Server-Side Streaming Processing of WS-Security

Nils Gruschka, Meiko Jensen, Luigi Lo Iacono, and Norbert Luttenberger

Abstract—With SOAP-based Web Services leaving the stadium of being an explorative set of new technologies and entering the stage of mature and fundamental building blocks for service-driven business processes—and in some cases even for mission critical systems—, the demand for non-functional requirements including efficiency as well as security and dependability commonly increase rapidly. Although, Web Services are capable of coupling heterogeneous information systems in a flexible and cost-efficient way, the processing efficiency and robustness against certain attacks do not fulfill industry-strength requirements.

In this paper, a comprehensive stream-based WS-Security processing system is introduced, which enables a more efficient processing in service computing and increases the robustness against different types of Denial-of-Service (DoS) attacks. The introduced engine is capable of processing all standard-conforming applications of WS-Security in a streaming manner. It can handle e.g. any order, number and nesting degree of signature and encryption operations, closing the gap towards more efficient and dependable Web Services.

Index Terms—Web Services, SOAP, WS-Security, Streaming Processing, DoS Robustness, Efficient Processing

1 INTRODUCTION

Enterprises are faced with greatly changing requirements influencing the way businesses are created and operated. They have become more pervasive with a mobile workforce, outsourced data centres, different engagements with customers and distributed sites. Information and communication technology (ICT) is therefore becoming a more and more critical factor for business. ICT moves from a business supporter to a business enabler and has to be partly considered as a business process on its own.

In order to achieve the required agility of the enterprise and its ICT, the concept of Service-Oriented Architectures [1] is increasingly used. The most common technology for implementing SOA-based systems is the SOAP-based Web Services [2]. Some applications like Software-as-a-Service (SaaS) [3], [4] or Cloud Computing [5] are inconceivable without Web Services. There are a number of reasons for their high popularity. SOAP-based Web Services enable flexible software system integration, especially in heterogeneous environments, and is a driving technology for inter-organization business processes. Additionally, the large amount of increasingly mature specifications, the strong industry support and the large number of Web Service frameworks for nearly all programming languages have boosted its acceptance and usage.

Since SOAP is an XML-based protocol, it inherits a lot of the advantages of the text-based XML such as message extensibility, human readability and utilization of standard XML processing components. On the other hand of course, SOAP also inherits all of XML’s issues. The main problems—used by critics since the start of Web Services—are verbosity of transmitted messages and high resource requirements for processing [6]. These issues are further increased when using SOAP security [7] through the need of handling larger messages and performing cryptographic operations. These issues possess performance challenges which need to be addressed and solved to obtain the efficiency and scalability required by large (cross-domain) information systems. These problems are especially severe e.g. in mobile environments with limited computing resources and low data rate network connections [8], or for high-volume Web Service transactions comprising a large number of service invocations per second. Further, high resource consumption is not only an economic or convenience factor it also increases the vulnerability for resource exhaustion Denial-of-Service attacks.

To overcome the performance issues, streaming XML processing provides promising benefits in terms of memory consumption and processing time. The streaming approach is not new, but has not found a widespread adoption yet. Reasons therefore are manifold. The main issue surely is the missing random access to elements inside the XML document which makes programming difficult. Therefore, a current trend is using stream-based methods for “simple” message pre-processing steps (e.g. schema validation) and tree-based processing inside the application.

WS-Security processing is double-edged in this sense.
On one hand, high resource consumption and the ability to detect malicious messages makes security processing an ideal candidate for streaming methods. On the other hand, it requires rather complex operations on the SOAP message. Thus, to date there exists no comprehensive stream-based WS-Security engine.

This paper presents how a secured SOAP message as defined in WS-Security can be completely processed in streaming manner. It can handle e.g. any order, number and nesting degree of signature and encryption operations. Thus, the system presented provides the missing link to a fully streamed SOAP processing which allows to leverage the performance gains of streaming processing as well as to implement services with an increased robustness against Denial-of-Service (DoS) attacks.

2 MOTIVATION AND BACKGROUND

To set the scenes for this paper, some motivating foundations and related state-of-the-art are briefly introduced in the subsequent sections.

2.1 Efficient Processing

There exist two general processing models for XML documents: document-based processing and stream-based processing. In the first one, the complete XML document (e.g. a Web Service SOAP message) is read, parsed and transformed into an in-memory object tree representation of the document. All XML processing is performed using this object tree. The most widespread implementation of this model is the W3C Document Object Model (DOM) [9].

When applying the second model, the XML document is read and parsed step by step, divided into parts (e.g. single XML elements) and passed to the application. The application then operates on these XML parts. One can distinguish two approaches for implementing the communication between the parser and the application: a pull and a push approach. With the pull approach, the application requests the next XML part from the parser. A well known implementation of this approach is the Streaming API for XML (STAX) [10]. Using the push approach, the parser calls the application, which waits for incoming XML events. That’s why this model is also called event-based XML processing. The most common realization of the event-based model is the Simple API for XML (SAX) [11].

