Security Based Heuristic SAX for XML Parsing

Wei Wang
Department of Automation
Tsinghua University, China
Beijing, China

Abstract - *XML based services integrate information resources running on different platforms or technologies to enhance the service efficiency. Thus, the volume of XML traffic on networks is increasing rapidly, and it demands for efficient XML processing algorithms to support high performance services, especially in XML security applications. This paper proposes a memory efficient XML parsing algorithm leveraging on security based heuristic and SAX (Simple API for XML) algorithm with schema validation, called Security Based Heuristic SAX (SBH-SAX), which exploits the characteristics of XML security applications and the connections between different XML processing functions. Experimental results demonstrated the overall performance improvement of XML parsing and XML security processing using SBH-SAX. Comparing to SAX based processing and DOM based processing, system with SBH-SAX requires the least memory usage in most security processing cases, and also performs twice faster in processing.*

Keywords: SBH-SAX, XML Parsing, XML Security

1. Introduction

XML based services, like Web services and AJAX based applications, are quickly gaining popularity, which integrate information resources running on different platforms or technologies, and enhance the efficiency of services. Thus, the volume of XML application traffic on networks is increasing rapidly and predicted to take about 45% of overall network traffic in 2008 [1].

There are two key challenges in the deployment of XML based services: security and performance.

● Security

Although XML can provide seamless connections between applications, it gives attackers some chances to invade the security of hosts via network at the same time. XML related security threats are mainly emerging as data compromise, XML based DoS (Denial of Service) and content-based attack. These security problems motivate the development of XML security processing functions, XML encryption and XML signature, which provide element-level protection.

● Performance

As XML traffic is growing, the system resources consumed by XML processing are over loading the system and decreasing the performance of XML based services.

XML processing functions in XML enabled network can be integrated in an XML dedicated appliance, such as XML firewall, onto which the XML processing in application servers is offloaded. Besides convenience for management, it can also reduce the overlap across services in the XML based applications and free the server resources to better handle other critical functions. The XML appliance demands for advanced XML processing algorithms to support high performance services, especially in XML security applications.

This paper presents a memory efficient XML parsing algorithm leveraging on security based heuristic and SAX (Simple API for XML) algorithm with schema validation, called Security Based Heuristic SAX (SBH-SAX), which exploits the characteristics of XML security applications and the connections between different XML processing functions.

The remainder of the paper is organized as follows. Section 2 analyzes the previous work for XML parsing with schema validation; Section 3 presents system characteristics of XML security system; Section 4 describes the proposed algorithm Security Based Heuristic SAX (SBH-SAX); Section 5 illustrates experimental analysis of security system with SBH-SAX compared with other popular algorithms; as a summary, Section 6 states the conclusion.

2. Previous Work

XML parsing with schema validation is one of the basic XML processing functions. It parses the information contained in an XML message and checks whether it can be considered as well-formed and valid. The results of XML parsing should provide enough support for following XML processing, like XML query, XML security. The algorithm has two parts: parsing and validation.

● Parsing

The information contained in an XML message will be picked up and represented for following processing. The welled-formedness of the XML message can be checked at the same time with a tag stack.

One of the most popular parsing algorithms is Tree Parsing [3]. It parses the XML message into a tree structure, in which character strings like element name, attribute value, are represented by nodes. Access to the information of XML message is essentially the same as a tree query. This algorithm takes much time and memory to create and store the tree structure.

A memory efficient parsing algorithm is Tokenized XML Format [4]. This algorithm cuts XML message into several pieces of character strings with corresponding information stored in memory, like "element content", "attribute name", etc. It is optimized by using a Code Table, which is efficient for memory usage and XML query. But this algorithm is difficult to use for security processing since it alters the XML message.

Another parsing algorithm is Non-Extractive Parsing [5]. This algorithm records a two-tuple integer array for each character string in XML message: one tuple for offset of the string, the other for length of the string. It is efficient for the memory usage and XML query, but does not well support XML security processing. Because the offsets in the parsing results are not enough to manipulate XML message for insert and replacement.

