5 WSSL API

The WSSL API was implemented in the Java programming language [19] using four main Java classes: WSSLTransform, NeededKey, WSSLSignatureVerification, and WSSLError. The WSSLTransform class contains the main methods used to secure and unsecure SOAP messages, the NeededKey object stores the content of the key_definition element of the WSSL document used to get the necessary information of the key, the WSSLSignatureVerification object stores the results of the verification of all the digital signatures in the secured SOAP document, and the WSSLError class reports error messages encountered during the securing and unsecuring of a SOAP message. In the following paragraph, we will explain the methods and attributes of these classes and describe how to build the client-proxy and server-side program in details.

5.1. The methods and attributes of class WSSLTransform

Class WSSLTransform consists of the main methods for encrypting and decrypting operation. It provides two constructors (see Figure 21). The constructor public WSSLTransform(String WSSLDocPath, short Mode) initializes the instance variables of the new WSSLTransform object including the default pathnames of cryptographic keys and algorithms. The first parameter is used to locate the pathnames of the WSSL document. The second parameter represents the role of the proxy. There are two kinds of values to select the mode of the proxy. One is REQUESTER_MODE, and the other is PROVIDER_MODE. The programmer can select the mode according to the role of the proxy. If the proxy is the client-side one, the mode should be REQUESTER_MODE. On the contrary, if the proxy represents the server-side one, the mode will be applied to PROVIDER_MODE. Finally, the constructor public WSSLTransform(String WSSLDocPath, short Mode, String keyManagerDir) is like the first constructor. The different is that the

second constructor can locates the pathnames of the directory where the key are stored directly.

public WSSLTransform(String WSSLDocPath , short Mode) **Parameters:** WSSLDocPath – The pathname string of the WSSL document . Mode - The value to represent the role of the proxy . The "WSSLTransform.REQUESTER_MODE" value is the client-side proxy. The "WSSLTransform.PROVIDER_MODE" value is the server-side proxy.
public WSSLTransform(String WSSLDocPath , short Mode , String keyManagerDir) **Parameters:** WSSLDocPaht – The pathname string of the WSSL document . Mode - the value to represent the role of the proxy . keyManagerDir – The pathname of the directory where the keys are stored.

Figure 21 : The constructor summary of WSSLTransform class

We classify the methods of WSSLTransform class into the following five groups: (1) environment settings, (2) securing and unsecuring processes, (3) needed keys setting, (4) signature verification, and (5) error handling. The remainder of this section describes these methods.

5.1.1. Environment Settings

The environment-setting methods are used to set up the operation environment of the WSSL API (see Figure 22).

public void getAlgorithmContainerPath( ) This method is used to get the pathname of the directory where the

cryptographic algorithms are stored.

Returns:

The string of the path of the algorithm container.

public void setAlgorithmContainerPath(String algorithmDir)

This method is used to set up the pathname of the directory where the cryptographic algorithms are stored.

Parameters:

algorithmDir - The string of the path.

public void turnOnMessage( )

The report message will be shown.

public void turnOffMessage( )

The report message will not be shown.

public int getSecuritySOAPForamt( )

To get the value which is used to represent the format of the secured SOAP message.

Returns:

The value of the format represented.

If the value is 1, the encrypted XML element is conformed to the WS-Security.

If the value is 2, the encrypted XML element is conformed to the DSL.

public void setSecuritySOAPForamt(short format)

To set up the value which is used to represent the format of the secured SOAP message.

Parameters:

1. The value of "WSSLTransform.WSS_FORAMT" represents that the format of secured SOAP message is conform to the WS-Security.

2. The value of “WSSLTransform.WSSL_FORAMT” represents that the format of secured SOAP message is conform to the DSL.

public String getTempDirPath( )

This method is used to get the pathname of the temporary directory. The temporary directory stores the DSL documents to secure or unsecure the SOAP message. And it also puts the certificates which is attached to the SOAP message to verify the signature.

Returns:

The string of the path of the temporary directory.

public void setTempDirPath(String tempDirPath)

This method is used to set up the pathname of the temporary directory.