Streaming processing has generally a large advantage regarding resource consumption. As no object tree representing the complete XML document has to be created, especially the memory consumption for streaming processing is much lower than for document-based processing. Additionally, there exist hybrid models combining the both aforementioned ones. A well known example is AXIOM [12] used inside the Apache Axis2 framework. It uses streaming processing as underlying technique but offers a tree-kind interface to the developer. One should keep in mind that models like this also greatly benefit from improvements made on the streaming processor. Thus, the approach presented in this paper is also useful for hybrid XML processing.

2.2 Denial-of-Service Robustness

With the resource efficiency properties of the stream-based processing approach comes another advantage which plays an important role especially in commercial Web Service settings. Recent results emphasize the vulnerability of Web Services with regard to Denial-of-Service (DoS) attacks [13]. It has been shown, that the resource requirements to process a SOAP request on the server-side using the conventional tree-based processing approach lead to a severe weakness which can be exploited by DoS attacks. Thus, stream-based processing not only increases efficiency but also—as an outcome—the robustness against resource exhaustion DoS attacks such as coercive parsing and oversize payload [14].

2.3 Prompt Fault Detection

Another benefit of the stream-based processing approach which results in a higher resilience against attack is the ability to detect bogus messages more timely. The tree-based approach requires the whole XML document to be parsed and processed before any application-specific operation can be initiated, transforming it into an in-memory representation of the XML element’s tree structure. For object-oriented programming languages, this typically results in a set of object instances, where each object represents one XML element from the document. These object instances are interlinked according to the XML structure, i.e. they usually contain a parent link to the element’s parent element, and a set of links to its child elements. Such an object tree representation tends to occupy much more in-memory space than the textual representation, as it also has to store object instance metadata.

Assuming the parsed XML document to contain an XML Schema violation (while still being well-formed), the flaws of the tree-based approach become clear. As shown in Figure 1 (a), the XML document must be read completely before any processing on the contained XML elements can start. For Web Service messages, this enables a malicious SOAP message sender to feed in a huge XML document of arbitrary, schema-violating contents, which the parser must read in completely before being able to detect the presence of a schema violation. This way, as the size of the SOAP message is in control of the sender, an attacker can cause heavy workloads for parsing XML documents at the server (see also [15], [16]).

In contrast to the tree-based approach, the stream-based XML processing model starts to parse and process a document’s content block wise, e.g. reading 10 KB of data from the XML file, and then running a stream-based parser on this fragment only before processing the next 10 KB. Depending on the distribution of XML elements
in the textual document representation, this results in an irregular stream of events for every XML element the parser extracts from the character blocks, as shown in Figure 1 (b).

For the special case of an XML Schema violation within the processed document, the stream-based approach enables an early processing of the XML contents. This way, a schema violation within a SOAP message can be detected before the message is completely read, enabling the server application to cut off the connection immediately. Thus, the huge document attack described above can not be performed in the same way.

For plain XML documents that do not contain encrypted fragments, the stream-based processing approach turns out to be superior—compared to tree-based approaches—in terms of performance. Once an XML document contains encrypted data fragments, however, the performance advantages vanishes. The issue here consists in the approach taken for processing the XML data contained in the encrypted block.

Again, the more convenient approach consists in storing the encrypted block completely in memory, then applying the cryptographic operation of decryption to it—resulting in a new, plain XML fragment. Then, the new XML fragment is parsed and processed subsequently. This approach is shown in Figure 1 (c). And again, this leads to the same issues in regard to attack exploitability. As the encrypted contents have to be stored twice (cipher
The total memory consumption rises. Additionally, an attacker could send a SOAP message with schema-invalid contents inside a huge encrypted block. Just like described above, this would cause the server to have that content stored and processed completely before being able to determine the invalidity.

Thus, a more efficient approach consists in decrypting encrypted XML document parts “on the fly”, and having their XML contents parsed and processed immediately. This approach, which is described in Section 3.9, is illustrated in Figure 1 (d).

Again, it poses the same restrictions to XML processing applications as the common stream-based XML processing approach does. Nevertheless, in the event of a schema invalidity or any other kind of XML processing fault within the encrypted contents, the approach obviously provides a highly efficient way to determine the fault as soon as possible. Thus, the reception of the full message can be cut off, and the performance overhead for processing these faulty messages can be reduced drastically.

2.4 Prompt Access Control Decision

A similar property to react as timely as possible to certain events can be exploited in access control. By this approach only parts of the message are processed in cases in which insufficient authorization can be determined [17]. Commonly, the security tokens containing the identity, authentication or authorization information are transported inside the same SOAP envelope as the Web Service request that is to be authenticated. Implementations using a tree-based processing model have to read the complete SOAP message before the processing can be started.

