

Improving WS-Security Performance with a Template-Based Approach

Satoshi Makino, Michiaki Tatsubori, Kent Tamura and Yuichi Nakamura

IBM Tokyo Research Laboratory

1623-14, Shimo-tsuruma, Yamato, Kanagawa 242-8502 Japan

Abstract

The poor performance of WS-Security (WSS) processing is often a topic of concern and prevents its wider adoption. We focused on byte-level similarities in WSS messages and implemented a template-based WSS processor. Inside the processor an automaton is employed that matches the incoming messages and extracts signature values and/or encrypted values. WSS operations including XML Canonicalization are performed against the extracted values, without costly XML parsing and traversal. This is more than twice as fast as the DOM-based WSS processor and our prior work with a stream-based processor.

Keywords: Web services, WS-Security, performance

1. Introduction

As Web services become globally accepted, there are strong demands to provide services that are secure and that can be used in a secure manner. To do this, the Web Services Security [3] (also known as WS-Security or WSS) and some related specifications were released in March 2004. WSS defines some XML data structure describing digital signatures, encryption, etc. on top of SOAP messages. Using WSS, end-to-end message-level security is possible as opposed to peer-to-peer security [7]. It is now possible for specific parts of the SOAP messages (e.g. credit card numbers) to be digitally signed or encrypted all the way through from the client to the service provider, even if intermediary nodes relay the messages. Peer-to-peer security such as SSL cannot do this because the intermediaries can decrypt the messages.

On the other hand, the use of WSS imposes a certain overhead on SOAP communications [10], in addition to the overhead from the verbosity of XML [9]. In order for WSS to be widely accepted, its performance improvement is highly necessary.

Standing on that understanding we implemented a stream-based WSS processor in [1]. While most developers chose DOM to implement WSS because of the ease of programming, we implemented a SAX-based streaming WSS processor in pursuit of performance. It showed fairly good performance in comparison to the DOM-based processor.

We have perceived the hints to further performance improvements from the comparison of WSS and SSL. According to [8], the overhead of SSL data transfer is in order of 0.1ms while WSS is in order of 1 to 10ms. This mainly comes from costly XML parsing in WSS. While SSL does not care about the payload content, WSS processors have to parse the content before the actual processing. XML parsing includes scanning the message from the beginning to the end, checking whether invalid characters are used, and generating some objects corresponding to the message such as a DOM tree and SAX events. In addition, WSS involves many other XML-intensive operations. If the message has been encrypted, decrypted content must be parsed again. XML Canonicalization [4] (C14n) is also the source of a lot of overhead, since it again scans the entire XML tree structure.

From these observations we came to the idea that we should skip as much of the XML processing as possible. We cannot get rid of XML processing insofar as we use the known XML APIs including DOM and SAX. Although the stream-based processor does perform better than the DOM-based one, it only replaced DOM with SAX and it is still performing costly XML processing. We should treat the message as a mere byte array, rather than relying on DOM or SAX. This paper describes our new high-performance WSS processor that utilizes a template-based approach. Templates consist of byte arrays and stored in an automaton. Incoming messages are matched against the automaton and WSS processing is performed based on the matching result, without relying on DOM or SAX APIs. Section 2 clarifies our motivations to

implement the template-based WSS processor through an analysis of the characteristics of WSS messages. Section 3 describes the processor in detail. Section 4 shows its performance measurements and its usefulness through the comparison with DOM- and stream-based WSS processors. Section 5 introduces some related work and Section 6 concludes this paper.

2. Motivation

We found that most SOAP messages have similar byte representations. SOAP messages created by the same implementation have the same message structure. In addition, the number of SOAP implementations is relatively small and the number of WSS implementations is even smaller. Therefore it seems that the number of message patterns with which WSS processor has to deal should be very small. This is especially more likely in business-to-business scenarios, where the message patterns are very limited and might be known to the server beforehand. In such cases, it is inefficient to repeatedly parse the almost-same messages each time one arrives.

```
<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope xmlns:xsd="..."
  xmlns:SOAP-ENV="..."
  xmlns:xsi="...">
  <SOAP-ENV:Body>
    <ns1:getQuote xmlns:ns1="...">
      <ns1:symbol>XXX</ns1:symbol>
    </ns1:getQuote>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Figure 1: StockQuote message created with Apache Axis

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="..."
  xmlns:xsd="..."
  xmlns:soap="...">
  <soap:Body>
    <getQuote xmlns="...">
      <symbol>XXX</symbol>
    </getQuote>
  </soap:Body>
</soap:Envelope>
```