All of the three parsing algorithms are well designed but not aiming at the specific XML security processing, i.e. XML encryption and XML signature. It parses the whole XML message, some parts of which may not even be used in the application.

● Validation

XML schema was approved as a W3C Recommendation on May 2nd, 2001 [2]. The validity of an XML message should be checked according to associated schema. An XML schema should be preprocessed into a data structure, which can be a tree structure [6] or a graph based on finite state automata [7]. The system will take the XML message as input and go through the schema tree or graph to check its validity.

SAX and DOM are two most popular programming interfaces for XML parsing. They are constructed using the algorithms described above.

SAX (Simple API for XML) [8] is an event-based parser, which raises events when it encounters the start or end tags. It processes XML message like a pipeline at a fast speed, but with no structure left in memory, so the following processing needs to re-parse the XML message into tree structure for XML manipulation.

DOM (Document Object Model) [9] is a tree-model parser, which converts an XML message to a tree structure in memory. With this parsing results DOM provides a convenient way for XML query and manipulation, but it consumes more time and memory at the same time.

SBH-SAX, proposed in this paper, is based on SAX. SBH-SAX keeps the schema validation part of SAX and optimizes the parsing part of it by introducing security based heuristic information. The event-based algorithm is designed to leverage the heuristic information of the following XML security processing and parses the XML message without costing redundant processing time and memory space. The parsing approach has some similarities with the Non-Extractive Parsing algorithm, for using the offset of a string, but has a different way to generate and express the concerned text, which will be signed or encrypted in the processing.

3. Characteristics of XML Security System

Generally speaking, an XML security system consists of XML parsing with schema validation, XML signature, XML encryption. They are integrated to form a XML processing pipeline.

Figure 1 shows the work flow of XML signature or encryption. We note that two steps of the whole process require support from the parser, as shaded blocks in the figure, namely the "Get the plain text" step and "Operate XML message" step. Information required in these steps are concerned text to be signed, concerned text to be encrypted, insert position of signature element, and replacement position of encryption element.

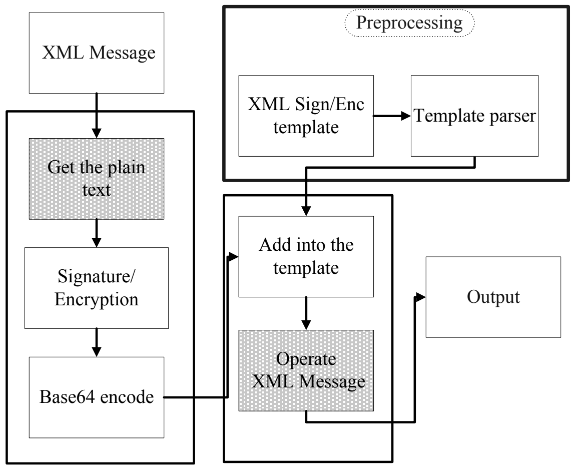
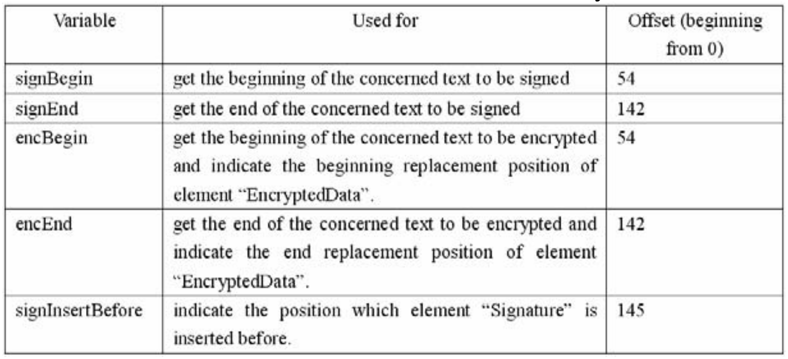


Figure 1: Work flow of signature or encryption.

DOM parser has considerable heavy workload for security processing, as mentioned in section 2. Obviously, a security-focused parser does not have to get all information as DOM does. Base on the observation, we proposed our parsing algorithm – SBH-SAX.