Stream-based processing instead allows one to access identity, authentication and authorization information contained in the request much earlier and thus enables one to reject bogus or other unauthorized messages much more timely, saving compute and memory resources. In cases of flooding attacks based on captured and replied messages, the stream-based processing again provides mechanisms to be more resilient and dependable in the presence of such kind of attacks.

2.5 Complex Programming Model

The stream-based processing approach has a variety of advantages in comparison to the tree-based processing approach. Nevertheless, streaming processing has not found wide-spread adoption yet. The more complex and inconvenient programming model is the main reason that the stream-based processing was mostly neglected until now. Without random document access and without the possibility of backward navigation a lot of operations are difficult to implement. This includes operations for processing a SOAP message containing WS-Security mechanisms, e.g. evaluation of forward and backward

references for digital signatures or evaluation of XPath expressions.

With the increasing usage of Web Services in real-world and mission-critical business processes, however, the advantages of stream-based processing are expected to out-weight this drawback in favor of a more efficient and high-performance Web Service processing. First attempts towards this directions have already been observed in the Web Service framework family from the Apache Foundation. Apache Axis [18] and Apache Axis2 [19] are the most widespread Web Service frameworks for Java. While Axis relies on the DOM processing model for passing SOAP messages between the internal processing handler, its successor Axis2 builds on a STAX-based processing model offering a convenient tree-like access to the SOAP message in addition.

However, the WS-Security component used in Axis2, called Rampart [20], still relies on tree-based processing, most probably because of the difficulties mentioned above regarding especially the handling of backwards references. Thus, for SOAP messages containing security means, the advantages of the underlying streaming message processing in Axis2 are not accessible.

2.6 Open Challenges

It gets clear that in order to effectively use stream-based SOAP processing (either solely or as a foundation for other processing models) a fully integrated and comprehensive stream-based WS-Security processing engine is required. To the knowledge of the authors, such an engine does not exist.

There has been work on stream-based processing of XML signatures [21] and also on stream-based decryption and encryption of XML documents [22]. However, these solution have some shortcomings and are not sufficient for full WS-Security processing integrated into overall SOAP processing.

In this paper, the concepts and algorithms for a comprehensive stream-based WS-Security engine is introduced and discussed. It has the following capabilities (differing from the prior work on this topic):

- processing XML signatures with backward references
- handling combination of signature and encryption in any order, number and nesting degree of
- resolution of cryptographic material including encrypted keys
- conformance to WS-I Security Profile [23]
- integration with streaming access control decision [17]

The paper therefore contributes a crucial bit of new technology and closes an existing gap in the stream-based processing chain towards more efficient and dependable Web Services.
3 Streaming WS-Security Processing

In this section, the algorithms for processing WS-Security enriched SOAP messages in a streaming manner are presented and discussed. To understand the algorithms and the problems solved by them, first of all an introduction of the WS-Security elements is given.

3.1 WS-Security

In contrast to most “classic” communication protocols, Web Services do not rely on transport-oriented security means (like TLS/SSL [25]) but on message-oriented security. The most important specification addressing this topic is WS-Security [26], defining how to provide integrity, confidentiality and authentication for SOAP messages. Basically, WS-Security defines a SOAP header (wsse:Security) that carries the WS-Security extensions. Additionally, it defines how existing XML security standards like XML Signature [27] and XML Encryption [28] are applied to SOAP messages.

For processing a WS-Security-enriched SOAP message at the server-side the following steps must be performed (not necessarily in this order):

- processing the WS-Security header
- verifying signed blocks
- decrypting encrypted blocks

This implies that not only processing the individual parts must be considered but also the references between the WS-Security components. This is especially important in the context of stream-based processing, since arbitrary navigation between message parts is not possible in this processing model.

Figure 2 shows an example of a WS-Security-secured SOAP message containing these references. Security tokens contain identity information and cryptographic material (typically an X.509 certificate) and are used inside signatures and encrypted keys and are backward referenced from those. Encrypted key elements contain a (symmetric) cryptographic key, which is asymmetrically encrypted using the public key of the recipient. This symmetric key is used for encrypting message parts (at the client side) and also for decrypting the encrypted blocks (at the server-side). Encrypted keys must occur inside the message before the corresponding encrypted blocks.