Figure 2: StockQuote message created with .NET

We also found that the application has far less interest in the literal message structure than the application-defined data structure. Figure 1 and Figure 2 show the StockQuote message examples emitted by

Apache Axis [6] and .NET respectively. (Some parts are omitted for readability.) For both messages, the StockQuote application focuses on the string “XXX” rather than on the associated namespace prefixes, etc. In other words, what is important for the application is returning the stock value corresponding to the symbol XXX, but not the parsing of the entire SOAP envelope. This idea is applicable to WSS processors, as well. That is, signing and encryption can be done by just extracting the relevant values as text, without XML parsing.

Therefore, we came up with the approach of performing byte-level matching against the incoming messages and extracting the relevant values. Byte-level matching is far faster than XML parsing and high-speed WSS processing can be expected.

3. Template-Based WSS Processor

Based on the observations above, we implemented a template-based WSS processor which matches the incoming messages with the given message templates. Here we describe its design and implementation. Note that this paper describes mainly the receiver side operations, although similar processing can be applied on the sender side as well.

3.1 Design Overview

Figure 3 shows an overview of our template-based WSS processor, where the solid and dotted lines denote the processing flow and the additional data flow respectively. First, the incoming message is matched against the automaton, which contains multiple message templates in a merged form. If the message matches any template, the WSS-relevant values such as the signature value and the encrypted value are extracted using the template and WSS processing is done based on the extracted values. Otherwise, the WSS processing is performed by an ordinary DOM-based processor and a new template corresponding to the unmatched message is generated. The new template is merged into the automaton, and thereafter any messages having the same message structure can be matched against the template. The resulting messages are same in both cases.

3.2 Message Template

The message template describes the byte-level structure of WSS messages. It consists of a few kinds of

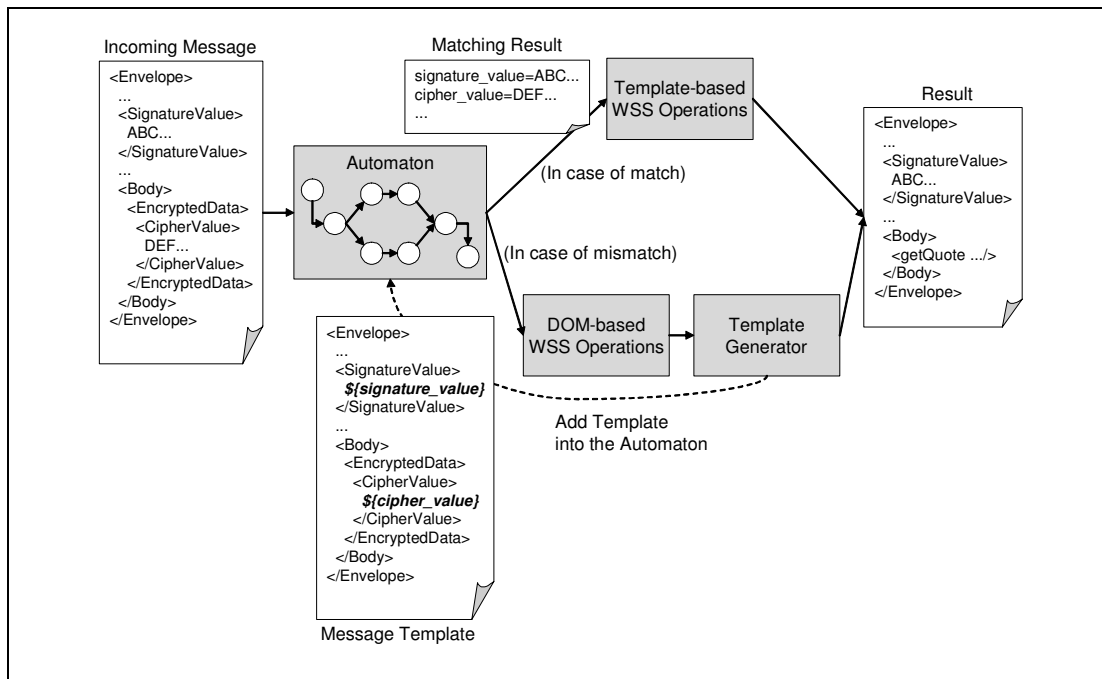


Figure 3: Template-based WSS processing model

message fragments: constant fragments corresponding to the unchanging parts in the messages and variable fragments corresponding to the variable parts. All the WSS-relevant values are regarded as variable parts. In Figure 3 the variable fragments are denoted as “\${variable_name}” and printed in bold italic.