Table 1: Offset values stored in memory



Variable	Used for	Offset (beginning from 0)
signBegin	get the beginning of the concerned text to be signed	54
signEnd	get the end of the concerned text to be signed	142
encBegin	get the beginning of the concerned text to be encrypted and indicate the beginning replacement position of element "EncryptedData".	54
encEnd	get the end of the concerned text to be encrypted and indicate the end replacement position of element "EncryptedData".	142
signInsertBefore	indicate the position which element "Signature" is inserted before.	145

4. Security Based Heuristic SAX (SBH-SAX)

SBH-SAX is designed to enhance the overall performance of XML security system including XML parsing, XML signature and encryption. The basic idea of SBH-SAX is that those and only those texts that are necessary for the following security processing should be parsed and maintained according to the specific security policy of an application. Two improvements are introduced into SAX: heuristic automata and offset. The paths of concerned texts, which are indicated by XPath [10] language, will be preprocessed into heuristic automata based on the security policy. By running the heuristic automata we can get the offsets of concerned texts in order to manipulate XML message in security processing.

4.1 Algorithm Explained by an Example

To illustrate the SBH-SAX algorithm proposed in this paper, an example of XML message is used for illustration. Figure 2 shows the XML message and its tree structure.

```
<?xml version="1.0"?>
<class>
      <number>1</number>
      <roster>
            <student>
                  <name>wangwei</name>
                  <sid>1</sid>
            </student>
      </roster>
</class>
```

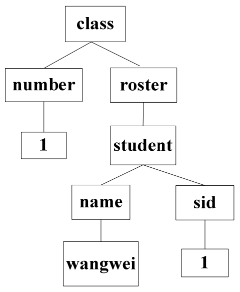


Figure 2: An XML message and its tree structure.

The security policy specifies that the concerned text to be signed and the concerned text to be encrypted are same, which are "/class/roster" in XPath. It means that element "roster" of root element "class", including its child elements and contents, should be all signed first and then encrypted. We assume the signature is in the form of enveloped, which means that the signature element should be added into the XML message as last child of the root element.

Both systems, with either SAX or DOM, have the same process to conduct this security processing. They first parse the original XML message into a tree structure shown in Figure 2 and then search the tree according to the path "/class/roster" to get the sub-tree of concerned text. Then the sub-tree is changed into text in XML format, which will be signed and encrypted to get the cipher texts. After the element "Signature" has been added to the tree and the element "EncryptedData" has replaced the element "roster", the systems outputs the tree in XML format.

In system with SBH-SAX, the path "/class/roster" is preprocessed into an automaton. In fact, five automata in this form are merged into one, named heuristic automata, which can generate five offsets shown in Table 1.

Heuristic automata embedded in SAX processes the XML message in pipeline and store values of necessary offsets in memory for following security processing, where some manipulations on character strings generate the output message.

4.2 Heuristic Information Processing

Heuristic information processing compiles security policy described in XPath language into heuristic automata, in order to find the concerned text to be signed or encrypted. This preprocessing work is designed based on the observation that most of the security policies are invariant in run-time security processing of a specific XML based service, and can be established when the connection is being set up. Therefore, the heuristic automata can be generated before processing XML messages of this connection.

Finding the concerned text can be considered as regular expression pattern matching in XML. Tree automata theory can be applied to handle it [11, 12]. However, we observe that XML security processing at network gateway position does not employee all features of the XPath language, and most of the security policies aim at only the structural part of XML message, namely the element level. Therefore, SBH-SAX only needs to support simple XPath in form of "/aaa/bbb", which can be preprocessed more efficiently in speed and memory usage. In addition the heuristic automata should output the offsets of some end positions of elements. We use "/aaa/\bbb" to denote the path of end of element "bbb" and add character string "</bbb>" to the heuristic automata in order to get the corresponding offset.

Actually there are five heuristic automata to generate five required offsets. We can merge them into one, which is show in Figure 3 with signature beginning "/class/roster", signature end "/class/\roster", encryption beginning "/class/road", encryption end "/class/\road" and signature insert "/\class". The heuristic automata are running to filter the XML message after checking its welled-formedness.

