Optimized Web Services Security Performance with Differential Parsing

Masayoshi Teraguchi[1], Satoshi Makino[1], Ken Ueno[1], and Hyen-Vui Chung[2]

[1] Tokyo Research Laboratory, IBM Research
1623-14, Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan
{teraguti, mak0702, kenueno}@jp.ibm.com
[2] IBM Software Group
11501 Burnet Rd. Austin, TX 78758-3415, USA
hychung@us.ibm.com

Abstract. The focus of this paper is to exploit a differential technique based on the similarities among the byte sequences of the processed SOAP messages in order to improve the performance of the XML processing in the Web Service Security (WS-Security) processing. The WS-Security standard is a comprehensive and complex specification, and requires extensive XML processing that is one of the biggest overheads in WS-Security processing. This paper represents a novel WS-Security processing architecture with differential parsing. The architecture divides the byte sequence of a SOAP message into the parts according to the XML syntax of the message and stores them in an automaton efficiently in order to skip unnecessary XML processing. The architecture also provides a complete WS-Security data model so that we can support practical and complex scenarios. A performance study shows that our new architecture can reduce memory usage and improve performance of the XML processing in the WS-Security processing when the asymmetric signature and encryption algorithms are used.

Keywords: Web Services, Web Services Security, Performance, XML parsing.

1 Introduction

Service-oriented architecture (SOA) is now emerging as the important integration and architecture framework in today's complex and heterogeneous enterprise computing environment. It promotes loose coupling so that Web services are becoming the most prevalent technology to implement SOA applications. Web services use a standard message protocol (SOAP [1]) and service interface (WSDL [2]) to ensure widespread interoperability even within an enterprise environment. Especially for the enterprise applications, securing these Web services is crucial for trust and privacy reasons and to avoid any security risks, such as malicious and intentional changes of the messages, repudiations, digital wiretaps, and man-in-the-middle attacks. The Web Services Security (WS-Security) specifications were released by OASIS in March 2004 [3].

They describe security-related enhancements to SOAP that provide end-to-end security with integrity, confidentiality, and authentication. As described in [7], WS-Security processing is categorized into two major operations: cryptographic processing

A. Dan and W. Lamersdorf (Eds.): ICSOC 2006, LNCS 4294, pp. 277 – 288, 2006.
© Springer-Verlag Berlin Heidelberg 2006

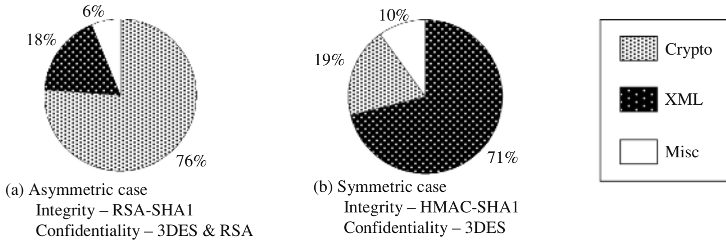

Fig. 1. Analysis of performance contribution of WS-Security processing on a DOM based implementation

and XML processing. In fact, we confirmed that these two operations contribute to the performance overhead of WS-Security processing through a preliminary experiment on a DOM based WS-Security implementation that we have developed using XML Security Suite technology [4] as the basis. The left side of Figure 1 shows the XML processing is the second constraint on performance when asymmetric algorithms are used. The right side of Figure 1 shows the XML processing is the primary limitation on throughput when symmetric algorithms are used.

