalization portion. For GHPX to canonicalize the whole document, we only need to open the C14N session in the start-document event handler and close it in the end-document event handler and its time of parsing has already included the time of canonicalization due to the streaming model. Both of them are set to be non-exclusive canonicalization and with comments. The test SOAP messages are the same as the ones used in the parsing experiment.

From Figure 8, we see that the advantage of streaming canonicalization is very significant. The improvement ratio is around 4-5 times. This implies that although the parsing performance of Libxml2 is slightly better than GHPX, building a complete DOM takes lots of the time and offsets all the performance gain brought by the well-implemented XML parser of Libxml2. The canonicalization takes even more much time than the time needed by parsing when the message size grows. Obviously the traversing and manipulation of the large DOM by the canonicalization rules incurs a huge performance penalty. Thus, DOM-based canonicalization generally cannot scale up to process large XML files due to the processing costs. Memory usage also limits its scalability, which will be shown later. In contrast, the streaming canonicalization makes the performance acceptable even for larger XML messages.

5.3 Performance of Signature Validation

To measure the performance of signature validation, we still use the test SOAP messages in the previous experiments. But they are first signed via the XML Security Library with an RSA private key. When validating the signature, a corresponding RSA public key is provided to the test program. OpenSSL is adopted as the basic cryptography library in the test cases for both GHPX and XML Security Library.

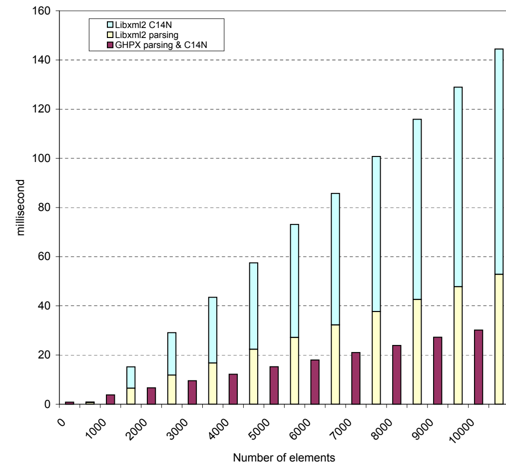

Figure 8. Canonicalization performance.

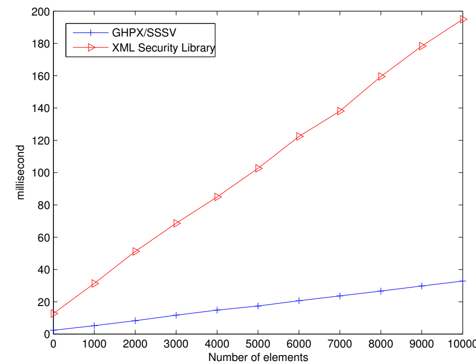

Figure 9. Performance of signature validation.

Figure 9 shows the comparative results of validation performance by the two systems. From that we can see the GHPX/SSSV is much faster than XML Signature Library for SOAP Signature validation. The improvement ratio, which is about 6-7, is even more than the one of the canonicalization test. The extra performance gain is contributed by the Streamable-Template-specific SOAP signature validation algorithm, which eliminates the need for explicit XPath node-set.

We also test the memory usage for GHPX/SSSV and XML Security Library. Figure 10 shows the memory usage reported for validating above signed messages set. From the figure we see XML Security Library uses memory roughly

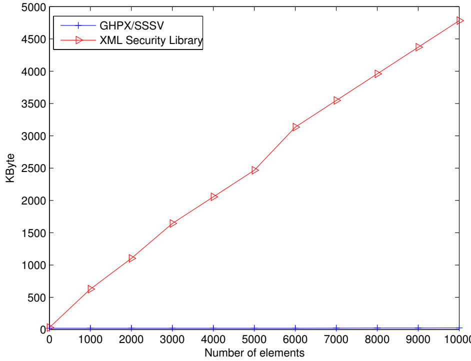

Figure 10. Memory usage of the signature validation.

linear in the size of the input data. Linear memory usage, which a constant factor of 8-9, shows how expensive a DOM and the DOM based XPath node-set are, and makes those non-streaming implementations really unsuitable for large XML files. In contrast, recall that GHPX/SSSV just needs a chunk of buffer to save the partial canonicalization result for the incremental digest algorithm regardless of the size of the input data. In this test the chunk size and the buffer for the canonical `<SignedInfo>` are all set be 1K bytes. Together with other auxiliary memory usages, the total usage of GHPX/SSSV is almost constant and negligible compared with the one of the XML Security Library.