Parameters:

tempDirPath - The string of the path.

public String getSecurityInformation( )

To get the information of the security. It contains the path and scope of the encryption. The security information will be showed as follows:

= = = = = =encryption= = = = = =

path:/Envelope/Body/transaction/credit_info scope:element

path:/Envelope/Body/transaction/order_info/person_info scope:element

= = = = = =signature= = = = = =

Time:AFTER

path:/encrypted/Envelope/Body scope:element

Returns:

The string of the information of the security.

public String getUnsecurityInformation( )

To get the information of the unsecurity. It contains the path and scope of the decryption . The unsecurity information will be showed as follows:

= = = = = =decryption= = = = = =

path:/Envelope/Body/confirmation scope:element

= = = = = =verification= = = = = =

Time:BEFORE

path:/Envelope/Body scope:element

Returns:

The string of the information of the unsecurity.

public void setWSSLDocumentPath(String wsslFilePath)

This method is used to set up the pathname of the WSSL document.

Parameters:

wsslFilePath - The string of the path.

public String getWSSLDocumentPath ( )

This method is used to get the pathname of the WSSL document.

Returns:

The string of the path of the WSSL document.

Figure 22 : Environment-setting methods

5.1.2. Securing and unsecuring processes

The secure and unsecure methods are used to perform the encryption and decryption, respectively (see Figure 23). These methods can achieve the particular goal in the different way depending on the name of the method. For example, if SOAP message want to be secured as the type of XML DOM tree, the secureAsNode method will be adopted. In practice, the target and resulting data can be encrypted or decrypted as

XML document, XML DOM tree, InputStreanm, byte Array. Beside, before executing these methods, it is necessary to use the setSOAPMessage method to set up the source of SOAP message.

Four example setSOAPMessage methods are as follows:
public void setSOAPMessage(Node sourceSOAP) Setting up the source of SOAP message to encrypt or decrypt. **Parameters:** sourceSOAP - The data type is org.w3c.dom.Node.
public void setSOAPMessage(String pathSOAP) Setting up the path of SOAP message to encrypt or decrypt. **Parameters:** pathSOAP –The string is the path of XML document.
public void setSOAPMessage(byte[] sourceSOAP) Setting up the source of SOAP message to encrypt or decrypt. **Parameters:** sourceSOAP - The type is a byte array.
public void setSOAPMessage(InputStream sourceSOAP) Setting up the source of SOAP message to encrypt or decrypt. **Parameters:** sourceSOAP - The type is java.io.InputStream..
Four example secure methods are as follows:
public Node secureAsNode( ) The target SOAP message will be secured as XML Node tree. **Return:**

The type of the secured SOAP message is XML Node tree.
public boolean secureAsFile(String outputPathSOAP) The target SOAP message will be secured as XML document. **Parameters:** outputPathSOAP –The path of the resulting SOAP message **Return:** True if the secured operation is successful.
public byte[ ] secureAsByteArray( ) The target SOAP message will be secured as a byte array. **Return:** The type of the secured SOAP message is a byte array.
public InputStream secureAsInputStream( ) The target SOAP message will be secured as java.io.InputStream. **Return:** The type of the secured SOAP message is a java.io.InputStream.
Four example unsecure methods are as follows:
public Node unsecureAsNode( ) The target SOAP message will be unsecured as XML Node tree. **Return:** The type of the unsecured SOAP message is XML Node tree.
public boolean unsecureAsFile(String outputPathSOAP) The target SOAP message will be unsecured as XML document. **Parameters:** outputPathSOAP – The path of the resulting SOAP message **Return:**

True if the unsecured operation is successful.
public byte[ ] unsecureAsByteArray( ) The type of the target SOAP message will be unsecured as a byte array. **Return:** The type of the unsecured SOAP message is a byte array.
public InputStream unsecureAsInputStream( ) The target SOAP message will be unsecured as java.io.InputStream. **Return:** The type of the unsecured SOAP message is java.io.InputStream.

Figure 23 : Secure and unsecure

5.1.3. Needed keys setting

The WSSLTransform class provides the method listed in Figure 24 for the programmer to get or set up the information of the needed key easily. The programmer can uses the getNeededKeys method to obtain the vector that stores NeededKey objects. The main function of the NeededKey class is to get the information of the key to show the specification of the key. Moreover, it also provide the methods for users to set up the key (see Figure 25).