An interesting characteristic of Web services is that all SOAP messages sent to a service have the almost same message structure. Based on this characteristic, some differential techniques that skip unnecessary XML processing, but which instead do only byte matching, have been proposed. [8][9] focus on reducing the general XML processing overhead (such as parsing, serialization, deserialization, and document tree traversal). In [8], only one template that memorizes the optimized basic structure of the message is constructed in advance. This template is used to extract only the differences between the input byte sequence and the data stored in the template. [9] also uses a single template, but it can be dynamically updated because the parser context is also stored in the template and this allows partial parsing. However it is difficult to apply these technologies to WS-security processing because WS-Security support is out of the scope of [9]. On the other hand, [10] considers improvements of the security-related XML processing (such as canonicalization and transformation). In [10], a message is divided into fixed parts and variable parts. A finite state automaton ("automaton" below) memorizes these parts as the basic structure of the message. But since the parser context is not stored in the automaton, it is impossible to partially parse the message or to optimize the data structure in the automaton.

In this paper, we address many of the problems in that previous works and describe a novel WS-Security processing architecture based on [10]. The architecture divides the byte sequence of a message into fixed parts and variable parts according to the XML syntax in the message and stores them in an automaton. The automaton consists of two parts: the states which store both the parser contexts and the WS-Security contexts, and the transitions which store the corresponding parts. Since the processor can extract the parser contexts from the automaton, it can resume a partial parsing and can dynamically update the data in the automaton without invoking another processor as in [10]. In addition, the data model in the automaton can be optimized even when the same structure appears repeatedly in the byte sequence. We also provide a more complete WS-Security data model relative to the one in [10], so we can support a

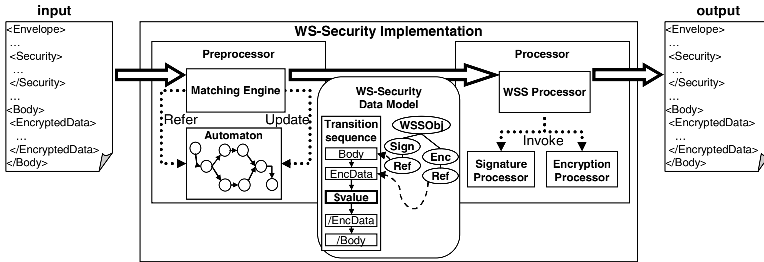

Fig. 2. Architecture of WS-Security processing with differential parsing

wider variety of practical scenarios than [10] covers. In this paper, we also conduct a performance study to evaluate memory usage and performance metrics. The performance study shows that our new architecture can reduce memory usage but retains almost same performance as the existing technology when the asymmetric algorithms are used, even though our method is more practical and more flexible.

The rest of the paper is organized as follows. We describe the details of our new architecture for WS-Security processing in Section 2. We introduce some related work using differential techniques in Section 3. We present our performance study in Section 4. Finally, we conclude the paper in Section 5.

2 WS-Security Processing with Differential Parsing

In this section, we describe a novel WS-Security processing architecture with differential parsing. Figure 2 shows the architecture of WS-Security processing. The architecture has two major components: the WSS preprocessor and the WSS processor. The WSS preprocessor manages an automaton, which has a more flexible and powerful internal data structure than the one described in [10]. It matches the byte sequence of an input SOAP message against the data that was previously processed and stored in the automaton, and constructs a new complete but still lightweight data model for the WS-Security processing. The WSS processor secures the SOAP message using the WS-Security data model. We can support a wider variety of practical scenarios than [10] supports by our new data model.

2.1 Internal Data Structure in an Automaton

Given a new input SOAP message as a byte sequence, the WSS preprocessor invokes its matching engine to match the byte sequence with the ones that were previously processed and stored in the automaton, without doing any analysis of the XML syntax in the message. If a part of the message (or the whole message) does not match any of the data stored in the automaton, then the matching engine parses only that part of the message and dynamically updates the automaton. When the matching engine parses the byte sequence, it is subdivided into the parts corresponding to the XML syntax in the message, according to the suggestion in [11]. Each divided part can be represented as either a fixed part or a variable part in the internal data structure. Figure 3 shows the internal data structure in the automaton. The data structure includes states (*Si* in