Finally, XML signatures have the following structure.

```
<ds:Signature>
  <ds:SignedInfo>
    <ds:CanonicalizationMethod/>
    <ds:SignatureMethod/>
    <ds:Reference @URI>
      <ds:Transforms>...</Transforms>
      <ds:DigestMethod/>
      <ds:DigestValue>... </DigestValue>
    </ds:Reference>
  </ds:SignedInfo>
  <ds:SignatureValue>...</SignatureValue>
  <ds:KeyInfo>...</KeyInfo>?
</ds:Signature>
```

The signature holds—in addition to specifying the cryptographic algorithms—a ds:Reference element for every signed block, the cryptographic signature value of the ds:SignedInfo element and a reference to the key necessary for validating the signature. A ds:Reference element itself contains a reference to the signed block, optionally some transformations and the cryptographic hash value of the signed block. References to signed blocks can be either backward or forward references. This has to be taken into account for the processing algorithm. There are several possibilities for realizing the reference. However, only references according to the XPointer specification [29] are recommended (see e.g. WS-I Basic Security Profile [23]). Thus, in the following we assume, that the referenced element contains an attribute of the form Id="myIdentification" and is referenced using the URI "#myIdentification" inside the ds:Reference element.

3.2 Architecture

Figure 3 shows the architecture of the system for stream-based processing of WS-Security-enriched SOAP messages called CheckWay [30]. It operates on SAX events.
created by a SAX parser and contains four types of Event Handlers. Instances of these Event Handler types are instantiated on-demand and linked together in an Event Handler chain operating on the stream of XML events. The first handler is responsible for processing the WS-Security header. As the header has a fixed defined position inside the SOAP message, the handler can be statically inserted inside the handler processing chain. For signed and encrypted blocks, however, this is different. These may occur at nearly arbitrary positions inside the SOAP message, and can even be nested inside each other. Thus, the Dispatcher handler is responsible for detecting signed and encrypted blocks and inserting a respective handler into the processing chain (at which position will be discussed below).

While detecting encrypted blocks is trivial (they start with the element xenc:EncryptedData), detecting signed blocks is more difficult as those elements are not explicitly marked. For forward references, the signature elements are (regarding the document order) before the signed block. Therefore, forward referenced signed blocks can be detected by comparing the ID attribute of that element with the list of references from the signature elements processed before. For backward references there is no possibility for a definite decision if an element is signed or not. The following solution for this problem has been developed. Every element before the end of SOAP header (only there backward references are possible) that contains an ID attribute is regarded as potentially signed and therefore the “signed block processing” is started. At the end of such a block the ID and the result of the signed block processing (i.e. the digest of this block) is stored. When processing a signature the included references are compared to the IDs stored from the potentially signed blocks and the stored digest is verified by comparison to the one inside the signature element.

3.3 Notation

Event-based processing is typically described using Finite State Machines (FSM) [31] with automaton transitions triggered by the events (here: SAX events). Thus, the notation of FSMs will be used in the subsequent sections to describe the algorithmic details of the WS-Security processing components shown in Figure 3. The most important events (shown in the state diagrams) are:

- **start(a, @x = c):** begin of element a, containing the attribute x with the value c
- **content(c):** XML text content with value c
- **end(a):** end of element a

The output of the automata are method invocations which initiate further processing described separately in the text.

The following values are stored during message processing:

- **Ref:** an ordered list of references to signed and encrypted blocks inside the SOAP message
- **CompletedDigest:** a set of references and digests of potentially signed blocks (used for backward references)
- **OpenDigest:** a set of references and digests from signature elements, which were not compared to the respective signed block (used for forward references)
- **Key:** a set of references and cryptographic keys from wsse:BinarySecurityToken elements
- **EncKey:** a set of references and cryptographic keys from xenc:EncryptedKey elements

Additionally, the following notation is used for cryptographic keys. Let key be the key contained in encrypted form inside the xenc:CipherData element in the encrypted key element. Let alg denote the algorithm, and keypub the key used for encrypting key. Let keypriv be the private key belonging to keypub.

3.4 WS-Security Header Processing Automaton

Figure 4 shows the automaton for the event-based processing of the WS-Security header. The automaton is started by the element wsse:Security inside the SOAP header. This element can occur more than once inside a SOAP message, but only once per recipient. Thus, the element is unambiguous when taking the attribute soap11:actor or soap12:role into account.

As one can see in the automaton, the WS-Security header can contain security tokens, timestamps, signatures, encrypted keys and references to encrypted blocks in arbitrary order. The method invocations triggered during processing of the WS-Security header are:

- **storeToken(ref, char):** the value char is decoded to the key key and stored inside the list Key (2)
- **checkTimestamp(c, e):** check, if the timestamp is valid, e.g. if $e > c$ and $e < \text{now}$ (with now the current time) (3)
- **storeReference(ref, type):** add (ref, type) to the end of the list Ref (7)

Some of the WS-Security header elements are more complex and need to be discussed separately.

3.5 Key Information from a ds:KeyInfo element

The ds:KeyInfo element is a general element for transporting information about cryptographic keys. In the context of WS-Security, the only allowed content is a wsse:SecurityTokenReference element. This element then contains a reference to a security token, stored either external or inside the same message. If the security token is inside the same message, the token must occur either before the token or as a direct child element. In either case, at the moment the ds:KeyInfo is read, the related security token is already known or can at least be effectively obtained.