Message templates are generated in cooperation with the DOM-based WSS processor, since the template-based processor initially has no information about the XML tree structure without help from DOM-based processor. In order to tell where the WSS-relevant values start and end, we implemented a specialized XML parser that can record the offset and length information for each XML node. Using the DOM created by the specialized parser, the DOM-based WSS processor can select all of the WSS-relevant nodes and specify them as the variable fragments with the offset and length information. Finally, the message parts that were not selected in the previous process are consolidated as a constant fragment, and a message template object is generated from these fragments. Note that DOM-based operations are performed only when a message with a new structure arrives. Most messages match the existing templates and the frequency of creating templates is low, based on the observations above.

3.3 Automaton

The template-based WSS processor has an automaton

inside. The automaton provides two interfaces, one for matching the incoming messages and one for learning the new message templates. If the incoming message matches any of the existing message templates, the matching result returned is a map object that contains pairs of variable names and the corresponding actual variable values. In Figure 3, for instance, the incoming message is matched against the message template and a map of {signature_value, ABC...}, {cipher_value, DEF...} and so on is returned. The WSS operations (see the next section) are done based on the extracted values. However, if the matching failed, the WSS processor would have had to create a new message template and make the automaton learn that template, in preparation for the later incoming messages for which the new template will be applied.

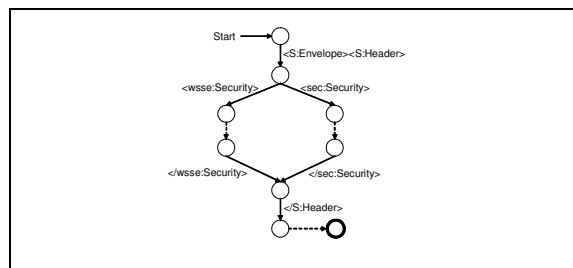


Figure 4: Message templates in an automaton

Learning message templates means updating the state diagram in the automaton. All of the templates inside

the automaton are stored in a merged form and can be represented as a state diagram. Figure 4 shows a simple example of a state diagram which has been created from two message templates, where a solid line and a dotted line denote an actual state transition and one or more transitions (intermediate states might be omitted), respectively. Concretely in this example, this means that “<S:Envelope> <S:Header> <wsse:Security> ... </wsse:Security> </S:Header> ...” and “<S:Envelope> <S:Header> <sec:Security> ... </sec:Security> </S:Header> ...” have already been learned. Since “<S:Envelope> <S:Header>” and “</S:Header> ...” are common to both messages, their paths are merged. As seen in the figure, one XML node (or tag) does not necessarily correspond to one state.

3.4 WSS Operations

After the WSS-relevant values have been extracted from the message, there are no substantial differences in the WSS operations themselves between the template-based WSS processor and the DOM-based one. They differ in the way they retrieve the values from the message. The template-based processor uses template matching while the DOM-based one traverses DOM tree.

For decryption, the content of the EncryptedKey element is first decrypted if present. All of the information necessary for decryption such as decryption key and algorithm has already been extracted from the message and stored in the map object. The decrypted key is used for decrypting the content of EncryptedData. If EncryptedKey is not present, the key for decrypting EncryptedData is directly specified in EncryptedData element itself. After EncryptedData is decrypted with the key, the byte array corresponding to the EncryptedData element is replaced with the decrypted value.

For signature verification, the signed message part (SOAP Body etc.) is canonicalized per the XML Canonicalization (C14n) specification. (For the details of C14n see the next section.) Then the output of C14n is digested using the algorithm specified in DigestMethod and verified with the value in DigestValue. If the verification succeeds, the SignedInfo element is canonicalized and verified with SignatureValue, using the algorithm specified in SignatureMethod.