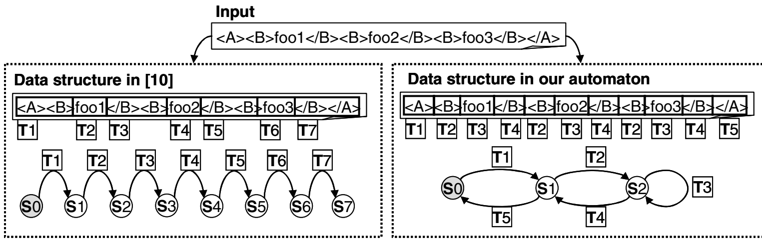

Fig. 3. The internal data structure in an automaton

Figure 3) and transitions (*Tj* in Figure 3). As shown in Figure 3, there is a difference between in the automaton in [10] and in our new automaton. In [10], it can't efficiently handle the same structure that appears repeatedly in the input because it doesn't consider the XML syntax. In Figure 3, the <*B*> elements appear repeatedly but they are stored as different transitions, such as T_1, T_3, T_5, and T_7. On the other hand, our new automaton can efficiently handle that. In Figure 3, the <*B*> elements are stored only in the transitions T_2 and T_4. In our new automaton, the state corresponds with the internal state in a parser and stores the parser context for partial parsing and the WS-Security context for construction of a WS-Security data model. The transition stores one fixed part or one variable part. The transition also stores a reference to the byte sequence, and the byte offset and the length of the snippet in order to get the original byte sequence without any additional concatenation, especially during encryption of the outbound message. The automaton doesn't allow two different states to have the same parser context and the same WS-Security context. This reduces the total memory usage even when same data structure appears repeatedly in the payload of a SOAP message. In Figure 3, we can merge two states into S_1 because there is no difference between the context after processing T_1 and the context after processing T_4. When the context before the processing of a transition is the same as after the processing, then the transition, such as T_3, can be a self-loop.

2.2 Lightweight WS-Security Data Model

[10] uses a very simple data model for WS-Security processing. But this makes it difficult to apply the data model to a wide variety of practical scenarios such as a model including multiple XML signatures, because it consists only of pairs of keys and WS-Security-relevant values. Therefore, we now define the more concrete and flexible, but still lightweight, WS-Security data model shown in Figure 4. The data model consists of two types of information: the WS-Security objects and the transition sequences. A WS-Security object includes all of the necessary information for the WS-Security processing done by the WSS processor. It is constructed in parallel as the WSS preprocessor matches the byte sequence with the data in the automaton. The logical structure of the WS-Security object is similar to the XML structure in the SOAP message secured by WS-Security. A transition sequence is a list of transitions that are traversed while matching the byte sequence with the data in an automaton. The transition sequence is used as the data representation of the input message instead of the byte sequence when the WSS processor secures the message based on the WS-Security data model.

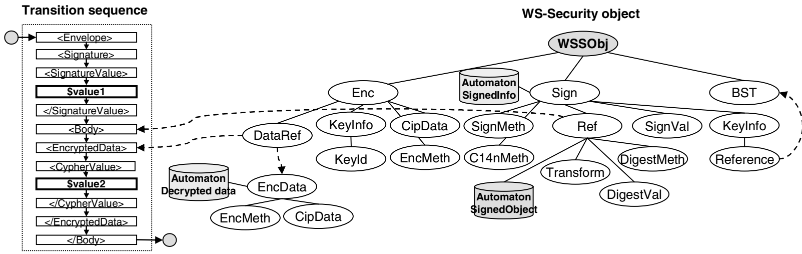

Fig. 4. The data model for WS-Security processing

2.3 WS-Security Processing Flow in the WSS Processor

In this section, we describe how the WSS processor applies WS-Security to the input SOAP message based on the WS-Security data model shown in the previous section. We use the following scenarios to simplify our explanations:

(1) For the inbound message, the WSS processor decrypts the contents of the SOAP body element first and then verifies the signature of the SOAP body element.
(2) For the outbound message, the WSS processor signs the SOAP body element first and then it encrypts the contents of the SOAP body element.