6 Conclusion and Future Work

In this paper, we have described the design and implementation of the GHPX/SSSV system for the streaming validation of SOAP Digital Signature. A distinguishing feature of GHPX is that it supports streaming canonicalization in the parser. Furthermore, SSSV provides a optimized processing model for the most common SOAP messages. Our empirical study shows that streaming canonicalization and simplified validation processing can improve the system performance and scalability significantly, hence being a crucial step toward the practical message level security for Grid applications. To provide better compatibility, we plan to integrate the streaming XPath query with GHPX/SSSV in order to support most XPath filtering transform. Also to investigate how much a Web/Grid service can take advantage of the streaming validation, we are going to integrate the streaming validation within the current Web/Grid service implementation. Our ultimate interest is to make the message level security be practical for Grid system.

Acknowledgment

We would like to thank Satoshi Shirasuna and Liang Fang for useful comments and discussion.

References

[1] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. The physiology of the grid: An open grid services architecture for distributed systems integration. In *Open Grid Service Infrastructure WG*, 2002.

[2] IBM. Web Services Security (WS-Security). `http://www-106.ibm.com/developerworks/webservices/library/ws-secure/`, 2002.

[3] N. I. of Standards and T. Technology. *FIPS Publication 180: Secure Hash Standard (SHS)*, May 11, 1993.

[4] F. Peng and S. S. Chawathe. XPath queries on streaming data. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 431–442, New York, NY 10036, USA, 2003. ACM Press.

[5] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[6] A. Sanin. XML Security Library project web page. `http://www.aleksey.com/xmlsec/`, 2004.

[7] S. Shirasuna, A. Slominski, L. Fang, and D. Gannon. Performance comparison of security mechanisms for grid services. In *5th IEEE/ACM International Workshop on Grid Computing*, 2004.

[8] T.J.Green, G.Miklau, M.Onizuka, and D.Suciu. Processing xml streams with deterministic automata. In *The 9th International Conference on Database Theory*, Siena, Italy, 2003.

[9] A. V.Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[10] D. Veillard. Libxml2 project web page. `http://xmlsoft.org/`, 2004.

[11] W3C. Namespaces in XML. `http://www.w3.org/TR/REC-xml-names/`, 1999.

[12] W3C. Simple Object Access Protocol (SOAP) 1.1. `http://www.w3.org/TR/2000/NOTE-SOAP-20000508/`, 2000.

[13] W3C. Canonical XML 1.0. http://www.w3.org/TR/2001/REC-xml-c14n-20010315, 2001.

[14] W3C. SOAP Security Extensions: Digital Signature. `http://www.w3.org/TR/2001/NOTE-SOAP-dsig-20010206/`, 2001.

[15] W3C. XML-Signature Syntax and Processing. `http://www.w3.org/TR/xmldsig-core/`, 2002.

[16] W3C. Extensible Markup Language (XML) 1.0 (Third Edition). `http://www.w3.org/TR/2004/REC-xml-20040204/`, 2004.

[17] V. Welch. Globus toolkit version 4 grid security infrastructur: A standards perspective. `http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/security/GT4-GSI-Overview.pdf`, 2004.

[18] Y.Diao, P.Fischer, and M.J.Franklin. Yfilter: Efficient and scalable of xml document. In *The 18th International Conference of Data Engineering*, San Jose, 2002.

APPENDIX

A template of the simple Soap Digital Siganture

```
<?xml version="1.0" encoding="UTF−8"?>
<SOAP−ENV:Envelope
 xmlns:SOAP−ENV="http://schemas.xmlsoap.org/soap/envelope/">
 <SOAP−ENV:Header>
  <SOAP−SEC:Signature
   xmlns:SOAP−SEC="http://schemas.xmlsoap.org/soap/security/2000−12"
   SOAP−ENV:actor="some−URI"
   SOAP−ENV:mustUnderstand="1">
   <ds:Signature xmlns:ds="http://www.w3.org/2000/09/xmldsig#">
    <ds:SignedInfo>
     <ds:CanonicalizationMethod
      Algorithm="http://www.w3.org/TR/2001/REC−xml−c14n−20010315">
     </ds:CanonicalizationMethod>
     <ds:SignatureMethod
         Algorithm="http://www.w3.org/2000/09/xmldsig#rsa−sha1"/>
     <ds:Reference URI="#Object">
      <ds:DigestMethod
              Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
      <ds:DigestValue>
      </ds:DigestValue>
     </ds:Reference>
    </ds:SignedInfo>
    <ds:SignatureValue></ds:SignatureValue>
   </ds:Signature>
  </SOAP−SEC:Signature>
 </SOAP−ENV:Header>
 <SOAP−ENV:Body
  xmlns:SOAP−SEC="http://schemas.xmlsoap.org/soap/security/2000−12"
  xml:id="Object">

 </SOAP−ENV:Body>
</SOAP−ENV:Envelope>
```