From the security token the cryptographic keys needed for decryption and signature verification can be derived. There are different possibilities for this procedure. A common case is the use of an X.509 certificate
as security token. This certificate contains the public key for signature verification. The associated private key is locally stored at the server system (e.g. for Java-based system typically inside a Java Keystore) and can be identified through the identifiers inside the certificate.

3.6 Encrypted Key Processing Automaton

Figure 5 shows the automaton for processing an xenc:EncryptedKey element contained in the WS-Security header. The processing starts with reading the encryption algorithm \texttt{alg} (1).

Inside the \texttt{ds:KeyInfo} element (2) a hint to the key pair \texttt{key\_priv} and \texttt{key\_pub} is given (see above).

The key \texttt{key\_priv} is used for initializing the decryption algorithm inside the function \texttt{initDecryption(alg)} (4). The function \texttt{decrypt(char)} decrypts then the content of the xenc:CipherData element using this algorithm in conjunction with \texttt{key\_priv}. The result is the (symmetric) key \texttt{key}, that is used later to decrypt encrypted content.

The references stated inside the xenc:ReferenceList claim the usage of the current key for those encrypted blocks. Thus the \texttt{storeKey(...)} function adds the pair \texttt{(ref, key)} to \texttt{EncKey} (7) to enable the decryption of the appropriate encrypted block (see below). Additionally, the pair \texttt{(ref, Enc)} is added to the end of the list of security references \texttt{Ref}.

3.7 Signature Processing Automaton

Figure 6 shows the automaton for processing a \texttt{ds:Signature} element from the WS-Security header. For verifying the signature value, the \texttt{ds:SignedInfo} block must be canonicalized and hashed. Thus, at the begin of that element the canonicalization and hashing is started by the function \texttt{startHashing()} (1).

The canonicalization algorithms for the \texttt{ds:SignedInfo} block are read (2). The WS-I Basic Security Profile includes only \textit{Exclusive C14N} [32] as canonicalization algorithm.

The signature algorithm (e.g. “RSA with SHA-1”) is read (3).

The reference \texttt{ref} is read from the URI attribute of the element \texttt{ds:Reference} (4).

The transformation algorithms are read and the set of transformations is stored into \texttt{t} (6, 7).

The hashing algorithm for the signed block is read. The digest value is read (10) and the function \texttt{checkDigest(char, t, ref)} is executed.

- If there exists a \texttt{D} with \texttt{(ref, D) \in CompletedDigest}, \texttt{ref} is a backward reference and thus the referenced element was already processed. Then, the value digest with \texttt{(t, digest) \in D} is the calculated digest of this element. If \texttt{char = digest} the calculated and the stated digest are equal. If this is not true, the signature is invalid and the SOAP message processing is stopped with an exception.
Fig. 5: Automaton for Encrypted Key Processing

Fig. 6: Automaton for Signature Element Processing
If no such $D$ exists, $ref$ is a forward reference and 
$(ref, t, digest)$ is added to $OpenDigest$.

At the end of the ds:Reference block the function 
storeReference(ref, type) is executed (11). This adds 
$(ref, type)$ to the end of the list $ref$.

At the end of the ds:SignedInfo block the digest 
of this block has completely been calculated: $si := \text{stopHashing}()$ (12).

The signature value $sig$ is read from the 
ds:SignatureValue element (14).

The key $key$ needed for signature verification is gathered 
from the ds:SignatureValue (as described in 
Section 3.5) (15).

At the end of the signature element the verification 
procedure of the signature is executed: 
verifySignature(key, si, sig). For that purpose, the “normal” signature verification function of the signature 
for the digest $si$, the signature value $sig$ and the key 
$key$ is executed. In case of RSA this is e.g.: calculate 
$E_{key}(Pad(sig))$ and compare the result to $si$ (Pad is 
the padding function according to PKCS #1 [33]).

### 3.8 Sequence of Encryption and Signature

A SOAP message can contain multiple encrypted or 
signed blocks at arbitrary positions inside the document. 
The Dispatcher (see Figure 3) detects these blocks as 
discussed above and inserts dynamically and as required 
instantiated handlers for processing these blocks 
into the event chain. To insert the handler at the correct 
position inside the chain, it is necessary to determine the 
order in which the encryption and signature operations 
were originally applied to the message. This is a common 
problem for processing XML Signature and XML Encryption. Repeated encryption of an element can be 
without a doubt detected and for repeated signing the order is irrelevant. However, problems occur from the 
combination of encryption and signature. In some cases it can not be decided, if an element was first signed 
and then encrypted or vice versa. The following message 
fragment illustrates such a situation:

```xml
<ds:Signature>
  ...<ds:Reference URI="#sig-1"/>
  ...</ds:Signature>
  ...
<ns1:a Id="#sig-1">
  <xenc:EncryptedData Id="#enc-1"/>
  ...</xenc:EncryptedData>
</ns1:a>
```

However, for verifying the signature it is necessary 
to know the correct order in which the security 
mechanisms have been applied. As a hint to this, 
the wsse:Security header contains the security reference 
(inside ds:Signature, xenc:EncryptedKey and 
xenc:ReferenceList) in exactly the same order 
as the processing operations are to be executed (see [23], 
R3212). This can also be used for event-based decrypting 
and signature verification.