2.3.1 WS-Security Processing Flow for the Inbound Message

For the inbound message, the WSS processor invokes a decryption processor to decrypt the contents of the SOAP body element based on the WS-Security data model. The encryption processor extracts the cipher value including the octets of the encrypted data from the WSS object and invokes an encryption engine to decrypt the cipher value and to get the byte sequence of the real content of the SOAP body element. Then the encryption processor invokes its matching engine to match the decrypted byte sequence with those stored in the automaton and update the transition sequence for the subsequent WS-Security processing. The matching engine also dynamically updates the automaton if necessary. Figure 5 shows the processing flow of decryption for the SOAP body element. In Figure 5, the <*EncryptedData*> element is replaced with the actual content of the SOAP body element (the <*getQuote*> *element*).

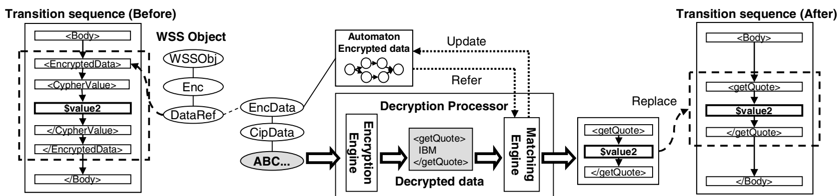

Fig. 5. Processing flow of decryption

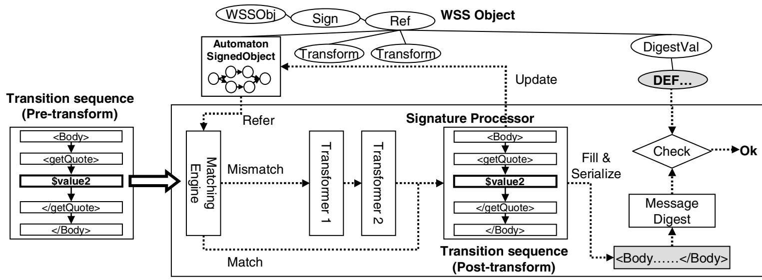

Fig. 6. Processing flow of digest value verification

After the decryption process is completed, the WSS processor invokes a signature processor to verify the signature of the SOAP body element based on the WS-Security data model. The signature verification process includes two steps: digest value verification and signature value verification. Figure 6 shows the processing flow for digest value verification. When the signature processor verifies the digest value, it first invokes the matching engine to match the input transition sequence with the ones stored in the automaton for the signed object, which means the SOAP body element in this case. The internal data structure in the automaton described here is slightly different from the one described in Section 3.1. This is because we avoid the same byte matching twice and reuse the input transition sequence to improve its performance. The state in the automaton for the signed object doesn't store the parser context since we don't use a parser in this case. The transition stores a transition T_i in the input transition sequence and another transition T_i' corresponding to T_i. T_i' is used to construct a post-transform transition sequence.

If there is a mismatch, the signature processor invokes the transformers for the corresponding transformation algorithm extracted from the WSS object, constructs the post-transform transition sequence, and dynamically updates the automaton. If the input matches, the signature processor can skip invocation of the transformers and automaton update because we can get the same post-transform transition sequence as would be constructed by the transformers. Figure 7 shows an example of transformations. In Figure 7, we assume that *transformer 1* handles XPath filter2 [5] and *transformer 2* handles the exclusive XML canonicalization [6].

When the post-transform transition sequences are constructed, the signature processor fills in the values extracted from the WSS object into the variable part in the transition sequence and serializes it to get a byte sequence. Then the signature processor invokes a message digest to calculate a digest of the byte sequence and verifies the digest with the digest value extracted from the WSS object.

The processing flow of signature value verification is basically the same as the flow of digest value verification shown in Figure 6. Therefore, we omit its details, though there are some differences in the figure: (1) the input transition sequence includes a <*SignedInfo*> element, (2) the automaton is for the <*SignedInfo*> element, (3) the transformers are changed to a canonicalizer, and (4) the message digest is changed to a signature engine.

