Analysis of Web Services Secure Conversation with Formal Methods *

Llanos Tobarra, Diego Cazorla, Fernando Cuartero and Gregorio Díaz
Escuela Politecnica Superior de Albacete
University of Castilla-La Mancha
Campus Universitario s/n
Albacete (Spain), 02071
{*mtobarra,dcazorla,fernando, gregorio*}*@dsi.uclm.es*

Abstract

Web Services Secure Conversation extends Web Services Trust to provide mechanisms for establishing security contexts*. A security context is an abstract concept that refers an authenticated state where the participants have derived secure session keys for multiples request/response exchanges. In this paper we model this protocol with the HLPSL language and we analyse it with the AVISPA toolbox.*

1. Introduction

Web Services technologies offer some advantages to e-commerce business like dynamic discovering and composition of Web Services, platform and programming-language independence, the widespread adoption of Web services mechanisms, etc But one of the main issues related to web services is that they are exposed to new security attacks. A proposal security architecture could be found in [7]. That document describes a collection of specifications for describing a Web Services security model.

The basic protocol of that architecture is Web Services Security [13]. WS-Security is a protocol that extends SOAP[17] in order to implement *message integrity* and *confidentiality*. WS-Security builds on the SOAP specification, structuring the use of essential security capabilities. It is based on XML Encryption [15] and XML Signature [16]. WS-Security uses binary tokens for authentication, digital signatures for integrity and content-level encryption for confidentiality. It has been proved that using only WS-Security is inefficient to secure Web Services ([11, 14, 3]).

Web Services Secure Conversation[8] extends WS-Security in order to introduce *security context*. A security context is a state where the participants of the context have been authenticated and have negotiated session keys. It is represented by a new type of security token that is issued and propagated using the mechanisms defined by WS-Trust [9]. Thus, WS-Secure Conversation describes how security context are established, renewed and canceled. It also describes how derived keys are computed and exchanged among the participants.

In line with the development of e-commerce and security protocols, some techniques have also been developed to model a system and check its properties on it. One of the most promising techniques of this type is *model checking*. Model checking [4] is a formal methods based technique for verifying finite-state-concurrent systems, and has been implemented in several tools. One of the main advantages of this technique is that it is automatic and allows us to see if a system works properly or not. In case the system does not work as expected, the model checking tool provides a trace that leads to the source of the error.

In this paper, we present a formal verification of WS-Secure Conversation using AVISPA (Automated Validation of Internet Security Protocol an Applications) framework [1]. AVISPA provides a high-level formal language HLPSL for specifying protocols and their security properties. Once we have specified the model of the system AVISPA translates it to an intermediate format IF. This is the input of several backends that are integrated into AVISPA framework: SATMC, OFMC, Cl-Atse and TA4SP. Besides, only one model is specified although it is analysed with the four backends. AVISPA also offers a graphical interface SPAN [6] that helps the specifying task.

Several papers [10, 5, 12] can be found which use formal methods to analyse Web Services, and in which formal methods are used to analyse the behavior and the performance of web services standards; these analyses help designers to correct any errors. More related to our work, however, is [2]. In that article, they analyse WS-Trust and WS-Secure Conversation with TulaFale tool [3]. This arti-

*This work has been supported by the spanish government (cofinanced by FEDER founds) with the project "Application of Formal Methods to Web Services" (TIN2006-15578-C02-02), and the JCCM project "Application of Formal Methods to the design and the analysis of Web Services and E-commerce"(PAC06-0008-6995)

cle is concentrated on extending the Tulafalse semantics for WS-Trust and WS-Secure Conversation. While we analyse several bindings proposed in [8] they are only focused on establishing the security context and they offer basic primitives for specific use cases.

This paper is organized as follows. In Section 2 WS-Secure Conversation is described in more detail. In Section 3 we build the AVISPA specification that models the system, describe the security requirements that the system should achieve, and verify the model against these requirements. Finally, in Section 4 we give our conclusions and some outlines for future work.

2. Web Services Secure Conversation

2.1. Security Context and Session Keys

A *security context* is an abstract concept that represents a state where the participants have been authenticated and they have computed session keys associated to the context.

The abstract concept of security context is represented by means of a token. In fig. 1 we can see the structure of a *security context token* (SCT). The `Identifier` element contains an unique URI which identifies the token. It allows other elements to refer the context. When a SCT is referred in a `KeyInfo` element or in the `Security` header, the key related to the context should be used to encrypt data. Sometimes the security context is updated and the participants could generate new session keys. So `Instance` element should be used to indicate which is the current key.