As encrypted blocks are always forward referenced, 
a combination of potentially signed blocks and encrypted 
blocks can never occur. Thus, for a potentially signed 
block the handler can always be attached to the end of the event chain. Hence, in the following paragraphs 
only forward referenced signed or encrypted blocks are discussed.

Let $Ref = \{(ref_1, op_1),..., (ref_n, op_n)\}$ and let 
$(H_1, ref_1),..., (H_k, ref_k)$ with $i_j \in \{1,...,n\}$ the handler 
(and the according reference) inside the current 
event chain. If a signed or encrypted block with the ID 
$ref$ occurs, then $ref \in Ref$ (as it is a forward reference) 
and thus there exists a $m \in \{1,...,n\}$ with $ref = ref_m$. 
A new instance of the particular processing handler 
is created and inserted at position $i$ (i.e. after handler 
$H_{j-1}$), whereas $i_j < m$ and $m < i$. This means that 
the order of the references in $Ref$ induce the order of the handlers. The following two examples (with extremely simplifed SOAP messages) illustrate this procedure.

**Example 1 (Encrypt before Signing)**

```xml
<ds:Signature>
  ...<ds:Reference URI="#sig-1"/>
  ...</ds:Signature>
  ...
<ds:Reference URI="#sig-2"/>
  ...</ds:Signature>
  ...
<xenc:ReferenceList>
  <xenc:DataReference URI="#enc-1"/>
  ...</xenc:ReferenceList>
  ...
<env:Body Id="#sig-1">
  ...
  </env:Body>
</ns1:a>
```

Here, the order of the secured elements (i.e. those containing an ID attribute) is the same as the order of the security references 
$(Ref = \{(\text{sig-1, Sig}),\text{(sig-2, Sig),(enc-1, Enc)})\})$. Thus, the handlers are chained up in the following 
order: $(H^\text{Sig,sig-1}), (H^\text{Sig,sig-2}), (H^\text{Enc,enc-1})$. Figure 7 shows the flow of the XML events through the 
resulting processing chain. According to the fact that 
the encryption was applied first, one can see, that the 
decryption is performed after both signature verification 
steps.

**Example 2 (Sign before Encrypting)**

```xml
<ds:Signature>
  ...<ds:Reference URI="#sig-1"/>
  ...</ds:Signature>
  ...
<xenc:ReferenceList>
  <xenc:DataReference URI="#enc-1"/>
  ...
</xenc:ReferenceList>
  ...
<env:Body>
  ...
</env:Body>
```


The processing chain has the following members: $(H^{\text{Sig}}_{\text{sig-1}}), (H^{\text{Sig}}_{\text{sig-2}})$.

As $\text{Ref} = \{(\text{sig-1}, \text{Sig}), (\text{enc-1}, \text{Enc}), (\text{sig-2}, \text{Sig})\}$, the encryption handler for processing the element with the ID $\text{enc-1}$ is inserted between the signed block handlers. Figure 8 shows the flow of the XML events through the resulting processing chain. One can observe, that the second signed block handler operates on the decrypted elements. This fits to the fact, that originally this element was first signed and then encrypted.

3.9 Encrypted Block Processing Automaton

If during message processing an element $\text{xenc:EncryptedData}$ (with ID $\text{ref}$) is read, the Dispatcher creates a new instance of the encrypted block handler and inserts it into the handler chain like the one presented above. The encrypted block handler itself implements the event processing shown in the automaton in Figure 9. It starts with reading the encryption algorithm $\text{alg}$ (1). For the key needed for decrypting there are two possibilities. Either it is given inside the $\text{ds:KeyInfo}$ element (2) or it is given by an encrypted key. In the latter case there exists a key with $\text{(ref, key) \in EncKey}$.

The parameters $\text{alg}$ and $\text{key}$ are used for initializing the encryption function (4); $\text{initDecryption(alg, key)}$. The text content of the $\text{xenc:CipherValue}$ element is decrypted using this function (5); $\text{decrypt(char)}$. The encrypted data represents the encrypted XML fragment (in serialized form). This data is processed by a SAX parser which creates the corresponding XML events. Thus, unlike all other handlers, the handler for encrypted blocks modified the event stream. All incoming events are absorbed and the events created from the decrypted content are emitted. This can also be seen from the examples shown in Figure 7 and 8.

After having processed the last event of the encrypted block (i.e. the closing tag of the $\text{xenc:EncryptedData}$) the encrypted block handler removes itself from the event chain.