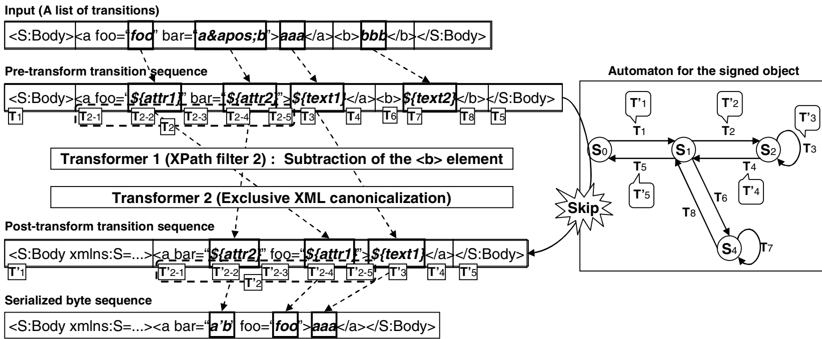

Fig. 7. Transformation example

2.3.2 WS-Security Processing Flow for the Outbound Message

For the outbound message, the WSS processor invokes a signature processor first to sign the SOAP body element based on the WS-Security data model. The signing process is a two-step process: digest value calculation and signature value calculation. The processing flow of digest value calculation is basically the same as the flow of digest value verification described in the previous section and details are not repeated. The only difference is that the signature processor for the outbound message stores the digest value in the WSS object after it invokes a message digester to calculate the digest value of the SOAP body element. The processing flow of the signature value calculation is also basically the same as the flow for signature value verification as described in the previous section, so we again skip the details. The only difference is that the signature processor for the outbound message stores the signature value in the WSS object after it invokes the signature engine to calculate the signature value of the *<SignedInfo>* element.

After signing, the WSS processor invokes an encryption processor to encrypt the content of the SOAP body element based on the WS-Security data model. The encryption processor extracts the original byte sequence of the content of the SOAP body element from the WSS object and invokes an encryption engine to encrypt the byte sequence. The WSS processor stores the encrypted cipher value in the WSS object, wraps the encrypted octet with the *<EncryptedData>* element, and updates the transition sequence for the subsequent WS-Security processing. Figure 8 shows the

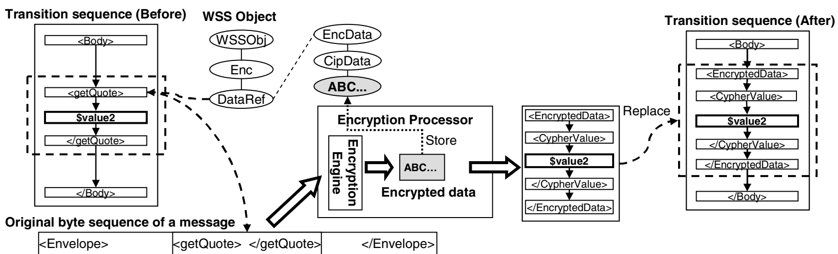

Fig. 8. Processing flow of encryption

processing flow of encryption of the SOAP body element. In Figure 8, the content of the SOAP body element has been replaced with the <*EncryptedData*> element.

3 Related Work

General Web cache and our processor share some things in common. For instance, they both store multiple received messages for later reuse. Usually Web cache stores the entire request URL and query string as cache key, which is highly inefficient in terms of memory usage. Only one-byte difference between messages tends to create separate cache entries, each representing the entire message. On the other hand, our processor divides a message into multiple pieces and common pieces are stored only once (merging commonalities between messages and collapsing repeating structure in one message), leading to an efficient data structure.