As we mentioned before, when participants establish a security context, they also exchange some key material in order to generate session keys for encrypting or computing digital signatures. Although two participants use the same key material, they can generate different keys. Thus the first time a security participant uses a new context derived key, this entity indicates it with a `DerivedKeyToken` (see fig. 1). In order to generate keys, participants should agree on some parameters(algorithm, key size, etc) and it is recommended to do it through policies. But they can also use some constant labels or nonces. They indicate these elements with the `DerivedKeyToken`. In a message we can find several `DerivedKeyToken`, one for each generated key that is used in the message.

```
<soap:Envelope>
  <soap:Header>
        ....
    <wsse:Security>
      ....
      <wssc:SecurityContextToken
        wsu:Id="SecurityToken-5b3f409a-4301f2461b3a">
        <wssc:Identifier>
          urn:uuid:d7648de7-acac-d879c6112a76
        </wssc:Identifier>
        <wsc:Instance>
          1
        </wsc:Instance>
       </wssc:SecurityContextToken>
      <wssc:DerivedKeyToken
        wsu:Id="SecurityToken-f20426d3-a4af3ccfb04a"
        Algorithm="http://.../sc/dk/p_sha1">
        <wsse:SecurityTokenReference>
          <wsse:Reference
           URI="#SecurityToken-5b3f409a-4301f2461b3a"
           ValueType="http://../sc/sct" />
        </wsse:SecurityTokenReference>
        <wssc:Generation>0</wssc:Generation>
        <wssc:Length>24</wssc:Length>
        <wssc:Label>
           WS-SecureConversation
           WS-SecureConversation
        </wssc:Label>
        <wssc:Nonce>
           qHHrc2P3PPx3j0otPrJ+mw==
         </wssc:Nonce>
      </wssc:DerivedKeyToken>
          ..............
    </wsse:Security>
  </soap:Header>
  <soap:Body>
    <xenc:EncryptedData>
      <xenc:EncryptionMethod
         Algorithm="http://.../xmlenc#aes256-cbc" />
      <KeyInfo xmlns="http://.../xmldsig#">
        <wsse:SecurityTokenReference>
          <wsse:Reference
            URI="#SecurityToken-f20426d3-a4af3ccfb04a"
            ValueType="http://.../dk" />
        </wsse:SecurityTokenReference>
      </KeyInfo>
      <xenc:CipherData>
        <xenc:CipherValue>
          ...
        </xenc:CipherValue>
      </xenc:CipherData>
    </xenc:EncryptedData>
  </soap:Body>
</soap:Envelope>
```

Figure 1. Examples of use of security context token and derived key token elements

2.2. Establishing a security context

Although in [8] they mention three ways of establishing a security context, we only focus on one of these three options. We consider that a initiator agent request a security context to a *Security Token Service* (STS) with a `RequestSecurityToken` message (see fig. 2). This agent could include some secret data that the SCS should take into account when it creates the security context. The initiator must authenticate himself with a security token, usually a `UsernameToken`. This token is included in the `Security` header when the request is secured with WS-Security. The Security Token Service returns a `RequestSecurityTokenResponse`. This response contains or refers to the Security Context Token and a `RequestProofToken`, which points to the key material associated to the security context. Both messages must be secure with WS-Security.

The generated security context token is distributed to the rest of the participants through a propagation `RequestSecurityTokenResponse` message.

2.3. Renewal of a security context

A security context has associated a set of claims and a lifetime when is established. But, sometimes, a participant desires to update the associated claims or to extend the validity time of a security context. WS-Secure Conversation

A.Request Security Context Token :

```
<soap:Envelope>
  <soap:Body>
    <wst:RequestSecurityToken>
      <wst:TokenType>
          http://.../sct
      </wst:TokenType>
      <wst:RequestType>
          http://.../trust/Issue
      </wst:RequestType>
      <wst:Entropy>
        <wst:BinarySecret
           Type="http://.../trust/SymmetricKey">
            tzYesb28omI6ogxfUSQTmA==
        </wst:BinarySecret>
      </wst:Entropy>
      <wst:Lifetime>
        <wsu:Expires>
           2006-12-04T21:08:06Z
        </wsu:Expires>
      </wst:Lifetime>
    </wst:RequestSecurityToken>
  </soap:Body>
```