3.10 Signed Block Detection and Processing

As stated in Section 3.2, there are two possibilities for detecting signed blocks. Before the WS-Security header has been processed, all elements containing an ID attribute must be regarded as potentially signed. For those elements a new instance of the signed block handler is created and inserted into the event chain (as described above). The handler executes the canonicalization and hashing of this message fragment and stores the result (together with the ID value) at the end of the list $\text{CompletedDigest}$. As the hashing algorithm is unknown at this stage of message processing, the hashing step is executed for all allowed hashing algorithms in parallel (currently: SHA-1 and SHA-256 [34]). If such a block is actually signed, these values are evaluated later during processing of the signature (see Section 3.7, 7th item).

As the WS-Security header already has been processed at this stage, signed blocks can only be referenced using forward references. Thus, only elements with IDs known from processing the signature elements must be taken into account. If an element with an ID attribute with the value $\text{ref}$ is read and there exists a $(\text{ref, digest}) \in \text{OpenDigest}$, this element is signed. In this case, a new instance of the signed block handler is created and inserted into the event chain (as described above). As before, the handler executes the canonicalization and hashing of this message fragment. If the fragment has been completely processed the resulting digest value is compared to $\text{digest}$. If they are different, the signature

---

Fig. 7: Event Processing of Signed and Encrypted Blocks – Example 1 (Encrypt before Signing)
Fig. 8: Event Processing of Signed and Encrypted Blocks – Example 2 (Sign before Encrypting)

Fig. 9: Automaton for Processing Encrypted Blocks

is invalid and the SOAP message processing is stopped with an exception. Otherwise, \((\text{ref}, \text{digest})\) is removed from \(\text{OpenDigest}\).

In either case the signed block handler is removed from the event chain. At the end of the SOAP message, \(\text{OpenDigest}\) is checked. If it is not empty, a block referenced from a signature is missing and thus the signature is invalid. In this case, also the SOAP message processing is stopped with an exception.

### 4 Evaluation

The algorithms introduced in this paper provide a comprehensive framework for event-based WS-Security processing. With this contribution, the implementation of SOAP message processing can be realized in a streaming manner including the processing of security means, resulting in a significant increase in efficiency compared to “traditional” and currently mainly deployed tree-based approaches. First of all, if the SOAP message is invalid the message is only read and parsed up to the point where any flaw is detected. This is especially important, when the message was part of a Denial-of-Service attack, or if a negative Access Control decision can be made at an early stage of the message processing. Further, even if only valid messages are considered the event-based approach has many advantages, especially with regard to memory consumption. It is well known that in-memory document trees are 10 to 100 times larger than the serialized form, while the XML events only increase the size by factor 1 to 1.5.

To prove these theoretical advantages, the approach presented here was prototypically implemented in a system called \(\text{CheckWay}\). This system was extensively tested and compared to standard frameworks. The tests were performed on an Athlon 64 3000+ with 1 GByte memory. The Java VM used is version 1.6 from Sun Microsystems.
The frameworks used for comparison are Apache Axis2 1.3 with the security module Apache Rampart 1.3. In all test scenarios runtime and memory consumptions needed for message processing at the server side have been measured. For metering the memory consumption, the Java Monitoring & Management Console JConsole [35]—which is part of the Java development toolkit since version 1.5—has been used. This tool allows connecting to a running Java VM and a readout of diverse parameters, amongst others the current memory consumption.

For evaluation purposes a number of test scenarios have been executed. This included: messages containing encrypted parts; messages containing (correctly and incorrectly) signed parts; messages containing nested encrypted and signed parts; policy and schema violation in different parts of the message including inside encrypted parts; messages from authorized and non-authorized senders. Here, the results of the three most meaningful test scenarios are presented and discussed.

For the first one, the Web Service was invoked with a series of messages with increasing number of elements inside the encrypted SOAP body (e.g. realized with an increasing parameter list). The test compared the resource consumption of CheckWay with Apache Rampart. Figures 10 and 11 shows the memory consumption and the runtime of both systems for different message sizes. It is noticeable that the runtime for CheckWay is only slightly higher than for Rampart (converging to approximately 10%). However, for the memory consumption CheckWay is magnitudes better than Rampart. Rampart needs between 30 to 150 times more memory than our approach.

The second scenario illustrates that adoption of the streaming approach not only leads to quantitative performance improvements but also has fundamental advantages compared to document-based models. Again, an encrypted message is used, this time containing a policy violation inside the encrypted message. In the concrete...
example, this violation was an oversized message violating a limitation inside the schema. This attack pattern is known as Attack Obfuscation (see [13] for details on execution and effects on the attacked system).