Some differential techniques, especially byte matching, have been proposed to avoid extensive XML processing in recent years. [8] assumes that a receiver knows in advance the structure of the SOAP messages (including the number of white spaces) to be exchanged. The receiver has to divide the message into two parts based on the XML syntax: fixed parts corresponding to element tags and variable parts corresponding to text nodes and attribute values, and hold them as a template before it processes the message. However this means that the entire message processing fails whenever any part mismatches the template, because the processor can't dynamically update the template. In addition, the processor can't hold more than one template at the same time.

In [9], an XML message was divided into fixed-length pieces. The pieces (P_1, …, P_n) are held as a template. The template also stores the parser context at each boundary of the portions so that the processor can resume a partial parsing by using the parser context between P_{i-1} and P_i when the input byte sequence doesn't match with P_i. The processor can terminate the partial parsing and restart byte matching if the parser context becomes the same as the one already stored in the template. However it is difficult to reduce the number of pieces even when the same data structure appears repeatedly in the payload of the message. In addition, the processor holds only one template, since the mismatched portions are replaced with the ones generated during partial parsing.

Similar to [8], [10] divides a message into fixed parts and variable parts. These parts are held in an automaton. The advantage of this approach is that the processor can hold

Table 1. The differences in the related work, where (a) is dynamic template generation, (b) is partial parsing, (c) is holding multiple templates, (d) is distinguishing between fixed parts and variable parts in the data model, and (e) is data model optimization

	(a)	(b)	(c)	(d)	(e)
Web cache	yes	no	yes	no	no
[8]	no	no	no	yes	yes
[9]	yes	yes	no	no	no
[10]	yes	no	yes	yes	No
This paper	yes	yes	yes	yes	yes

multiple message templates in the automaton. However since the parser context is not stored, it is impossible to optimize the data structure in the automaton.

This paper addresses many of the problems in these systems. Our processor divides the message into fixed parts and variable parts according to the XML syntax in the input byte sequence and holds them in an automaton. The automaton also stores parser contexts. Therefore, the processor can resume a partial parsing and dynamically update the data in the automaton without invoking another processor as in [10]. In addition, the data model in an automaton can be optimized even when the same structure appears repeatedly in the byte sequence. Table 1 shows a summary of the differences in the related work, where Web cache can be regarded as holding multiple templates, each containing one large fixed part which representing the entire message..

4 Performance Study

This section describes a performance study that we conducted to evaluate the memory usage and performance of our differential technique. Section 4.1 presents the experiment in terms of memory usage and Section 4.2 shows the experiment in terms of performance.

4.1 Experiment in Terms of Memory Usage

We conducted an experiment to examine how the memory required for our data model differs from the memory needed for the data model in [10] on the service provider. We ran all of the tests on a ThinkPad[1] T42 (Intel Pentium[2] M 745 1.8 GHz, 1.5 GB RAM, Windows[3] XP Professional Edition). Five different services were used

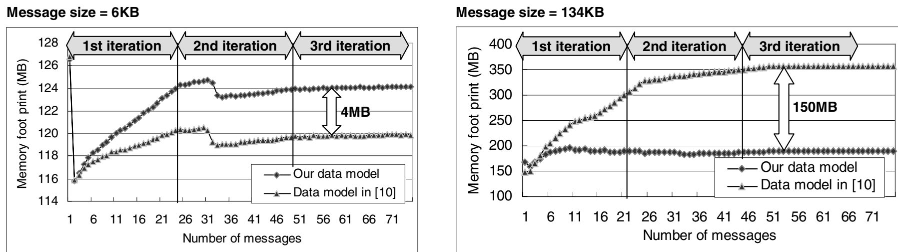

Fig. 9. Memory usage comparison between our data model and the data model in [10]

in the experiment. Then we prepared five different messages per service (for a total of 25 different messages). Each message included the same XML structure that appeared repeatedly in the payload of the SOAP body element. We sent them three times to the services with a Web services client. Therefore, the client sent a total of 75 messages

[1] ThinkPad is a trademark of Lenovo in the United States, other countries, or both.

[2] Intel and Pentium are trademarks of Intel Corporation in the United States, other countries, or both.