When getting the NeededKey object, it contains the sufficient information itself to secure or unsecure. And, the information in which the Neededkey class involves is defined in the key_specification and message_to_user subelement of the key_definition element. The programmer can use the methods of the NeededKey class to construct the system which interacts with users. For example, method getKey_link gets the link name of the key definition that is referred to the security pattern section in the WSSL document, method getKey_mode acquires the value which is in the mode attribute of the

Key_definition element, method getKey_property obtains the content of the subelement which is under the key_specification element by the name of the element, and method getMessage_to_user gets the comment of the key.

After understanding the purpose of the key, the method setup which the NeededKey class provides can be used to load the key from the file directory or java key store. Beside, the WSSLTransform class also supply the method addKey and addKeyFromKeyStore to set up key manually.

public Vector getNeededKeys( ) This gets a vector that stores NeededKey objects. **Return:** A vector that contains NeededKey objects.
public void addKey(String keyID, String keyMode, String keyName, String keyType,String keyLocation) This key will be loaded from the file system. **Parameter:** keyID - The linking name of the key, it will be refered by the definition in a security pattern or digital_signature. keyMode – The mode of the key. The string value can be “dynamic_selected_key” or “static_key”. keyName - The name of the key file. keyType - Algorithm name. It can be “X.509” or “RSA”. keyDir - The directory where the key file is stored.

public void addKeyFromKeyStore (String keyID,String keyMode,

StringkeyName, String keyPass, String keyType,

String keyStore,String keyStorePass)

This key will be loaded from the java key store.

Parameter:

keyID - The linking name of the key, it will be refered by the definition in a security pattern or digital_signature.

keyMode –The mode of the key. The string value can be “dynamic_selected_key” or “static_key”.

keyName - The key name is in the java key store.

keyPass - The password of accessing the key.

keyType - Algorithm name, e.g. RSA, X.509

keyStore - The file name of the keystore.

keyStorePass - The password of the keystore.

Figure 24 : The needed key setting

public void setup(String key_path)

This key will be loaded from the path of the key.

Parameter:

key_path –Tthe path of the key file.

public void setup(String key_name, String key_location)

This key will be loaded from the file system.

Parameter:

key_name - The name of the key file.

key_location - The directory where the key file is stored.

public void setup(String key_name, String key_pass, String key_stroe,

String key_store_pass)

This key will be loaded from the java key store.

Parameter:

key_name - The key name is in the java key store.

key_pass - The password of accessing the key.

key_store - The file name of the keystore.

key_store_pass - The password of the keystore.

public String getKey_link( )

This gets the link name of the key_definition element.

Return:

The string of link name of the key.

public String getKey_mode( )

This gets the mode which the key_definition element adopts.

Return:

There are three kinds of string value. The value could be "static_key","dynamically_selected_key", or "key_applied_to_signature".

public String getKey_property( )

This gets the name and content of all the subelements which is in the key_specification element. It will be formulate as "name:content\n". ('\n' is linefeed char.)

For example :

```
<key_specification>
    <key_type>certificate</key_type>
    <PKI>X.509v3</PKI>
    <key_issuer>CN=tomcat,OU=ICLAB</key_issuer>
```

</key_specification>

The value of the string will be

key_type: certificate\nPKI: X.509v3\n key_issuer:CN=tomcat,OU=ICLAB\n.

Return:

The string of “name:content\n”.

public String getKey_property(String property)

This obtains the content of the subelement of the key_specification element by the element name. For example, if the key_specification element is as follows, inputting the value “key_type” will get the content “certificate”.

```
<key_specification>
    <key_type>certificate</key_type>
    <PKI>X.509v3</PKI>
    <key_issuer>CN=tomcat,OU=ICLAB</key_issuer>
</key_specification>
```

Parameter:

property – the string of the element name.

Return:

The string of the element content.

public String getMessage_to_user( )

This gets the content of the message_to_user element which is in the key_defintion element.

Return:

The string of the element content.

public String getKey_name( )

This gets the name of the key file.

Return: The string of the key name.
public int getKey_source( ) This gets the source of the key. **Return:** 0 – The key source is the file directory. 1 – The key source is the java key store.
public String getKey_location( ) This gets the path of the directory where the key file is stored. **Return:** The string of the directory path.
public String getKey_pass( ) This gets the password of the key entity in the java key store. **Return:** The string of the password.
public String getKey_store( ) This gets the path of the java key store. **Return:** The string of the path where the java key store is stored.
public String getKey_store_pass( ) This get the password of the java key store **Return:** The string of the password.