B. Request Security Context Token Response:

```
<soap:Envelope>
  <soap:Header />
  <soap:Body>
    <wst:RequestSecurityTokenResponse>
      <wst:KeySize>256</wst:KeySize>
      <wst:RequestedSecurityToken>
        <wssc:SecurityContextToken>
          <wssc:Identifier>
              urn:uuid:d7648de7-acac-d879c6112a76
          </wssc:Identifier>
        </wssc:SecurityContextToken>
      </wst:RequestedSecurityToken>
      <wst:RequestedProofToken>
        <wst:ComputedKey>
          http://.../trust/CK/PSHA1
        </wst:ComputedKey>
      </wst:RequestedProofToken>
      <wst:Entropy>
        <wst:BinarySecret Type="http://.../SymmetricKey">
          XNFzpWGEqXngsKgq/f+vtA==
        </wst:BinarySecret>
      </wst:Entropy>
      <wst:Lifetime>
        <wsu:Expires>
          2006-12-04T21:08:07Z
        </wsu:Expires>
      </wst:Lifetime>
    </wst:RequestSecurityTokenResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 2. Examples of request and response SOAP messages for establishing a security context token

defines two bindings for these proposes.

A participant could add new claims by means of the Amending binding. This way, the context is ”amended”. That agent could send an amended SCT in a `RequestSecurityTokenResponse` without previous related request. The additional claims of the security context must be included inside new security tokens at this response message.

The Renew binding allows participants to extend the lifetime of an SCT. This binding is based on WS-Trust Renew binding. A context participant sends a request to the Security Token Server, which previously generated the SCT. This request must include a proof of possession of the SCT associated keys. Besides, the participant must reauthenticate the claims associated to the security context. They could exchange new key material during the renew protocol, but this secret date must not be encrypted with the previous context keys.

2.4. Cancellation of a security context

A security context could be canceled before it expires. If a security context is canceled, it can not be renewed nor amended. If a participant wants to cancel a context it will send a `RequestSecurityToken` (see fig. 3) message. `CancelTarget` element that refers to the SCT is included inside this request message. The agent must include a proof of possession of the security context key as well.

Once the cancel request is sent, the participant must consider the security context canceled, even if it does not receive any response. The response message is only informative.

A. Cancel request:

```
<soap:Envelope>
  <soap:Body>
    <wst:RequestSecurityToken>
      <wst:RequestType>
          http://.../trust/Cancel
      </wst:RequestType>
      <wst:CancelTarget>
        <wsse:SecurityTokenReference>
          <wsse:Reference
            URI="urn:uuid:d7648de7-acac-
                d879c6112a76"
            ValueType="http://../sct" />
        </wsse:SecurityTokenReference>
      </wst:CancelTarget>
    </wst:RequestSecurityToken>
  </soap:Body>
</soap:Envelope>
```

B. Cancel Response:

```
<soap:Envelope>
  <soap:Header />
  <soap:Body>
    <wst:RequestSecurityTokenResponse>
      <wst:RequestedTokenCancelled />
    </wst:RequestSecurityTokenResponse>
  </soap:Body>
</soap:Envelope>
```

Figure 3. Examples of request/response message for canceling a security context token

3. Analysis of WS Secure Conversation

In this section, we will analyse the main WS-Secure Conversation bindings described in the previous section with the AVISPA toolbox. These bindings allow to establish, to renew and to cancel a security context among two parties. In addition we analyse how two endpoints can exchange data using the derived keys associated to an SCT.

In our model, we will consider an intruder who can per-

form the following actions:

- Overhear and intercept all the messages over the network.
- Modify the messages. The intruder can add bytes, delete bytes or change the value of several bytes.
- Generate new messages using its initial knowledge or parts of the overheard messages.
- Send a new or captured message to another entity in the system.

We will assume that the intruder cannot perform any cryptanalysis.

Our analysis is focused on WS-Secure Conversation, however, other Web Services specifications are taken into account. In particular, it is considered that Web Services Addressing and WS-Security are applied in combination with WS-Secure Conversation.

Each message contains a `MessageId` header, a `RelatesTo` header, a `To` header and a `ReplyTo` header. The `MessageId` elements is a unique random identifier for the message. In this way the server could check if it has already received the same message . Message identifiers are represented by nonces, called M_i in our model, where i is the sequence number of the message. A response message also includes a `ReplyTo` element that indicates the correlated request message identifier. So the correlation between both messages could be checked by the requester. The `To` element and `ReplayTo` element indicate the intended receiver and the sender, which are agent variables in our models.

We consider that every message is secured by WS-Security. Key material and secret data are encrypted. In addition, the main elements are digitally signed. It is also recommended to include timestamps. However, AVISPA does not offer time semantics, so we can not perform time analysis.

3.1. Establishing a security context

First, we analyse the binding for establishing security context between a client and a Security Token Service (Fig. 4). In our model, there are two types of agents: clients and servers. The client role represents an initiator participant that request a new security context token. The server role represents the Security Token Server.

As we mentioned before, the Establish binding is composed by two SOAP messages. The first one is the `RequestSecurityToken`. The M_1, C and S variables represent the WS-Addressing `MessageID`, `ReplayTo` y `To` elements. A `UsernameToken` is represented by $\{C.Pwd\}$. This token is ciphered with the Security Token Service public key PKS. The message body is composed of two elements. A constant *request* which indicates the objective of the message. And some secret data represented by the nonce $SecretC$. This variable is also ciphered by the SCS public key PKS in order to keep it in secret.

The sender digitally sign main variables of the message: M_1, C, S, $request$ and $SecretC$. Sha is a function that computes the digest of its parameters.

The second message is the `RequestSecurityTokenResponse`. It includes the previously mentioned WS-Addressing elements in this message. A digital signature of main elements in the message $M_2, M_1, response, SCT$ and $SecrectS$ is computed by the server. The key material $SecretS$ is cipher with the client public key PKS as well.

Avispa input language, HLPSL, has semantics for sets. It has basic set operations to insert elements, to delete elements and to find elements in set. Inside the local state of the server we define a set called $Context$. Each element of this set is a concatenation of an agent identifier, its password, an SCT variable, and the two secret elements $SecretC$ and $SecretS$. Once the Token Services generates a new SCT and new key material, it records this information with the `UsernameToken` in the $Context$ set. This way the server could retrieve information related to the context.

We verify the following properties:

- The secrecy of the variables *SecretC, SecretS* y *Pwd*. It is evident these values must secret in order to guarantee that the derived session keys are secret.
- The client and the server agree on the value of Pwd and SCT. Thus, the client is authenticated at the server by Pwd and the server generates shared SCT with the client.

We find the following attack on this specification:

1. C $\rightarrow$ I: M_1.C.S.$\{$C.Pwd$\}_{pks}$. $\{$sha(M_1. C. S. request. SecretC)$\}_{pks}$. request. $\{$SecretC$\}_{pks}$
2. I $\rightarrow$ S: M_1.C.S.$\{$C.FalsePwd$\}_{pks}$. $\{$sha(M_1. C. S. request. SecretC)$\}_{pks}$. request. $\{$SecretC$\}_{pks}$