[3] Windows is a trademark of Microsoft Corporation in the United States, other countries, or both.

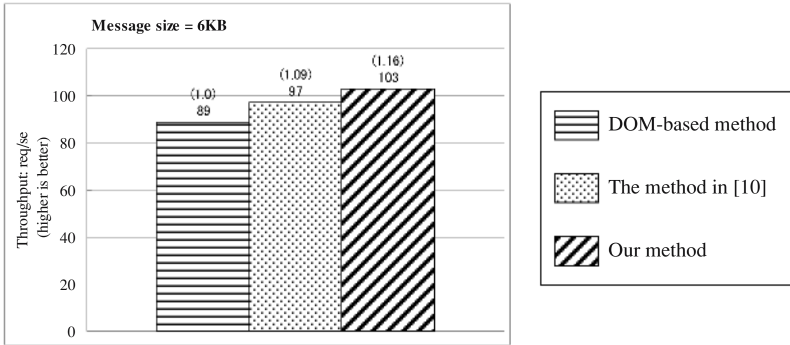

Fig. 10. Performance comparison between our method and the method in [10]

to the services. Figure 9 shows the experimental results. Figure 9 indicates that our data model can save about 150 MB of memory when sending the 13 KB of message though it required slightly more memory when sending the 6KB of message.

4.2 Experiments in Performance Number

We conducted an experiment to compare the performance of our method and the performance of the method in [10] on the service provider. We ran all of the tests with on an IBM xSeries[4] 365 (Intel Xeon[5] MP 3.0 GHz, 4-way, 4 MB L3 Cache, 8 GB RAM, with HyperThreading disabled, on Windows Server 2003 Enterprise Edition). Figure 10 shows the experimental results when asymmetric signature and encryption algorithms are used and the 6KB of message is received on the service provider. In the graph, the x-axis is the kind of implementation (DOM-based method, the method

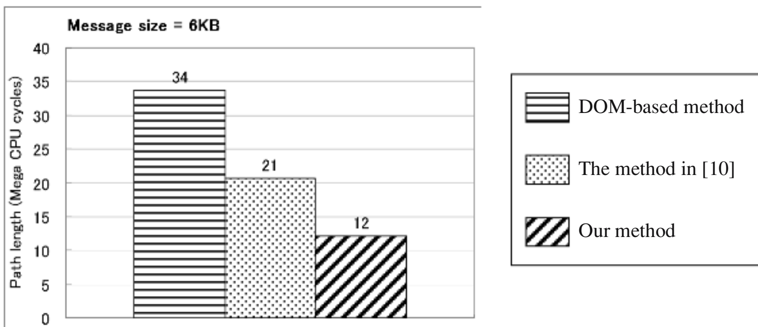

Fig. 11. Path length comparison between our method and the method in [10]

in [10], and our method) and the y-axis is throughput (requests/sec). Since the XML processing constitutes a second greater portion of the total WS-Security processing in

[4] IBM and xSeries are trademarks of International Business Machines Corporation in the United States, other countries, or both.

[5] Xeon is a trademark of Intel Corporation in the United States, other countries, or both.

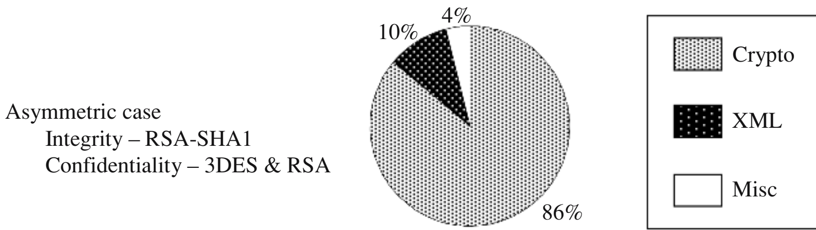

Fig. 12. Analysis of performance contribution in our method

the case using asymmetric algorithms, our method makes a contribution to performance improvement. Figure 10 also indicates that performance number of our method is faster than the method in [10], even though our method is more practical and more flexible.