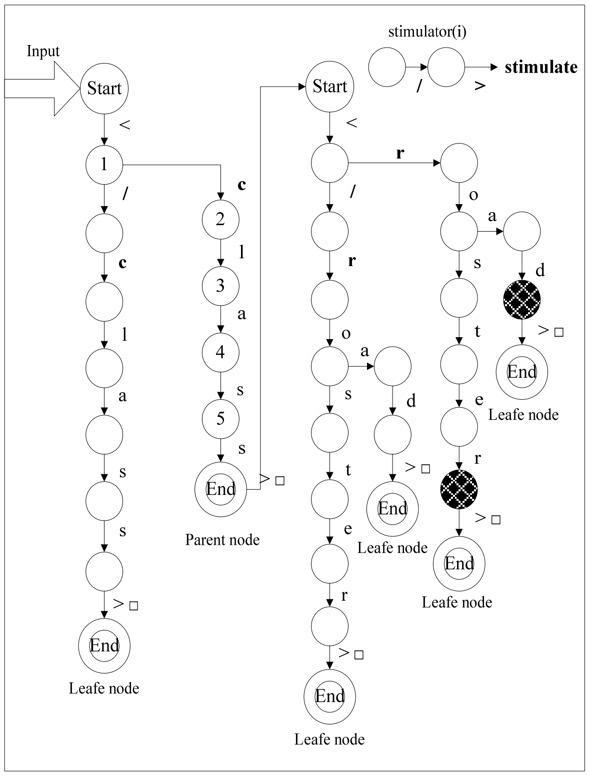


Figure 3: Merged heuristic automata

Using the example in section 4.1, which is shown in Figure 2, we describe how the heuristic automata work. XML message is input to the automata as a stream beginning from the state "start". There is a goto function *g*, which maps a pair consisting of a state and an input symbol into a state or the message *fail*. For example, *g*(start, <) = 1, *g*(1, c) = 2, *g*(1, x) = *fail*, *g*(state "end" in parent node, >) = state "start" in child node. The failure function *f* is going to the state "start" of current level. But if there is no node to be matched in this level, which is to say all states "end" have been accessed, then *f*(state "end") = state "start" in parent node. The failure function can be very simple because the welled-formedness check is performed first using SAX. If the XML message is well-formed, the automata can work well; otherwise, the parser will have reported an error and stop processing. When the "end" state of the leaf node is reached, a match is found and the *output* function is used to record the offset

The following algorithm shown in Figure 4 summarizes the behavior of heuristic automata.

Algorithm. heuristic automata

Input. XML Message $x = a_1a_2...a_n$, where each a_i is a symbol in XML message. Heuristic automata *H* with goto function *g*, failure function *f* and output function *output*, as described above. We use 0 to represent state "start".

Output. Offset which the XPath language "/xxx/xxx/xxx" indicates.

Method.

```
begin
    state ← 0
    for i ← 1 until n do
        begin
            while g(state, a_i) = fail do state ← f(state)
            state ← g(state, a_i)
            if state = "end" of leafe node then
            begin
                output(state)
                state ← f(state)
            end
        end
end
```

Figure 4: Algorithm of heuristic automata.

There are three exceptions in XML messages we must handle in the heuristic automata, which are not shown in Figure 4 for simplicity of expression. They are as follows:

a. <xxx yyy="<zzz>">. When <zzz> is equal to some string to be matched, the automata will go to a wrong state. The solution is to stop the automata when a quotation mark appears, and to start it when another quotation mark comes in.

b. <xxx sss="yyy">. When there is space after the element name "xxx", which means it is not followed by ">", the automata will not go to the state "end". The solution is that we can add character space " " to the automata, which is shown as "□" in Figure 3.

c. <xxx sss="yyy"/>. When the element ends without “</xxx>”, the automata cannot find the end of the element. Here a stimulator is created, shown at the top right corner of Figure 3. When a shaded state in the figure is reached, its corresponding stimulator is activated. The stimulator then reads the XML stream as well. When it ends with “stimulate”, the end of the element is found. The automata will output an offset and close the corresponding branch by assigning 1 to the access sign.