3. S $\rightarrow$ I: $M_2.M_1$.C. $\{$sha(M2. M1. C. response. SCT. SecretS)$\}_{inv(pks)}$. response.SCT. $\{$SecretS$\}_{pkc}$

As we can observe the client ciphers its username and the corresponding password with the public key of the server. An intruder could generate a false password. It could replace the username token for a false username token as well, because it knows the public key of the server. This attack could be avoided if UsernameToken is included in the computation of the digital signature of the message. We do this modification to our model and we do not find any attack with Avispa back-ends.

3.2. Exchange data

Once two endpoint have established a secure context, they can use the context derived keys. In [8] they propose the following formula $P\_SHA1(secret, label + nonce)$ to compute a session key, where *secret* is the exchanged key material, *label* is agreed with endpoints policies and *nonce* is some random data exchanged by the endpoints. The nonce is included in the `DerivedKeyToken` when the derived key is used. The derivation key process is represented by a function called $DerivedKey$. It accepts as parameters the secret data exchanged in the establishing protocol, *SecretC* and *SecretS*, and a nonce, *Nc* for the client and *Ns* for the web service server. It returns a symmetric key. The labels are ignored because they are constants.

In this model we analyse the exchange of data using a security context (Fig. 4). A client requests a web service at the server and it responses. At this moment, they have established a security context *SCT* successfully. Each endpoint computes only one session key. The security token identifier SCT is included in all the messages. The `DerivedKeyToken` is represented by a nonce, Nc or Ns, depending on the agent. As in the previous model, the server has a $Context$ set where it stores the security context SCT, their associated key material $SecretC$ and $SecretS$, and the UsernameToken $\{C.Pwd\}$. When it receives the request it checks if the security context is related to that agent and retrieves the key material from that set.

We verify that the request and the response elements are secret with AVISPA backends and we do not find any attack.

3.3. Renewal of a security context token

In Fig. 4 we can see the SPAN representation of our model for the Renew binding. The request message is almost the same message that for the establish request message. But it includes a new $Lifetime$ element. This variable indicates the new validity period of time. When the server receives this message, it searches in the *Context* set for the security context information. It checks if the username token is correct and that there is a previous security context SCT. It retrieves too the key material $SecretS$ and $SecretC$. If all the verifications are correct, it generates new key material. It returns a `RequestSecurityTokenResponse` message with the new $Lifetime$ value. Otherwise the protocol finishes.

We verify the following properties:

- The secrecy of the key material $SecretS$ and $SecretC$, as before.
- The client and the server agree on the values of SCT and $Lifetime$, thus they authenticate each other with these values.

We do not find any attack.

3.4. Cancel a security context

Last, we analyse the Cancel binding (Fig. 4). We consider two session keys, KC generated by the client and KS generated by the server.

We verify that the client and the server agree on the security context SCT, and they exchage the cancel request and the cancel response. We find the following attack:

1. C $\rightarrow$ I: M_1. C. S. cancel. sct. $\{$sha(M1. C. S. cancel. sct)$\}_{kc}$
2. I $\rightarrow$ S: M_i. I. S. cancel. badsct. $\{$sha(M_i. I. S. cancel. badsct)$\}_{ki}$
3. S $\rightarrow$ I: M_2. M_i. I. cancelresponse. $\{$sha(M_2. M_i. I. cancelresponse)$\}_{ki}$

The intruder intercepts the cancel message and it sends its own cancel message. Thus the server ignores that the SCT context is canceled and it will accept the following messages with this context until it expires. In previous sections, we verified that the session keys are secure. As a consequence of that, the encrypted data is kept as a secret and the intruder can not guess it. Message identifiers are taken into account in order to avoid replay attacks. If these message identifiers are omitted, the intruder could replay any previous request message before the context expires.

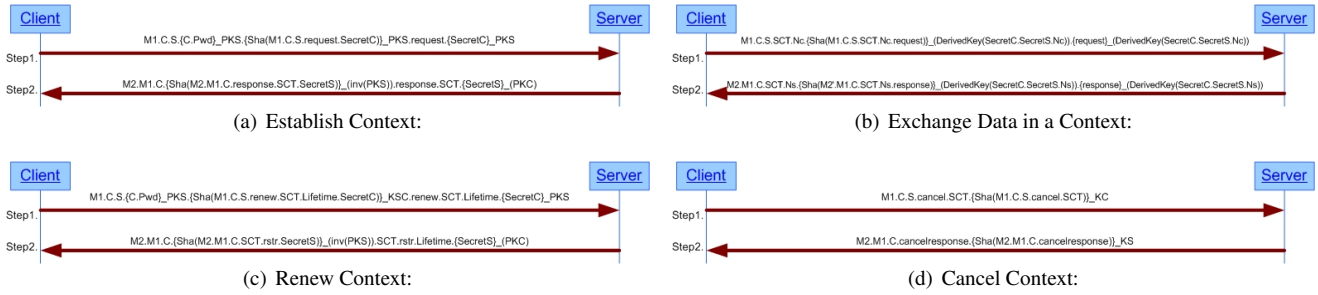
4. Conclusions

In this paper, we have analyzed the main mechanisms of management of the security context tokens proposed in [8]. First, we have focused on the establishment of the security context token. In this analysis we found an attack that demonstrates the importance of combining WS-Secure Conversation with WS-Security in an appropriate way. So that the integrity of all the elements of the message should be guaranteed.