3.5 XML Canonicalization

C14n is defined for masking any literal differences in the message and avoiding damage to the signature’s validity during the message processing. It includes the following operations:

1. Rewriting empty tags into the pair of start and end tags
2. Character-level normalization (i.e. rewriting ' into ’)
3. Reordering of attributes and namespace declarations
4. Adding and/or removing namespace declarations

Step 4 is impossible without help from a DOM-based WSS processor, since namespace information defined in the ancestor nodes is required. Therefore we introduced a concept of a pre-template and post-template for C14n. These templates are created by the DOM-based WSS processor for the first message, and after that the template-based processor can use them for the messages which have the same structure. Again, the DOM-based processor is invoked only once for a message structure and its overhead can be regarded as negligible over the long term, according to our assumptions.

The C14n pre-templates are used for matching with the incoming messages. All the text nodes and attribute nodes (not including namespace declarations) are considered to be variable parts in the template. On the other hand, C14n post-templates represent the results of C14n. Each variable part in the pre-template corresponds to the one in the post-template with the same name, but their ordering might be changed, according to Steep 3 in the above list. The results of Step 1 and 4 are also described in the post-templates. Finally, the values that have been matched with the pre-template are normalized according to Step 2 and output into the positions specified in the post-template.

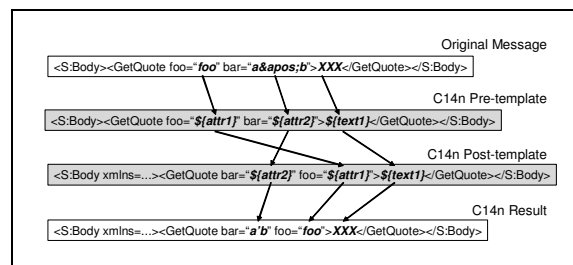


Figure 5: Template-based C14n operation

Figure 5 shows an overview of the template-based

C14n operations, where the bold italic texts denote the variable parts. The positions of “foo” and “bar” attributes are swapped, since they have to be in alphabetical order according to the C14n specification. In addition, appropriate namespace declarations are added into S:Body element and ' is normalized to an apostrophe.

Character normalization requires scanning the entire string and has processing costs much like XML parsing. It is possible for each variable part in the C14n templates to have a sub-automaton and the result of the normalization is associated with each node in the sub-automaton. Then the costly text scanning can be replaced with fast byte matching, but this is based on the assumption that the same text arrives many times. If the text differs every time, the text has to be added into the sub-automaton each time and never matches. However, this approach does make sense if the variation of text is limited, such as in the format of xsd:boolean and xsd:enumeration defined in W3C XML Schema [5]. Since the decision of whether or not to use sub-automaton might require schema information about the message structure, we have deferred this to future work.

4 Evaluation

We conducted some performance measurements comparing three types of WSS processor (template, stream, DOM) and the results showed that the template-based processor performs best. We also analyzed the properties of the template-based processor and show the advantages over the other two processors. First we measured the processing time of each WSS processor while varying the message size. Each processor accepts a byte array containing a SOAP message and processes it. The measurements throughout this paper were done using a Pentium-4 2 GHz workstation with 2 GB of main memory. Symmetric key algorithms were used for both encryption (Triple DES) and signature (HMAC) in all cases. Figure 6(a) and Figure 6(b) show the results for the sender-side and the receiver-side respectively, where the X-axis denotes the number of variable text nodes in the SOAP message. We refer to this parameter as “message size” throughout this paper, while the actual message size (in bytes) is around 65 times bigger than the specified number. The Y-axis shows the processing time.