We also conducted an experiment to examine the path lengths required for our processing method compared to the path lengths needed for the processing method in [10]. We ran all of the tests on an IBM xSeries 365 (Intel Xeon MP 3.0 GHz, 4-way, 4 MB L3 Cache, 8 GB RAM, with HyperThreading disabled, on Windows Server 2003 Enterprise Edition). In the experiment, we first used an internally developed tool to get a call graph. Then we analyzed the path lengths calculated from the call graph. Figure 11 shows the experimental results. Figure 11 indicates that our method can shorten path lengths required for the XML processing, compared with the method in [10].

Finally we conducted an experiment to analyze the performance contribution in our method. Figure 12 shows the experimental results. Figure 12 represents that our method can reduce the percentage of the XML processing in the while WS-Security processing compared with the percentage of the XML processing shown in the asymmetric case in Figure 1.

5 Concluding Remarks

In this paper, we have presented a new architecture for WS-Security processing with differential parsing to improve the XML performance in the WS-Security processing. In our architecture, the WSS preprocessor matches the byte sequence of an input SOAP message with the data that were previously processed and stored in an automaton. If the byte sequence completely matches with the data in the automaton, it means that we can skip all unnecessary XML processing in the WS-Security processing. On the other hand, if there is any mismatch, the processor can partially parse only the unmatched parts of the byte sequence of the message because the parser contexts are also stored in the automaton. While parsing the parts, it divides into the fixed parts and the variable parts according to the XML syntax in the message, and updates the automaton with the divided parts. We also proposed a more complete and more flexible data model for WS-Security processing so that we could support a wider variety of practical scenarios that [10] does not cover.

The performance study in terms of memory usage showed that our architecture requires less memory than needed for the architecture described in [10]. The

performance study in terms of performance number also showed that there is not a large difference between the performance of our architecture and the architecture described in [10] when the asymmetric signature and encryption algorithms are used, though the internal data structure in the automaton is more flexible and the data model for WS-Security processing is more complete.

References

1. Simple Object Access Protocol (SOAP) Version 1.2, http://www.w3.org/TR/soap12/
2. Web Services Description Language (WSDL) 1.1, http://www.w3.org/TR/wsdl
3. Web Services Security: SOAP Message Security 1.1, http://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf
4. XML Security Suite, http://www.alphaworks.ibm.com/tech/xmlsecuritysuite
5. XML-Signature XPath Filter 2.0, http://www.w3.org/TR/xmldsig-filter2/
6. Exclusive XML Canonicalization Version 1.0, http://www.w3.org/TR/xml-exc-c14n/
7. Hongbin Liu, Shrideep Pallickara, and Geoffrey Fox, Performance of Web Services Security, Technical Report, 2004, http://grids.ucs.indiana.edu/ptliupages/publications/WSSPerf.pdf
8. Yoichi Takeuchi, Takashi Okamoto, Kazutoshi Yokoyama, and Shigeyuki Matsuda, "A Differential-analysis Approach for Improving SOAP Processing Performance," The 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service (EEE'05), pp. 472-479, 2005
9. Nayef Abu-Ghazaleh and Michael J. Lewis, "Differential Deserialization for Optimized SOAP Performance," ACM/IEEE SC 2005 Conference (SC'05), pp. 21-31, 2005
10. Satoshi Makino, Michiaki Tatsubori, Kent Tamura, and Yuichi Nakamura, "Improving WS-Security Performance with a Template-Based Approach," IEEE International Conference on Web Services (ICWS'05), pp. 581-588, 2005
11. Toshiro Takase, Hisashi MIYASHITA, Toyotaro Suzumura, and Michiaki Tatsubori, "An adaptive, fast, and safe XML parser based on byte sequences memorization," The 14th international conference on World Wide Web (WWW 2005), pp. 692-701, 2005