Figure 25 : The method of the NeedeKey class

5.1.4. Signature verification

After each execution of the unsecure method, the result of the signature verification will be store in the WSSLTransform object. The programmer can use the getWSSLSignatureVerifications method shown in Figure 26 to obtain the vector that stores WSSLSignatureVerification objects (see Figure 27). Every WSSLSignatureVerification object can be used to check the correctness of the verification. Besides, it also supply the sufficient method to construct the authorizing mechanism in the proxy program. For example, the programmer can use the getCertificateDate method to get the certificate. By the content of the certificate, the server-side proxy can authorize the appropriate privilege to a service requester.

public Vector getWSSLSignatureVerifications( ) This gets a vector object that stores the WSSLSignatureVerification objects.

Figure 26 : Signature verification

For example “WSSLSignatureVerification” methods are as fellows.
public boolean isResult( ) This gets the result of the verification. **Return:** The boolean value of “true” or “false”.
public String getSignatureName( ) This gets the signature name which defined in the WSSL document. **Return:** The string of the signature name.
public String getSignatureValue( ) This gets the value of the digital signature.

Return:

The string of the digital signature.

public byte[ ] getCertificateData( )

This get the certificate which is used to verify the digital signature.

Return:

A byte array which stores the certificate.

Figure 27 : The methods of the WSSLSignatureVerification class

5.1.5. Error handling

The WSSL API also considers the handling of many errors which occur in the operation of encrypting and decrypting SOAP messages. When unendurable errors such as receiving an incorrect WSSL document or SOAP message, the proxy program implemented by the WSSL API should be terminated. However, the proxy program should tolerate some endurable errors and activate error handling routine to deal with them. For example, when one of the cryptographic keys is incorrect, the proxy program may still be able to decrypt other ciphertexts or verify the digital signature. Since there may be endurable errors during executing the method listed in Figure 23, the WSSLTransform class provides the method getWSSLErrors shown in Figure 28 for programmers to get the error message. The method getWSSLErrors will returns a vector which stores WSSLError objects. The programmer can use getErrorState, getErrorCode, getErrorCause and getErrorMessage in Figure 29 to obtain the error codes (see Figure 43 in Appendix B) and design own error handling routine.

public Vector getWSSLErrors( )

This gets a vector object that stores the WSSLError objects.

Figure 28 : Error handling method

For example “WSSLError” methods are as fellows.

public String getErrorState( )

This gets the error state. There are five situations in the error state. They are listed as follow:

1. The environment setting of the WSSL API.
2. Securing SOAP message in the requester-side.
3. Unsecuring SOAP message in the requester-side.
4. Securing SOAP message in the provider-side.
5. Unseucring SOAP message in the provider-side.

Return:

The string of the value.

public String getErrorCode( )

This gets an error code.

Return:

The string of the value.

public String getErrorCause( )

This gets the cause of the error.

Return:

The string of the value.

public String getErrorMessage( )

This gets a message including the error code and the cause of the error.

Return:

The string of the value.

Figure 29 : the methods of the WSSLError class

5.2. The client-side proxy

Figure 30 lists the pseudo Java code that employs the WSSL API to secure a SOAP document, which could represent the code of the client-side proxy. First, lines S1,1 and S1,2 instantiate an object of the WSSLTransform class. Note that the second parameter should be "WSSLTransform.REQUESTER_MODE". The WSSL API parses the WSSL document "WSSLDoc.xml" during the instantiation of the object. Then, lines S1,3 and S1,4 instruct the WSSL API to read the SOAP message that is to be secured.

Second, the proxy program obtains the required keys, which are those used to secure the SOAP message. These keys can be obtained using "getNeededKey()" (see line S2,1), and are stored in a vector. It is then necessary to set up the keys sequentially. Note that the keys may be stored in various types of media, such as a Java key store, hard disk, USB disk, or smart card. The code should first check the "mode" of the keys and obtain all of their related properties. The setting up can be performed automatically or via interaction with the user. In our example, we assume that the subprogram "USER_INTERFACE_GET_KEY" will help with this task (see line S2,10). Finally, the obtained keys should be set up by invoking "setup()" (see line S2,11).