4.1 Performance Measurements

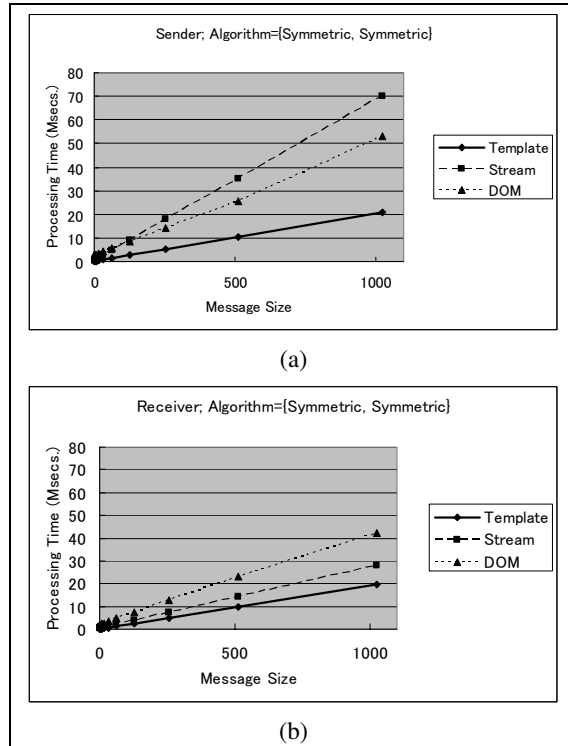


Figure 6: Performance with various message sizes

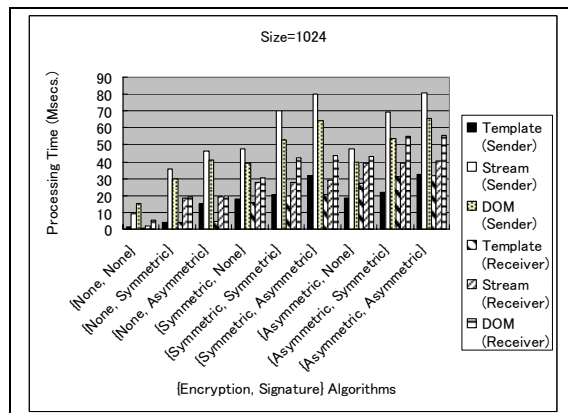


Figure 7: Performance with various cryptographic algorithms

We also measured the processing time while varying the signature and encryption algorithms. In addition to the symmetric key algorithms, asymmetric key encryption (RSA-v1.5) and signatures (RSA) are widely used for WSS. We measured all of these combinations and the results are shown in Figure 7. Here the X-axis denotes the combination of encryption and signature algorithms. For example, {None, Symmetric} means that the message is not encrypted but signed using the symmetric key algorithm. The

Y-axis again denotes the processing time. The message size is 1024 in all cases. In asymmetric key encryption, note that the message is encrypted using a randomly generated symmetric key and the key is in turn encrypted using the asymmetric key algorithm, rather than encrypting the message itself using the asymmetric key algorithm.

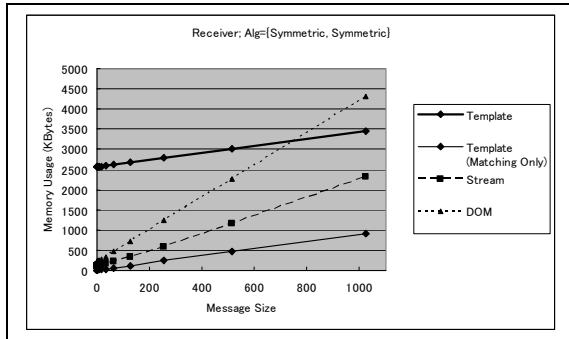


Figure 8: Memory usage with various message sizes

The receiver-side memory usage of each WSS processor is shown in Figure 8. The X- and Y-axes denote the message size and the memory usage (in kilobytes) respectively. Symmetric algorithms are used for both signatures and encryption. Here the thin solid line means the memory usage of the template matching and the related processing, excluding the automaton itself. This amount of memory is consumed each time a message comes. The difference between this and the overall memory consumption (shown with thick solid line) corresponds to the automaton, which is loaded statically on the virtual machine.

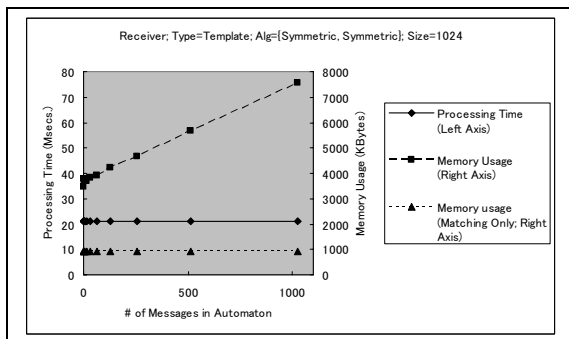


Figure 9: Performance with various automaton contents

The template-based processor memorizes multiple message templates inside an automaton and this may impact the performance. We measured the receiver-side processing time and memory usage when

the automaton contains various numbers of message templates. Figure 9 shows the results, where the X-axis is the number of templates inside the automaton. The left and right Y-axis denote the processing time and the memory usage (in kilobytes) respectively. The message size is 1024 and symmetric algorithms are used for both the signatures and encryption. We also measured the memory usage of template-matching part.