4.3 Information Expression with Offset

When the concerned texts are found by heuristic automata, offsets are used to indicate their positions in XML message instead of parsing them out. The message and the offsets are maintained to support the following security processing, which is faster and consumes less memory for information storage than parsing all messages into DOM tree.

An example is shown in Table 1, section 4.1, in which all five necessary items are listed. Each item has a corresponding heuristic automaton. The five heuristic automata are merged into one as described in section 4.2. With these offsets, the XML message for security processing can be effectively handled through the manipulation on the character strings.

Offset is used instead of the complicated DOM tree. Although not as flexible as DOM tree, it is sufficient for text locating and efficient for security processing.

5. Implementation and Experiment

Some experiments were carried out in such a SBH-SAX implementation system and compared with other XML security systems using different parsers, including SAX and DOM. The impacts on processing speed are evaluated with experimental data on three factors: the size of concerned text, the proportion of concerned text to whole XML message, and the level of XPath language. Memory usages of these three systems are also analyzed.

The hardware platform used in our experiments is IXP425 with embedded 533 MHz XScale core, and 64 MB memory [13]. The software platform is based on snapgear-3.3.0. We use Xerces-c-2.6.0 from Apache XML Project to implement SAX and DOM parsers and use NPE (Network Processing Engine) B in IXP425 to accelerate signature and encryption.

5.1 Conditions of the Experiments

The experiments are conducted under the following set up and configuration.

Firstly, the XML messages are generated manually according to different experimental targets. The performances are measured by overall processing time per message, including XML parsing, signature and encryption. The values are calculated by the average of results from three test trails.

Secondly, the concerned texts are the text to be signed and the text to be encrypted. For simplicity of construction, the signature and encryption are applied to the same text in the experiments. Then we can make unified adjustment to size of the concerned text.

Finally the signature algorithm used in the experiments is HMAC-SHA1. It is actually a MAC (Message Authentication Codes) algorithm, which cannot provide non-repudiatability. The encryption algorithm is 3DES. The keys are preshared, which means that keys are allocated and known by both sender and receiver in advance.

5.2 Impact of Size of the Concerned Text

The performances of systems with SBH-SAX, SAX and DOM are shown in Figure 5. The proportion of concerned text to whole XML message is fixed as 1:1, which actually means that nearly the whole XML message should be signed and encrypted. The XPath has two levels as “/aaa/bbb”. The size of the concerned text varies from 1KB to 100KB, which is denoted by the horizontal axis. The processing time in milliseconds per XML message is denoted by the vertical axis.

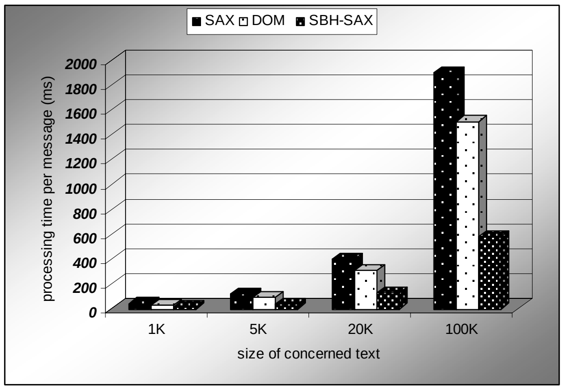


Figure 5: Performance of different sizes.

From the figure we can see that system with SBH-SAX has the least processing time, in other words, the best performance. Compared to the system with DOM, the next to the best performer, system with SBH-SAX has reduced the processing time by more than 50%, which means that the processing speed is twice as fast. As the size of XML message increases, the time saved by SBH-SAX is more significant. It can be seen from the time proportion of SBH-SAX to DOM, which decreases from 49.6% to 38.5%. The reason is that there is more time used in format transformation in DOM parser as the size of concerned text increases, and SBH-SAX parser has no such operation.