Third, we are now ready to secure the SOAP message. Line S3,1 creates an object to store the error messages. Note that it is possible for multiple errors to occur in a single operation (i.e., securing or unsecuring). We can secure the SOAP message by invoking "secureSOAP()" (see line S3,2), with the secured SOAP message stored in XML tree "Node". Fourth, the secured SOAP message is sent to the server-side proxy, which will return the secured response message (see line S4,1). Fifth, the received secured SOAP message (i.e., R_s in Figure 3) should be unsecured. Note that there is no need to again set up the required keys since this has already been done in the code segment from

lines S2,1 to S2,9. The invocation of method “unsecureSOAP()” unsecures the SOAP message. In addition to decrypting cipher data in the secured SOAP message, this method verifies all the embedded digital signatures. The method “getSignatureVerification()”is used to obtain the results of verifying digital signatures. Since the result is stored in a vector, we can use a loop to check the individual results (see lines S6,1 to S6,8). Finally, the unsecured SOAP message is sent to the service requester (see line S7,1).

```
        //Obtain an instance of WSSLTransform class.
        // (1) "WSSLDoc.xml" is the WSSL document shown in Figure 3.
        // (2) The parameter WSSLTransform.REQUESTER_MODE is used for the
S1,1    //       client-side proxy.
S1,2    WSSLTransform wsslTransform =
           new WSSLTransform("WSSLDoc.xml",WSSLTransform.REQUESTER_MODE);
        // Receive X from service requester
S1,3    Node requestSOAP = Nework.Receiver_From_Service_Requester();
S1,4    wsslTransform.readSOAPMessage(requestSOAP);


        //Interact with the user to obtain the needed keys
S2,1     Vector neededKeys = wsslTransform.getNeededKey();
S2,2     for(int i=0;i<neededKeys.size();i++) {
S2,3        NeededKey neededKey = (NeededKey) neededKeys.get(i);
S2,4        String mode = neededKey.getKeyMode();
S2,5        if(mode.equals("dynamically_selected_key ")){
S2,6            String key_type= neededKey.getKeyProperty("key_type");
S2,7            String PKI= neededKey.getKeyProperty("PKI");
S2,8            String key_issuer= neededKey.getKeyProperty("key_issuer");
S2,9            String message_to_user = neededKey.getMessageToUser();
```

```
                //Use the above information to interact with the user to obtain the correct key
S2,10           byte[] obtainedKey=USER_INTERFACE_GET_KEY(
                                                key_type,PKI,key_issuer, message_to_user);
S2,11           neededKey.setup(obtainedKey);


              }
S2,12         if(mode.equals("static_key")){
S2,13           String key_type= neededKey.getKeyProperty("key_type");
S2,14           String PKI= neededKey.getKeyProperty("PKI");
S2,15           String key_issuer= neededKey.getKeyProperty("key_issuer");
S2,16           String key_subject = neededKey. getKeyProperty("key_subject");
S2,17           String message_to_user = neededKey.getMessageToUser();


                //Use the above information to interact with the user to obtain the correct key
S2,18           byte[] obtainedKey=USER_INTERFACE_GET_KEY(
                                        key_type,PKI,key_issuer,key_subject, message_to_user);
S2,19           neededKey.setup(obtainedKey);
              }
            }


          // Secure X to obtain Xs
S3,1        WSSLError errorMessage= new WSSLError();
S3,2        Node securedRequestSOAP = wsslTransform.secureSOAP(requestSOAP,
                                                                errorMessage);


          // Send Xs to server-side proxy
S4,1        Network.Send_to_Server-Side_Proxy(securedRequestSOAP);


          // Receive Rs from server-side proxy and unsecure Rs to obtain R
S5,1        Node securedResponseSOAP= Network.Receive_From_Server-Side_Proxy();
```

```
S5,2    WSSLError errorMessage= new WSSLError();
S5,3    Node unsecuredSOAP = wsslTransform.unsecureSOAP(securedResponseSOAP,
                                                        errorMessage);

      // handle the result of the verification
S6,1    Vector signatureVerfications=wsslTransform.getSignatureVerification();
S6,2    for(int i=0; i< sigVerfications.size();i++) {
S6,3      SignatureVerification signatureVerification = (SignatureVerification)
                                                        sigVerfications.get(i);
S6,4      boolean result = signatureVerification.getResult();
S6,5      If (result==true){
S6,6          //Code to store the result of the verification .
S6,7      }else{
S6,8          //Code to handle the situation in which the signature is incorrect.
          }
        }

      // Send R to service requester
S7,1    Network.Send_to_Service_Requester(unsecuredSOAP);
```