4.2 Discussion

The results show that the template-based WSS processor performs faster than the other two types of processors, for any message sizes and for any algorithm combinations. Its processing time is 29.8% of the stream-based processor at the sender side, 46.5% of the DOM-based processor at the receiver side, and 41.8% of the other two processors in total (sender plus receiver, where the processing time of the DOM- and stream-based processors are almost equal).

Avoiding costly XML parsing, in either the DOM or SAX fashion, contributed most to the high performance of the template-based processor. In addition we identified some DOM- or SAX-specific issues that have lowered the performance of each processor.

Although the stream-based processor performs better than the DOM-based one on the receiver side, it tends to perform worse at the sender side. This mainly stems from event recording at the sender side, since the stream-based processor has to calculate and insert a signature value for the message part (i.e. SOAP Body) that appears after the WSS header element. This is impossible in one-pass processing, and so the processor has to intercept and record all of the SAX events emitted from the parser until it encounters the end of a signed message part. After it reaches that point the processor is able to calculate a signature value and can replay the recorded events for the following event handlers. This topic is discussed again in the next section.

Although in [1] we concluded that the stream-based processor performs better even on the sender side, this turned out to be true only if the message is relatively small. The messages that we used were as small as 64 in terms of the message size parameter used in this paper. For small messages it does perform better even in the measurements in this paper (see Figure 6(a)).

For the DOM-based processor, it performs worse on the sender side than on the receiver side. The reason is

that the DOM tree is updated more frequently on the sender side than on the receiver side. Although the receiver-side processor has to parse the decrypted content again, the frequent DOM updates at the sender side require more time.

The template-based processor is not affected by either SAX event recording, DOM manipulation or repeated DOM parsing. Therefore it can perform the WSS operations with the lowest overhead on both sender and receiver sides.

The template-based processor consumes more memory than the other two processors when the message is small. We admit that the initial memory usage of the automaton (around 2.5 megabytes) is relatively big and are working on its optimization. On the other hand, it is less affected by the message size increase and it will consume less memory even than the stream-based processor when the message is larger.

The additional processing time of matching a message against an automaton with multiple message templates is small enough and can be ignored. This is because we have to compare only the first few bytes with those in each branched state in the automaton to eliminate the mismatching templates. On the other hand, a certain amount of memory (around 4 kilobytes in this case) is consumed for one message template. We regard this as an acceptable cost and the template-based processor would be able to cover the case in which more and more message template has to be used as the WSS specifications and their implementations evolve.

Note that the identical messages were repeatedly passed to each processor during our measurements. This means that the automaton in the template-based WSS processor was never updated. Although such a situation might be unlikely in the real world, we consider that the overhead of updating the automaton is low, based on our assumption that the update rarely occurs and its cost can be ignored over the longer term.

4.3 Advantages over the other types of WSS Processors

Here we identify some properties in template-based processor as advantages over the other approaches.

First, the programming model in the template-based processor is simpler than the DOM-based processor and far simpler than the stream-based processor. The stream-based processor consists of a very complex pipeline of SAX event handlers, such as a signature

handler and an encryption handler. This makes debugging and detailed performance analysis extremely difficult, especially when the handlers are in a complex class hierarchy. Since the same methods such as `startElement()` and `characters()` are repeatedly invoked in a pipelined fashion, it is almost impossible to identify which operation really has a bug or performance bottleneck. For DOM, it requires boring repeated invocations of DOM APIs such as `getElementsByTagNameNS()` to navigate the DOM tree and to retrieve any WSS-relevant values, although it is much easier than for SAX. On the other hand, the template-based processor can retrieve the values easily by looking up a map object, as described in Section 3.

The applicability of the processors to the various types of WSS messages is also worth consideration. The streaming nature of the SAX programming model imposes a serious limitation on the stream-based WSS operations. When a strict streaming programming model used where the SAX handlers cannot see forward or backwards to coming or past events, they cannot generate a message in which the signature value appears before the signed part and they cannot consume a message in which the signature value appears after the signed part. While the former case is very common (the measurements in this paper fall here), the latter is common as well, such as when signing the other SOAP header elements in the message. In order to break this restriction it is necessary to record the events and introduce a large amount of overhead. The template- and DOM-based processors do not have such a limitation and the entire message is always available for them.