5.3 Impact of Proportion of Concerned Text to Whole XML Message

The performances of systems with SBH-SAX, SAX and DOM are shown in Figure 6. The size of concerned text is fixed as 1KB. The XPath has two levels as "/aaa/bbb". The proportion varies from 100% to 12.5%, which is denoted by the horizontal axis.

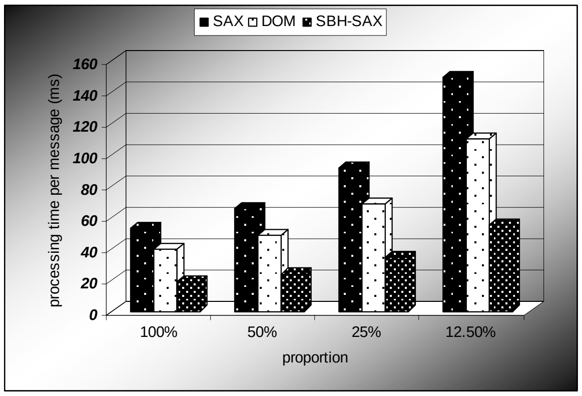


Figure 6: Performance of different proportions.

There is processing time wasted in systems with SAX and DOM to parse the unconcerned text into DOM tree. SBH-SAX is designed to avoid this unnecessary information processing. The time saved by SBH-SAX increases from 20.3ms to 54.4ms per message along with reduced proportion, representing the benefit brought by the optimization with security based heuristic.

5.4 Impact of Level of XPath Language

The performances of systems with SBH-SAX, SAX and DOM are shown in Figure 7. The size of concerned text is fixed as 1KB. The proportion is fixed as 50%. The level of XPath varies from 2 to 5, which is denoted by the horizontal axis. It means that the form of XPath varies from "/aaa/bbb" to "/aaa/bbb/ccc/ddd/eee".

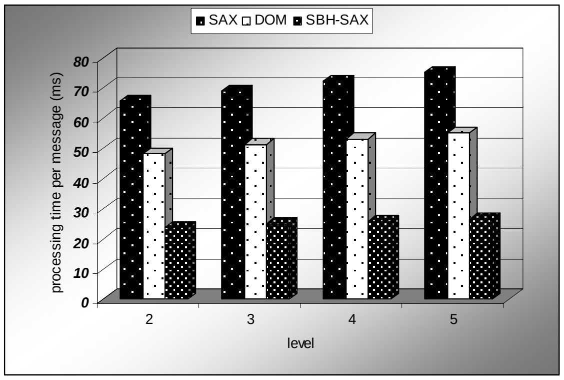


Figure 7: Performance of different levels.

As the level of XPath increases, the processing time does not increase significantly. The time saved by SBH-SAX is nearly constant, because the query time of XML message only takes a small ratio of the whole XML processing and the impacts of level increase on these three systems are nearly the same. It means a more complicated search of the DOM tree in systems with SAX and DOM, while at the same time there are more states to be matched in heuristic automata of the system with SBH-SAX.

5.5 Memory Usage

Multiplying factor is a metric to measure the memory usage of XML parsing algorithm, which is the size of the memory usage divided by the size of the XML message itself. The multiplying factor of DOM varies from 5 to 10 for different XML messages [14]. For SBH-SAX, the multiplying factor varies according to the complexity of the heuristic automata and the size of the XML message. The offsets can be neglected, which are no more than 20 Bytes.

The above discussion is about one XML message and one security policy. It is hard to evaluate which of the two, DOM and SBH-SAX, is better in memory usage. But in most of the cases for security processing, which have lots of XML message based on the same security policy, or not too many policies, SBH-SAX is proved to be more memory efficient than DOM for two main reasons.

Firstly, the heuristic automata can be shared by different XML messages based on the same security policy. So if we have 5KB heuristic automata which can be shared by 100 XML messages in 1KB being processed. The multiplying factor is (100+5)/100 = 1.05, which is much smaller than DOM (above 5). The essential reason is that the memory usage of data structure in SBH-SAX is unrelated to the size of XML message. It is decided by the security policy, which is relatively invariant in a specific service.