We have also analyzed what it happens once the context is generated how they guarantee the confidentiality and integrity of the message. We did not found any attack in this case.

Together with the establishment of the context, the renovation of the token has been analyzed. This binding allows to enlarge the period of validity of a security context token. We have verified that, under the configuration that we have selected, it is safe.

Lastly, the cancellation of the security context token has been verified. In this case we have found an attack. An intruder can eliminate the cancellation request and it avoids



(a) Establish Context:

(b) Exchange Data in a Context:

(c) Renew Context:

(d) Cancel Context:

Figure 4. Representation of the protocol models generated by SPAN

that it arrives to the server. So, the server could accept false messages while the security context is no longer valid.

As conclusion we can highlight that WS-Secure Conversation allows to establish security context in combination with the appropriate mechanisms of other protocols, such as WS-Addressing and WS-Security. This security context can guarantee the security in an exchange of messages under these group of specifications.

Our future work is concerned with building a model of the system which allows us to analyse all the cases for establishing a secure context without Security Token Service. We also would like to analyse the amending binding.

References

[1] A. Armando, D. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuellar, P. Hankes Drielsma, P.-C. Heám, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch, J. Santiago, M. Turuani, L. Viganò, and L. Vigneron. The avispa tool for the automated validation of internet security protocols and applications. In K. Etessami and S. K. Rajamani, editors, *Proceedings of the 17th International Conference on Computer Aided Verification (CAV'05)*, volume 3576 of *LNCS*. Springer, 2005. Available at http://www.avispa-project.org/publications.html.

[2] K. Bhargavan, C. Fournet, A. Gordon, and R.Corin. *"Secure Sessions for Web Services"*, August 2004. At http://research.microsoft.com/projects/samoa/secure-sessions-with-scripts.pdf.

[3] K. Bhargavan, C. Fournet, A. D. Gordon, and R. Pucella. Tulafale: A security tool for web services. In *Formal Methods for Components and Objects: Second International Symposium, FMCO 2003*, volume 3188 of *Lecture Notes in Computer Science*, pages 197 – 222. Springer, November 2003.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[5] G.Díaz, J. Pardo, E. Cambronero, V. Valero, and F. Cuartero. *"Verification of Web Services with Timed Automata"*. *1st Int'l Workshop on Automated Specification and Verification of Web Sites*, 2005.

[6] Y. Glouche and T. Genet. Span, a security protocol animator for avispa. version 1.0. At http://www.irisa.fr/lande/genet/span/, September 2006.

[7] IBM and Microsoft. Security in a web services world: a proposed architecture and roadmap. At http://www-128.ibm.com/developerworks/library/specification/ws-secmap/, April 2002.

[8] IBM, B. Systems, Microsoft, C. Associates, Actional, VeriSign, L. . Technologies, Oblix, O. Technologies, P. Identity, Reactivity, and R. Security. Web services secure conversation language. At http://www-128.ibm.com/developerworks/library/specification/ws-secon/, February 2005.

[9] IBM, B. Systems, Microsoft, C. Associates, Actional, VeriSign, L. . Technologies, Oblix, O. Technologies, P. Identity, Reactivity, and R. Security. Web services trust language (ws-trust). At http://www-128.ibm.com/ developerworks/webservices/ library/specification /ws-trust/, February 2005.

[10] J. Johnson, D. Langworthy, L. Lamport, and F.H.Vogt. *"Formal Specification of a Web Services Protocol"*. *Electronic Notes in Theoretical Computer Science 105 (2004) 147-158*, February 2004.

[11] E. Kleiner and A.W.Roscoe. *"Web Services Security: a preliminary study using Casper and FDR"*. *Proceedings of the Workshop on Automated Reasoning for Security Protocol Analysis (ARSPA 2004)*, June 2004.

[12] S. Nakijima. *"On verifying Web Services Flows"*. *Proc. SAINT 2002 Workshop*, pages 223–224, January 2002.

[13] OASIS. Web services security: Soap message security. At http://www.oasis-open.org/specs/index.php#wssv1.0, March 2004.

[14] M. L. Tobarra, D. Cazorla, F. Cuartero, and G. Díaz. Application of formal methods to the analysis of web services security. In M. Bravetti, L. Kloul, and G. Zavattaro, editors, *Formal Techniques for Computer Systems and Business Processes, European Performance Engineering Workshop, EPEW 2005 and International Workshop on Web Services and Formal Methods, WS-FM 2005, Versailles, France, September 1-3, 2005, Proceedings*, volume 3670 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2005.

[15] W3C. Xml encryption syntax and processing. http://www.w3.org/TR/xmlenc-core/, December 2002.

[16] W3C. Xml-signature syntax and processing. At http://www.w3.org/TR/xmldsig-core/, February 2002.

[17] W3C. Soap 1.2 part 1: Messaging framework. http://www.w3.org/TR/soap12-part1/, June 2003. W3C Recommendation.