The DOM-based processor has the widest applicability, since almost all the WSS-related specifications implicitly assume the DOM programming model. The template-based processor has the second widest applicability. Basically all of the WSS operations that the DOM-based processor can do can also be done with the template-based processor, as far as we know. However we cannot deny the possibility that the new WSS-related specifications might define an operation that is fundamentally impossible for the template-based processor.

5 Related Work

Here we introduce the related works briefly and show the benefits in our template-based WSS processor.

5.1 Template-based SAX Parser

Our approach can be applied to more generic XML processing as well. In [2] we proposed an XML parser that memorizes byte arrays of XML documents and the resultant SAX events. They are stored in an automaton. When the parser reads an XML document and some parts of the document matches a byte array, it reuses the memorized SAX events instead of parsing the entire document again. Only the unmatched parts are parsed, and the byte arrays and corresponding SAX events are also memorized for reuse.

This paper and [2] share the same concept but this paper shows its uniqueness in WSS-specific processing, especially in C14n using pre- and post-templates. In addition, WSS processing enjoys more merit of byte matching than the generic XML processing, since WSS includes some byte-oriented processing such as C14n, encryption and decryption.

5.2 Other Works

Devaram et al. proposed a client-side caching mechanism for SOAP messages [11]. The authors and we work from the similar observations that the XML processing is the main cause of SOAP performance degradations. They implemented a Partial Caching mechanism where some portions in the cache entries can be replaced. Although this approach can avoid caching a large number of messages that have the same message structure but for which some text values differ, the authors admit that the replacement mechanism is not always efficient and it works well only when the messages have only a few parameters.

Abu-Ghazaleh et al. also proposed a mechanism for avoiding repeated serialization of SOAP messages [12]. Like our template-based WSS processor, a notion of “template” was introduced. There a fixed length of byte array is allocated for each variable portion, and the messages are padded with white space and/or shifted according to the actual length of the serialized variable value. However the padding is very troublesome for WSS processing, since the added white space makes the XML signature invalid. This mechanism can be applied only at the sender-side.

6 Concluding Remarks

We implemented a template-based WSS processor and reduced the processing time by around 60% compared to the DOM- and stream-based processors. Retrieving the WSS-relevant values and the C14n processing are

done rapidly by byte matching in an automaton, rather than by costly XML processing. We also showed the programmability and applicability of the template-based processor.

Apparently it would be faster if the entire web services processing becomes template-based. In our current implementation the web services application does parse the message to construct an application object, while WSS processor does not. We are focusing on this problem and implementing a mechanism to build application objects through template-based processing.

References

- [1] S. Makino, K. Tamura, T. Imamura and Y. Nakamura, Implementation and Performance of WS-Security, in *International Journal of Web Services Research*, Vol. 1, No. 1, 2004.
- [2] T. Takase, H. Miyashita, M. Tsubori and T. Suzumura, An Adaptive, Fast, and Safe XML Parser Based on Byte Sequences Memorization, to appear in *Proceedings of the WWW 2005*.
- [3] Web Services Security Core Specification, <http://docs.oasis-open.org/wss/2004/01/oasis-2004-01-wss-soap-message-security-1.0.pdf>
- [4] Canonical XML, <http://www.w3.org/TR/xml-c14n>
- [5] XML Schema, <http://www.w3.org/XML/Schema>
- [6] Apache Axis, <http://ws.apache.org/axis/>
- [7] S. Hada, SOAP security extensions: digital signature, IBM developerWorks, <http://www.ibm.com/developerworks/webservices/library/ws-soapsec/>
- [8] G. Apostolopoulos, V. Peris and D. Saha, Transport Layer Security: How much does it really Cost?, in *Proceedings of the IEEE INFOCOM 1999*.
- [9] M. Govindaraju, A. Slominski, K. Chiu, P. Liu, R. van Engelen and M. J. Lewis, Toward Characterizing the Performance of SOAP Toolkits, in *Proceedings of the IEEE/ACM GRID 2004*.
- [10] H. Liu, S. Pallickara and G. Fox, Performance of Web Services Security, Technical Report, 2004, <http://grids.ucs.indiana.edu/ptliupages/publications/WSSPerf.pdf>
- [11] K. Devaram and D. Andresen, SOAP Optimization via Parameterized Client-Side Caching, in *Proceedings of the IASTED PDCS 2003*.
- [12] N. Abu-Ghazaleh, M. J. Lewis and M. Govindaraju, Differential Serialization for Optimized SOAP Performance, in *Proceedings of the IEEE HPDC-13*, 2004.