Secondly, most of the XPaths describing security policy are not too long, so the corresponding heuristic automata will not be too big in size. We observe that in most of the security processing cases the heuristic automata are less than 5KB and the XML message size varies significantly, from 1KB to more than 1MB (for some XML messages carrying data). This reduces the multiplying factor of SBH-SAX too.

So in most cases with a relatively invariant security policy, simple XPath and bigger XML message, the memory usage of SBH-SAX is much less than DOM, and the multiplying factor is nearly 1.

5.6 Discussions

We conduct the experiments of processing time in three ways, different sizes of concerned texts, different proportions, and different XPath levels. The results of these dedicated tests validate the essential features of SBH-SAX algorithm that enables the better performance of the system with it.

1. SBH-SAX is optimized by reduction of the unnecessary text processing. This feature is achieved through heuristic automata. As the proportion of unnecessary text increases, the reduction of processing time is becoming relatively more.

2. SBH-SAX is also optimized by simplification of the concerned text representation. This feature is achieved through offset method. As the size of concerned data increases, the simplification method shows better and better performance in both processing speed and memory usage.

Although SBH-SAX has faster processing speed, its memory usage is not always less than DOM. If the size of XML message is small or the security policy is complicated, SBH-SAX could require more memory than DOM. However in most cases, it requires less memory. Essentially it depends on the size of heuristic automata of SBH-SAX and the XML messages being processed.

6 Conclusion and Future Work

Previous XML security systems are not efficient for security processing in both speed and memory. The reason is that they process unconcerned texts and store them in memory in a complicated way.

SBH-SAX is an XML parser based on SAX, processing XML message in pipeline. It is better used in a specific security system, which has security policy set beforehand. The heuristic security policy is utilized to form heuristic automata as a searching engine. Only concerned texts are processed and stored in memory by offsets. Experimental results show that, system with SBH-SAX requires the least memory usage in most security processing cases, and also has above twice processing speed.

Future work can be conducted to optimize the SBH-SAX algorithm by compressing the heuristic automata or utilizing more heuristic information. The algorithm can be also applied to other XML applications, which have a requirement of manipulation on XML messages in a way set beforehand.

Acknowledgement

The author would like to thank Mr. Quan Huang for his help in the IXP425 based implementation. Thanks also due to colleagues at the Network Security Lab of Research Institute of Information Technology, Tsinghua University, for their generous suggestions and encouragement.

References

1. J. Bloomberg and R. Schmelzer, "A Guide to Securing XML and Web Services," White Paper of Zapthink LLC, Jan 2004.
2. W3C XML Schema, http://www.w3.org/XML/Schema.
3. Y. Papakonstantinou and V. Vianu, "Incremental Validation of XML Documents," In Proc. of 9th International Conference on Database Theory (ICDT), 2003.
4. B. Nag, "Acceleration Techniques for XML Processors," In Proc. of XML 2004, 2004.
5. J. Zhang, "Non-Extractive Parsing for XML," http://www.xml.com.
6. M. Murata, D. Lee, and M. Mani, "Taxonomy of XML Schema Languages using Formal Language Theory," In Proceedings of 2001 Extreme Markup Languages Conference, 2001.
7. N. Wang, P.S. Housel, G. Zhang, and M. Franz, "An Efficient XML Schema Typing System," Technical Report of UC Irvine, 2003.
8. SAX Project, http://www.saxproject.org/.
9. W3C DOM, http://www.w3.org/DOM/.
10. W3C XPath, http://www.w3.org/TR/xpath.
11. M. Murata, "Transformation of Documents and Schemas by Patterns and Contextual Conditions," In Proc. of 3rd International Workshop on Principles of Document Processing (PODP), 1996, volume 1293 of LNCS, pages 153-169, Springer-Verlag, 1997.
12. A. Neumann and H. Seidl, "Locating Matches of Tree Patterns in Forests," In Proc. of 18th Conference on the Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 1998, volume 1530 of LNCS, pages 134-145, Springer-Verlag, 1998.
13. "Intel IXP425 Network Processor," Product Brief of Intel, http://www.intel.com/design/network/prodbrf/27905104.pdf
14. http://vtd-xml.sourceforge.net/technical/4.html