Figure 30: Code of the client-side proxy for securing a SOAP message

5.3. The server-side proxy

The code for unsecuring a SOAP message in the server-side proxy is provided in Figure 31, which is very similar to the code for securing a SOAP message shown in Figure 30. The differences are that, first, the second parameter should be "WSSLTransform.PROVIDER_MODE" during the instantiation of the WSSLTransform object in line S1,2. This sets up the unsecure SOAP messages. The second difference relates to the methods used to obtain the required keys. The server-side proxy does not

to set up dynamically selected keys. However, it always has to handle keys whose mode is "key_applied_to_signature" (see lines S2,12 to S2,18). The WSSL API supports obtaining related information needed to verify the authentication policy.

```
        //Obtain an instance of WSSLTransform class.
        // (1) "WSSLDoc.xml" is the WSSL document shown in Figure 3.
        // (2) The parameter WSSLTransform.PROVIDER_MODE is used for the server-side
        //       proxy.
S1,1    WSSLTransform wsslTransform =
S1,2          new WSSLTransform("WSSLDoc.xml",WSSLTransform.PROVIDER_MODE);


        // Receive Xs from client-side proxy
S1,3    Node securedRequestSOAP = Nework.Receive_From_Client-Side_Proxy();
S1,4    wsslTransform.readSOAPMessage(securedRequestSOAP);


        //Obtain the needed keys automatically
S2,1     Vector neededKeys = wsslTransform.getNeededKey();
S2,2     for(int i=0;i<neededKeys.size();i++) {
S2,3        NeededKey neededKey = (NeededKey) neededKeys.get(i);
S2,4        String mode = neededKey.getKeyMode();
S2,5        if(mode.equals("static_key")){
S2,6              String key_type= neededKey.getKeyProperty("key_type");
S2,7              String PKI= neededKey.getKeyProperty("PKI");
S2,8              String key_issuer= neededKey.getKeyProperty("key_issuer");
S2,9              String key_subject= neededKey.getKeyProperty("key_subject");


                  //Use the above information to obtain the correct key automatically to decrypt
S2,10             byte[] obtainedKey=GET_PRIVATE_KEY(
                                                key_type,PKI,ket_issuer,key_subject);
```

```
S2,11          neededKey.setup(obtainedKey);
               }
S2,12          if(mode.equals("key_applied_to_signature")){
S2,13              //Check the owner of the key is authorized to make the transaction
S2,14              String key_type= neededKey.getKeyProperty("key_type");
S2,15              String PKI= neededKey.getKeyProperty("PKI");
S2,16              String key_issuer= neededKey.getKeyProperty("key_issuer");
S2,17              String key_subject= neededKey.getKeyProperty("key_subject");

S2,18          If (!CHECK_AUTHENTICATION_POLICY(key_type,PKI,ket_issuer,key_subject))
                   Nework.Send_Error_Message_TO_Client-Side_Proxy();;
               }
           }

       // Unsecure Xs to obtain X
S3,1       WSSLError errorMessage= new WSSLError();
S3,2       Node unsecuredSOAP = wsslTransform.unsecureSOAP(securedRequestSOAP,
                                                           errorMessage);

       // handle the result of the verification
S4,1       Vector signatureVerfications=wsslTransform.getSignatureVerification();
S4,2       for(int i=0; i< sigVerfications.size();i++) {
S4,3           SignatureVerification signatureVerification=(SignatureVerification)
                                                           sigVerfications.get(i);
S4,4           boolean result = signatureVerification.getResult();
S4,5           If (result==true){
S4,6               //Code to store the result of the verification .
S4,7           }else{
S4,8               //Code to handle the situation in which the signature is incorrect.
               }
```

	}
	// Send X to service provider
S5,1	Network.Send_to_Service_Provider(unsecuredSOAP);
	// Receive *R* from service provider and secure *R* to obtain R_s
S6,1	Node responseSOAP = Nework.Receiver_From_Service_Provider();
S6,2	WSSLError errorMessage= new WSSLError();
S6,3	Node securedResponseSOAP = wsslTransform.secureSOAP(responseSOAP, errorMessage);
	// Send R_s to client-side proxy
S7,1	Network.Send_to_Client-Side_Proxy(securedResponseSOAP);

Figure 31: Code of the server-side proxy for unsecuring a SOAP message