For a streaming XML processor it is possible to decrypt and parse the content in one step, hence detecting such attacks rapidly. In contrast, the DOM approach requires a complete parsing run over the full message prior to decryption. This already includes all of the encrypted oversized payload’s ciphertext, without DOM being able to detect the policy violation contained in the encrypted fragment. Since it is in the control of the attacker to decide on the amount of ciphertext it sends to the victim, this implies that in the DOM model an attacker can choose an “unlimited” amount of ciphertext (or even garbage text denoted as being ciphertext) in order to force the victim’s system to parse that content into an in-memory DOM tree representation. Such kinds of attacks can be detected and fended with the streaming approach, but not with common tree-based approaches.

The lower curve in Figures 12 shows the resulting memory consumption of the overall SOAP processing system. One can see that the message is only processed up to the point where the policy violation occurs. Thus, for example by using a schema containing size constraints the overall memory and runtime needs can be limited independent of the actual message. In [14] it is shown how such a schema can be created from a WSDL. The upper curve shows for comparison the resource consumption for an uninterrupted processing of the complete message. For a document-based processing system, the values always have such a characteristic. Thus, independent of additional validation components (e.g., schema validation) it is not possible to reasonably limit the resource consumption in order to fend the Attack Obfuscation attack. Our streaming system on the other hand effectively fend this attack.

In the third scenario, an access control violation is assumed. This occurs for example if an attacker has no valid authentication credentials for the service he is attacking. In [17] it is shown, that (in all practical relevant cases) no later than at the end of the security header a negative access control decision can be made, even if the attacker uses the credentials of a valid eavesdropped message. Using streaming processing of the SOAP message, the access control decision can be made at an early stage of the security processing leading to constant runtime and memory consumption independent of the actual message size (see Figure 13). This is an important property and difference to document-based systems. In these, an attacker can induce a large resource consumption using such a message and can create a Denial-of-Service despite of an working access control system.

5 Conclusion

The paper introduces a comprehensive framework for event-based WS-Security processing. Although the streaming processing of XML is known and understood for almost as long as the existence of the XML standard itself, the exploitation of this processing model in the presence of XML security is not. Due to the lack of algorithms for event-based processing of WS-Security, most SOAP frameworks include the option for streaming based processing only for unsecured SOAP messages. As soon as these messages are secured by WS-Security mechanisms, the security processing is performed relying on the DOM tree of the secured message, hence loosing all advantage of the event-based processing model.

With this contribution, the implementation of SOAP message processing can be realized in a streaming manner including the processing of security means, resulting in significant improvements compared to “traditional” and currently mainly deployed tree-based approaches. The main advantages of the streaming model include an increased efficiency in terms of resource-consumption and an enhanced robustness against different kinds of DoS attacks. This paper introduces the concepts and algorithms for a comprehensive stream-based WS-Security component. By implementing configurable chains of stream processing components, a streaming WS-Security validator has been developed, which verifies and decrypts messages with signed and/or encrypted parts against a security policy. The solution can handle any order, number and nesting degree of signature and encryption operations filling the gap in the stream-based processing chain towards more efficient and dependable Web Services.

The evaluation of the implemented prototype system impressively shows the impact of the streaming WS-Security processing on resource consumption, which was also confirmed in independent works (cf. e.g. [36]). The highest effect is gained for messages containing some kind of policy violation. As these messages are potentially malicious here protection from resource consumption is needed most and thus a high robustness to attacks is achieved.

Next steps include the extension of the above mentioned configurable processing chains with components for handling attachments and for checking message sequences for BPEL-composed Web Services. First preliminary results are available [37], but need still to be integrated and evaluated. Another related research direction is heading towards the development of consistent security policies in large workflows.

References


Nils GRUSCHKA studied computer science at the Christian-Albrechts-University of Kiel, Germany, and received his Ph.D. in 2008. His thesis presents methods for Web Service protection using efficient Web Service message validation. Currently he works as a research scientist at NEC Laboratories Europe in the field of Web Service security. Nils is a member of the ACM and GI.

Meiko JENSEN studied computer science at the Christian-Albrechts-University of Kiel, Germany, and received his degree in 2007. Currently he is a Ph.D. student at the University of Bochum, researching in the field of Web Service security, and is particularly interested in service composition security, XML security, security modeling, and attacks on Web Services. Meiko is a member of ACM, IEEE, and GI.

Luigi LO IACONO studied computer sciences with a major in systems and security engineering and received his Ph.D. from the University of Siegen (Germany) in 2005. He has previously worked in academic and industry research labs and currently works as a lecturer at the European University of Applied Science.

Norbert LUTTENBERGER worked at Siemens AG in Erlangen and the IBM European Networking Centre in Heidelberg. He received his Ph.D. in Engineering in 1989 from the University of Erlangen-Nürnberg. He became professor for Data Communications and Computer Networks at the University for Applied Sciences in Gelsenkirchen, Germany. Since 2000 he is professor for Communication Systems at the Christian-Albrechts-University in Kiel, Germany. His research interests include wireless sensor networks and XML language technology.